Alma Mater Studiorum - University of Bologna

DEIS - DEPARTMENT OF ELECTRONICS, COMPUTER SCIENCE AND SYSTEMS
PhD Course in Electronics, Computer Science and Telecommunications

XXI cycle – scientific-disciplinary sector ING/INF05

Specification and Verification of Declarative Open Interaction Models

A Logic-Based Framework

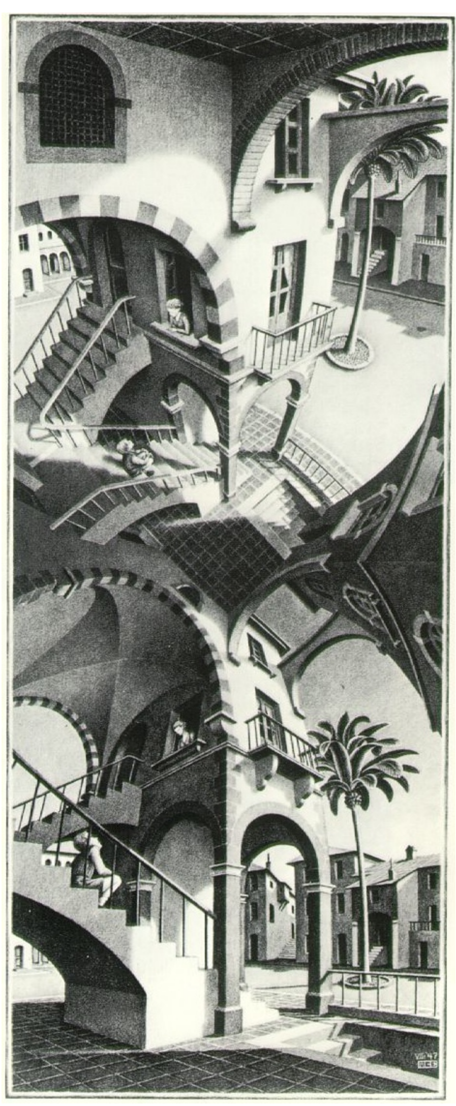
Candidate:
Marco Montali

Supervisor:
Prof. Paola Mello

PhD Course Coordinator:
Prof. Paola Mello

Final Examination Year 2009

Dedicated to Valentina and Giulia,
and to Evita and Micol.



M. C. Escher, *Up and Down*®(1947).

ABSTRACT

The advent of distributed and heterogeneous systems has laid the foundation for the birth of new architectural paradigms, in which many separated and autonomous entities collaborate and interact to the aim of achieving complex strategic goals, impossible to be accomplished on their own. A non exhaustive list of systems targeted by such paradigms includes Business Process Management, Clinical Guidelines and Careflow Protocols, Service-Oriented and Multi-Agent Systems.

It is largely recognized that engineering these systems requires novel modeling techniques. In particular, many authors are claiming that an open, declarative perspective is needed to complement the closed, procedural nature of the state of the art specification languages. For example, the ConDec language has been recently proposed to target the declarative and open specification of Business Processes, overcoming the over-specification and over-constraining issues of classical procedural approaches. On the one hand, the success of such novel modeling languages strongly depends on their usability by non-IT savvy: they must provide an appealing, intuitive graphical front-end. On the other hand, they must be prone to verification, in order to guarantee the trustworthiness and reliability of the developed model, as well as to ensure that the actual executions of the system effectively comply with it.

In this dissertation, we claim that Computational Logic is a suitable framework for dealing with the specification, verification, execution, monitoring and analysis of these systems. We propose to adopt an extended version of the ConDec language for specifying interaction models with a declarative, open flavor. We show how all the (extended) ConDec constructs can be automatically translated to the CLIMB Computational Logic-based language, and illustrate how its corresponding reasoning techniques can be successfully exploited to provide support and verification capabilities along the whole life cycle of the targeted systems.

SOMMARIO

L'avvento dei sistemi distribuiti ed eterogenei ha gettato le basi per la nascita di nuovi paradigmi architetturali, nell'ambito dei quali entità separate ed autonome collaborano ed interagiscono al fine di raggiungere obiettivi strategici complessi, impossibili da realizzarsi in solitudine. Una lista non esaustiva di alcuni sistemi affrontati mediante tali paradigmi include la gestione dei processi aziendali, le linee guida in campo medico e i protocolli di cura, i sistemi orientati ai servizi e agli agenti.

L'importanza di fornire nuovi strumenti di modellazione per l'ingegnerizzazione di questi sistemi è largamente condivisa in letteratura. In particolare, vari autori affermano la necessità di affiancare una prospettiva aperta e dichiarativa alla visione chiusa e procedurale degli strumenti di modellazione presenti nello stato dell'arte. Ad esempio, il linguaggio ConDec è stato recentemente proposto per affrontare la modellazione dei processi aziendali superando i limiti dei classici approcci procedurali, i quali impongono di sovra-specificare e sovra-vincolare i modelli. Da un lato, il successo di questi nuovi linguaggi di modellazione dipende fortemente dalla loro usabilità, soprattutto da parte di utenti sprovvisti di doti tecniche: tali linguaggi devono supportare una modalità di specifica grafica e di semplice comprensione. Dall'altra parte, deve essere possibile verificare questi linguaggi, al fine di garantirne l'affidabilità e la correttezza, così come di assicurare che le esecuzioni reali del sistema ne stiano davvero seguendo le prescrizioni.

La tesi sostenuta in questa dissertazione è che la logica computazionale è un valido strumento per trattare la specifica, la verifica, l'esecuzione, il monitoraggio e l'analisi di questi sistemi. Si propone di adottare una versione estesa del linguaggio ConDec per specificare i modelli di interazione seguendo un approccio dichiarativo ed aperto. Si mostra come tutti i costrutti messi a disposizione da ConDec (esteso) possano essere automaticamente tradotti nel linguaggio CLIMB, basato su logica computazionale. Infine, si illustra come le tecniche di ragionamento associate a CLIMB possano essere impiegate con successo per fornire supporto e capacità di verifica lungo tutto il ciclo di vita di tali sistemi.

PUBLICATIONS OF THE AUTHOR

[1] F. Chesani, P. Mello, M. Montali, S. Storari, and P. Torroni. On the Integration of Declarative Choreographies and Commitment-based Agent Societies into the SCIFF Logic Programming Framework. *Multiagent and Grid Systems, Special Issue on Agents, Web Services and Ontologies: Integrated Methodologies*, 6(2), 2010.

[2] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative Specification and Verification of Service Choreographies. *ACM Transactions on the Web - Under the second round of reviews*, 2009.

[3] M. Montali, F. Chesani, P. Mello, and P. Torroni. Verification of Choreographies During Execution Using the Reactive Event Calculus. In *Proceedings of the 5th International Workshop on Web Service and Formal Methods (WS-FM2008)*, pages 129–140, 2009.

[4] F. Chesani, P. Mello, M. Montali, and P. Torroni. Modeling and Verification of Business Processes and Choreographies in ALP. *Il Milione - Viaggio nella Logica Computazionale in Italia, Special Issue of Intelligenza Artificiale dedicated to Prof. Alberto Martelli*, 2009 (expected). Accepted for publication.

[5] F. Chesani, P. Mello, M. Montali, and P. Torroni. Ontological Reasoning and Abductive Logic Programming for Service Discovery and Contracting. In A. Gangemi, J. Keizer, V. Presutti, and H. Stoermer, editors, *Proceedings of the 5th Workshop on Semantic Web Applications and Perspectives (SWAP2008)*, volume 429 of *CEUR Workshop Proceedings*, 2009.

[6] F. Chesani, P. Mello, M. Montali, F. Riguzzi, M. Sebastianis, and S. Storari. Checking compliance of execution traces to business rules. In *Proceedings of BPM 2008 Workshops*, volume 17 of *Lecture Notes in Business Information Processing*. Springer Verlag, 2009.

[7] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Exploiting Inductive Logic Programming Techniques for Declarative Process Mining. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC), Special Issue on Concurrency in Process-Aware Information Systems*, 5460:278–295, 2009.

[8] M. Baldoni, C. Baroglio, G. Berio, A. Martelli, V. Patti, M.L. Sapino, C. Schifanella, M. Alberti, M. Gavanelli, E. Lamma, F. Riguzzi, S. Storari, F. Chesani, A. Ciampolini, P. Mello, M. Montali, P. Torroni, A. Bottrighi, G. Casella, L. Giordano, V. Gliozzi, V. Mascardi, G. Pozzato, P. Terenziani, and D. Theseider Dupre.

Web Service-Oriented Modeling, Verification and Reasoning Techniques. *Intelligenza Artificiale*, 2009 (expected). Accepted for publication.

[9] M. Montali, P. Mello, F. Chesani, F. Riguzzi, S. Storari, and M. Sebastianis. Compliance Checking of Execution Traces to Business Rules: an Approach Based on Logic Programming. In *23th Convegno Italiano di Logica Computazionale (CILC 2008)*, 2008.

[10] M. Montali, M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verification from Declarative Specifications Using Logic Programming. In M. Garcia De La Banda and E. Pontelli, editors, *24th International Conference on Logic Programming (ICLP)*, number 5366 in Lecture Notes in Computer Science, pages 440–454. Springer Verlag, 2008.

[11] L. Luccarini, G. L. Bragadin, M. Mancini, P. Mello, M. Montali, and D. Sottara. Process Quality Assessment in Automatic Management of Wastewater Treatment Plants Using Formal Verification. In *Proceedings of Simposio Internazionale di Ingegneria Sanitaria Ambientale (SIDISA 2008)*, 2008.

[12] F. Chesani, E. Lamma, P. Mello, M. Montali, S. Storari, P. Baldazzi, and M. Manfredi. Compliance Checking of Cancer-Screening Careflows: an Approach Based on Computational Logic. In A. ten Teije, S. Miksch, and P. Lucas, editors, *Book Chapter of Computer-Based Medical Guidelines and Protocols: a Primer and Current Trends*. IOS Press, 2008.

[13] V. Bryl, P. Mello, M. Montali, P. Torroni, and N. Zannone. B-Tropos: Agent-oriented requirements engineering meets computational logic for declarative business process modeling and verification. In F. Sadri and K. Satoh, editors, *Post-Proceedings of the 8th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-VIII), Revised Selected and Invited Papers*, volume 5056 of *Lecture Notes in Computer Science*, pages 157–176. Springer Verlag, 2008. Based on a Presentation given at the 22th Convegno Italiano di Logica Computazionale (CILC 2007).

[14] M. Montali, F. Chesani, P. Mello, and S. Storari. Testing Careflow Process Execution Conformance by Translating a Graphical Language to Computational Logic. In R. Bellazzi, A. Abu-Hanna, and J. Hunter, editors, *Proceedings of the 11th International Conference on Artificial Intelligence in Medicine (AIME'07)*, volume 4594 of *Lecture Notes in Computer Science*, pages 479–488. Springer Verlag, 2007.

[15] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, Paola Mello, Marco Montali, and Paolo Torroni. Expressing and Verifying Contracts with Abductive Logic Programming. *Electronic Commerce, Special Issue on Contract Architectures and Languages*, 12(4): 9–38, 2008.

[16] E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Learning DecSerFlow Models from Labeled Traces. In *First International Workshop on the Induction of Process Models*, 2007.

[17] E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Inducing Declarative Logic-Based Models from Labeled Traces. In M. Rosemann and M. Dumas, editors, *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 344–359. Springer Verlag, 2007.

[18] F. Chesani, P. Mello, M. Montali, and S. Storari. Agent Societies and Service Choreographies: a Declarative Approach to Specification and Verification. In *International Workshop on Agents, Web-Services and Ontologies: Integrated Methodologies (AWESOME'007)*, 2007.

[19] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. A Rule-Based Approach for Reasoning about Collaboration between Smart Web-Services. In M. Marchiori, J. Z. Pan, and C. de Sainte Marie, editors, *Proceedings of the First International Conference on Web Reasoning and Rule Systems (RR'07)*, volume 4524 of *Lecture Notes in Artificial Intelligence*, pages 279–288. Springer Verlag, 2007.

[20] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Web Service contracting: Specification and Reasoning with SCIFF. In E. Franconi, M. Kifer, and W. May, editors, *Proceedings of the 4th European Semantic Web Conference (ESWC'07)*, volume 4519 of *Lecture Notes in Artificial Intelligence*, pages 68–83. Springer Verlag, 2007.

[21] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, S. Storari, and P. Torroni. A Computational Logic-Based Approach to Verification of IT Systems. In H. G. Hegering and H. Reiser, editors, *Proceedings of the 14th Annual Workshop of HP Software University Association (HP-SUA 2007)*. HP Software University Association, Infonomics-Consulting, 2007.

[22] F. Chesani, P. Mello, M. Montali, M. Alberti, M. Gavanelli, E. Lamma, and S. Storari. Abduction for Specifying and Verifying Web Service Choreographies. In *4th International Workshop on AI for Service Composition (AISC 2006)*, 2006.

[23] F. Chesani, P. De Matteis, P. Mello, M. Montali, and S. Storari. A Framework for Defining and Verifying Clinical Guidelines: a Case Study on Cancer Screening. In F. Esposito, Z. W. Ras, D. Malerba, and G. Semeraro, editors, *Proceedings of the 16th International Symposium on Foundations of Intelligent Systems (ISMIS 2006)*, volume 4203 of *Lecture Notes in Artificial Intelligence*, pages 338–343. Springer Verlag, 2006.

[24] F. Chesani, A. Ciampolini, P. Mello, M. Montali, and S. Storari. Testing Guidelines Conformance by Translating a Graphical Language to Computational Logic. In *Workshop on AI Techniques in Healthcare: Evidence-Based Guidelines and Protocols*, 2006.

[25] M. Alberti, F. Chesani, E. Lamma, M. Gavanelli, P. Mello, M. Montali, and P. Torroni. Policy-Based Reasoning for Smart Web Service Interaction. In A. Polleres, S. Decker, G. Gupta, and J. de Brujin, editors, *Proceedings of the First International Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services*, volume 196 of *CEUR Workshop Proceedings*, pages 87–102, 2006.

[26] M. Alberti, F. Chesani, E. Lamma, M. Gavanelli, P. Mello, M. Montali, S. Storari, and P. Torroni. Computational Logic for the Runtime Verification of Web Service Choreographies: Exploiting the SOCS-SI Tool. In M. Bravetti, M. Nùñez, and G. Zavattaro, editors, *Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM'06)*, volume 4184 of *Lecture Notes in Computer Science*, pages 58–72. Springer Verlag, 2006.

[27] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Policy-Based Reasoning for Smart Web Service Interaction. In G. Tummarello, editor, *Proceedings of the 3rd Workshop on Semantic Web Applications and Perspectives (SWAP'06)*, volume 201 of *CEUR Workshop Proceedings*, 2006.

[28] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and M. Montali. An Abductive Framework for A-Priori Verification of Web Services. In A. Bossi and M. J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 39–50. ACM Press, 2006.

[29] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and M. Montali. A-Priori Verification of Web Services with Abduction. In *21th Convegno Italiano di Logica Computazionale (CILC 2006)*, 2006.

[30] A. Ciampolini, P. Mello, M. Montali, and S. Storari. Using Social Integrity Constraints for On-the-Fly Compliance Verification of Medical Protocols. In A. Tsymbal and P. Cunningham, editors, *In Proceedings of the 18th IEEE Symposium on Computer Based Medical Systems (CBMS'05)*, pages 503–505. IEEE Computer Society, 2005.

[31] M. Alberti, F. Chesani, A. Ciampolini, P. Mello, M. Montali, S. Storari, and P. Torroni. Protocol Specification and Verification by Using Computational Logic. In F. Corradinin, F. de Paoli, E. Merelli, and A. Omicini, editors, *Proceedings of the 6th AI*IA/TABOO Joint Workshop "From Objects to Agents" (WOA 2005): Simulation and Formal Analysis of Complex Systems*, pages 184–192. Pitagora Editrice Bologna, 2005.

We should not be looking for heroes,
we should be looking for good ideas.

— Noam Chomsky

ACKNOWLEDGMENTS

I wish to thank all the people who have supported me during this PhD. First of all, I would like to thank my supervisor, Prof. Paola Mello, for having constantly supported and encouraged my research activity. She guided me through the intriguing world of Computational Logic with her deep knowledge and open mind. Thanks to Federico Chesani, for having shared with me these last three years as a colleague and a friend. Working with him has been a great pleasure... Federico, this dissertation is also dedicated to you! Thanks to Paolo Torroni, for having always encouraged me in pursuing my research and for having shared with me his strong research attitude. I am also greatly indebted to Marco Gavanelli, for his constant support on the SCIFF Framework. A special thank goes to Prof. Wil van der Aalst and Maja Pesic. The fruitful and valuable discussions I had with them during my two visits at TU/e strongly contributed to define the goal of my PhD activity. Without their work on ConDec and Declarative Business Process Management, this dissertation would have never been possible.

I would like to thank also all the people who contributed to my education and research activity. In lexicographic order: Marco Alberti, Matteo Baldoni, Cristina Baroglio, Alessio Bottrighi, Evelina Lamma, Michela Milano, Fabrizio Riguzzi, Davide Sottara, Sergio Storari, Paolo Terenziani and all the people in the LIA aquarium. My research activity has been partially supported by the FIRB Project *TOCAI.IT: tecnologie orientate alla conoscenza per aggregazioni di imprese in internet* and by the PRIN 2005 Project *Linguaggi per la specifica e la verifica di protocolli di interazione fra agenti*.

Grazie a Valentina, per essere la mia perfetta metà, per la sua bellezza e profondità, le sue osservazioni, i suoi libri e il suo amore per le cose importanti. Grazie a Giulia: il solo fatto di rendermi partecipe di come vede il mondo con i suoi occhi ha cambiato la mia vita. Grazie a papà e mamma, per avermi tramandato la loro concezione della vita, per avermi insegnato a guardare oltre la punta del mio naso e avermi sempre supportato nelle mie scelte e decisioni. Grazie ad Anna, per l'allegria che mi mette ogni volta che ci incontriamo e per la sua musica. Grazie ai nonni e agli zii, senza il loro supporto non sarei mai potuto arrivare fino a qui. Grazie all'arte di Gigi e alla compagnia di Annetta. Grazie ad Antonio, per essere stato un grandissimo compagno di studi ed un ancor più grande fratello di vita. Grazie a Marco&Marco, la nostra amicizia è uno dei più importanti cardini nella mia vita. E grazie a tutti i nuovi amici bolognesi!

CONTENTS

1 INTRODUCTION 1
1.1 Contributions of the Dissertation 3
1.1.1 Specification of Interaction Models 3
1.1.2 Static Verification of Interaction Models 4
1.1.3 Run-time Verification, Monitoring and Enactment Facilities 4
1.1.4 A-Posteriori Verification of Execution Traces 5
1.2 Organization of the Dissertation 5
1.2.1 Part I: Specification 6
1.2.2 Part II: Static Verification 6
1.2.3 Part III: Run-Time and A-Posteriori Verification 7

I SPECIFICATION 9
2 DECLARATIVE OPEN INTERACTION MODELS 11
2.1 Open Declarative Interaction Models: an Intuitive Characterization 12
2.1.1 Activities, Events and Execution Traces 12
2.1.2 Characterization of Time 13
2.1.3 Procedural Vs Declarative Interaction Models 14
2.1.4 Open Vs Closed Interaction Models 14
2.2 Business Process Management 14
2.2.1 Limits of Procedural Business Process Modeling 15
2.2.2 The Need For Flexibility 17
2.3 Service Oriented Computing 18
2.3.1 Service Oriented Architecture 18
2.3.2 Orchestration and Choreography 19
2.3.3 Limits of Procedural Choreography Modeling 21
2.4 Multi-Agent Systems 23
2.5 Clinical Guidelines 25
2.5.1 The Role of Basic Medical Knowledge in Clinical Guidelines 26
2.5.2 Semi-openness of Clinical Guidelines 27
2.6 Lessons Learnt 28
2.7 Challenges in Declarative Open Interaction Models 30
2.8 Grounding the Framework 32
3 THE CONDEC GRAPHICAL LANGUAGE 35
3.1 ConDec in a Nutshell 35
3.2 ConDec models 36
3.3 Constraints 37
3.3.1 Existence Constraints 37
3.3.2 Choice Constraints 38
3.3.3 Relation Constraints 39
3.3.4 Negation Constraints 41
3.3.5 Branching Constraints 43
3.4 A ConDec Choreography 43

3.5 Usability of the Language 46
3.6 Linear Temporal Logic 48
3.6.1 LTL Models 49
3.6.2 Syntax of LTL 49
3.6.3 Semantics of LTL 50
3.7 Translation of ConDec to LTL 51
4 THE CLIMB RULE-BASED LANGUAGE 53
4.1 The CLIMB Language in a Nutshell 54
4.2 The CLIMB Syntax 55
4.2.1 Event Occurrences and Execution Traces 55
4.2.2 Constraint Logic Programming 57
4.2.3 Expectations 58
4.2.4 Integrity Constraints 60
4.2.5 The Static Knowledge Base 64
4.2.6 SCIFF-lite and Composite Events 66
4.3 CLIMB Declarative Semantics 67
4.3.1 Abduction 67
4.3.2 Abductive Logic Programming 69
4.3.3 Representing a system and its executions 70
4.3.4 SCIFF-lite Specifications 72
4.3.5 A Declarative Notion of Compliance 73
4.4 Equivalence and Compositionality 77
4.4.1 Equivalence w.r.t. Compliance 78
4.4.2 Compositionality w.r.t. compliance 79
5 TRANSLATING CONDEC TO CLIMB 83
5.1 Translation of a ConDec Model to CLIMB 84
5.2 Translation of Events 85
5.3 Embedding a Qualitative Characterization of Time in a Quantitative Setting 85
5.3.1 Temporal Contiguity 85
5.3.2 Compact Execution Traces 86
5.4 Translation of Constraints 87
5.4.1 Translation of Existence Constraints 88
5.4.2 Translation of Choice Constraints 89
5.4.3 Translation of Relation Constraints 91
5.4.4 Translation of Negation Constraints 94
5.4.5 Dealing with Branching ConDec Constraints 95
5.4.6 Equivalence Between ConDec Constraints 95
5.5 Soundness of the Translation 96
5.5.1 Trace Mapping 96
5.5.2 Compliance Preservation 97
5.5.3 Proof of Soundness 97
5.6 On the Expressiveness of SCIFF 99
5.6.1 A Separated Normal Form for LTL Formulae 100
5.6.2 Translation of SNF Formulae to SCIFF-lite 101
5.6.3 Translation of Arbitrary LTL Formulae to SCIFF-lite 104
6 EXTENDING CONDEC 109
6.1 Temporal-constrained Relationships 109

6.1.1 Temporal Contiguity in a Quantitative Setting 110
6.1.2 Quantitative Formalization of Chain Constraints 110
6.1.3 Metric ConDec Constraints 111
6.2 Data-Related Aspects 114
6.2.1 The MXML Meta-Model 114
6.2.2 The Life Cycle of ConDec Activities 115
6.2.3 An illustrative example 116
6.3 Introducing Data in ConDec^{++} 116
6.3.1 Representing Non-Atomic Activities in ConDec^{++} 117
6.3.2 Formalizing the Activity Life Cycle 117
6.3.3 Modeling the Submit-Review Example 119
6.3.4 Cross-Flow Constraints 120
7 RELATED WORK AND SUMMARY 123
7.1 Related Work 123
7.1.1 Business Process Management 123
7.2 Clinical Guidelines 125
7.2.1 Service-Oriented and Systems 126
7.2.2 Multi-Agent Systems 127
7.3 Summary of the Part 128

II STATIC VERIFICATION 131
8 STATIC VERIFICATION OF DECLARATIVE OPEN INTERACTION MODELS 133
8.1 Desiderata for Verification Technologies 133
8.2 Verification of a Single Model vs a Composition of Models 134
8.3 Static Verification of Properties 135
8.3.1 Existential vs Universal Properties 136
8.3.2 General vs Particular Properties 136
8.3.3 On the Safety-Liveness Classification 138
8.3.4 A ConDec Example 140
8.4 A-priori Compliance Verification 141
8.5 Compatibility and Legal Compositions 142
8.5.1 Compatibility Between Local Models 143
8.5.2 From Openness to Semi-Openness 145
8.5.3 Augmenting ConDec Models with Roles and Participants 146
8.6 Conformance With a Choreography 149
9 Proof Procedures 153
9.1 The SCIFF Proof Procedure 154
9.1.1 Data Structures and Proof Tree 155
9.1.2 Transitions 157
9.2 Formal Properties of the SCIFF Proof Procedure 163
9.2.1 Soundness 163
9.2.2 Completeness 163
9.2.3 Termination 164
9.2.4 ConDec Models and Termination of the SCIFF Proof Procedure 165
9.3 The g-SCIFF Proof Procedure 165

9.3.1 Generation of Intensional Traces 166
9.3.2 Data Structures Revisited 166
9.3.3 Transitions Revisited 167
9.3.4 Comparison of the Proof Procedures 168
9.4 Formal Properties of the g-SCIFF Proof Procedure 169
9.4.1 Soundness 169
9.4.2 Completeness W.r.t. Generation of Traces 169
9.4.3 Termination 171
9.4.4 ConDec Models and Termination of the SCIFF Proof Procedure 171
9.5 Implementation 171
10 Static Verification of ConDec Models With g-SCIFF 173
10.1 Existential and Universal Entailment in CLIMB 174
10.1.1 Specification of Properties with ConDec 174
10.1.2 Formalizing Existential and Universal Entailment 175
10.2 Verification of Existential Properties With g-SCIFF 176
10.2.1 Conflict-freedom Checking Via g-SCIFF 176
10.2.2 Existential Entailment with g-SCIFF 177
10.3 Verification of Universal Properties With g-SCIFF 178
10.3.1 Complementing Integrity Constraints 178
10.3.2 Reduction of Universal Entailment to Existential Entailment 179
10.4 ConDec Loops and Termination Issues 181
10.4.1 Reformulation of ConDec relation constraints 183
10.4.2 Unbounded Specifications and Looping ConDec Models 185
10.5 Pre-processing of ConDec Models and Loop Detection 188
10.5.1 Transformation of ConDec Models to AND/OR Graphs 188
10.5.2 Detection of $\wedge$- and $\vee$-loops 189
10.5.3 Pre-Processing Procedure 192
10.6 Dealing With an Infinite Number of Finite Derivations 195
10.6.1 Succession Constraints and Infinite Branching Proof Trees 195
10.6.2 Solving the Infinite Branches Anomaly 197
11 EXPERIMENTAL EVALUATION 199
11.1 Verification Procedure with g-SCIFF 200
11.2 Scalability of the g-SCIFF Proof Procedure 201
11.2.1 The Branching Responses Benchmark 202
11.2.2 The Alternate Responses Benchmark 203
11.2.3 The Chain Responses Benchmark 206
11.3 Using Model Checking For the Static Verification of ConDec Models 209
11.3.1 Model Checking 209
11.3.2 Verification of ConDec Properties By Satisfiability and Validity Checking 211
11.3.3 Reduction of Validity and Satisfiability Checking to Model Checking 213
11.3.4 Verification Procedure by Model Checking 214

11.4 Comparative Evaluation 215
11.4.1 Evaluation Benchmarks 215
11.4.2 Experimental Results 216
11.5 Discussion 217
12 RELATED WORK AND SUMMARY 221
12.1 Related Work 221
12.1.1 Verification of Properties 221
12.1.2 A-priori Compliance Verification 225
12.1.3 Model Composition 227
12.1.4 Interoperability and Choreography Conformance 228
12.2 Summary of the Part 229
III RUN-TIME AND A-POSTERIORI VERIFICATION 231
13 RUN-TIME VERIFICATION 233
13.1 The Run-Time Verification Task 234
13.2 Run-time Verification with the SCIFF Proof Procedure 235
13.2.1 Reactive Behaviour of the SCIFF Proof Procedure 235
13.2.2 Open Derivations 236
13.2.3 Semi-Open Reasoning 238
13.3 The SOCS-SI Tool 240
13.4 Speculative Run-Time Verification 241
13.4.1 Speculative Verification with the g-sciff Proof Procedure 242
13.4.2 Interleaving the sciff and g-sciff Proof Procedures 243
14 MONITORING AND ENACTMENT WITH REACTIVE EVENT CALCULUS 245
14.1 Event Calculus 246
14.1.1 The Event Calculus Ontology 246
14.1.2 Domain-Dependent vs Domain-Independent Axioms 247
14.1.3 Reasoning with Event Calculus 248
14.2 The Reactive Event Calculus 249
14.3 REC Illustrated: A Personnel Monitoring Facility 251
14.3.1 Formalizing the Personnel Monitoring Facility in REC 252
14.3.2 Monitoring a Concrete Instance 253
14.3.3 The Irrevocability Issue 254
14.4 Formal properties of REC 255
14.4.1 Irrevocability of REC 255
14.5 Monitoring Optional Constraints with REC 261
14.5.1 Representing ConDec Optional Constraints in REC 261
14.5.2 Identification and Reification of Violations 264
14.5.3 Compensating Violations 266
14.5.4 Monitoring Example 266
14.6 Enactment of ConDec Models 269
14.6.1 Showing Temporarily Unsatisfied Constraints 271
14.6.2 Blocking Unexecutable Activities 271
14.6.3 Termination of the Execution 273

15 DECLARATIVE PROCESS MINING 275
15.1 Grounding the Process Mining Framework: SCIFF Checker, DecMiner, ProM 277
15.2 The SCIFF Checker ProM Plug-in 278
15.2.1 CLIMB Textual Business Rules 279
15.2.2 A Methodology for Building Rules 280
15.2.3 Specification of Conditions 281
15.2.4 Compliance Verification with Logic Programming 282
15.2.5 Embedding SCIFF Checker in ProM 283
15.3 Case Studies 284
15.3.1 The Think3 Case Study 285
15.3.2 Screening Guideline of the Emilia Romagna Region 288
15.3.3 Quality Assessment in Large Wastewater Treatment Plans 289
15.4 The DecMiner ProM Plug-in 291
15.4.1 Inductive Logic Programming For Declarative Process Discovery 292
15.4.2 Embedding DecMiner Into the ProM Framework 293
15.5 The Checking-Discovery Cycle 294
16 RELATED WORK AND SUMMARY 297
16.1 Related Work 297
16.1.1 Run-Time Verification and Monitoring 297
16.1.2 Enactment 299
16.1.3 Log-Based Verification 300
16.1.4 Discovery 301
16.2 Summary of the Part 302

IV CONCLUSIONS AND FUTURE WORK 305
17 CONCLUSIONS AND FUTURE WORK 307
17.1 Conclusions 307
17.2 Future Work 309

BIBLIOGRAPHY 313

LIST OF FIGURES

Figure 1 The BP life cycle. 15
Figure 2 A procedural Business Process (BP) modeled in BPMN. 16
Figure 3 Procedural vs declarative perspective in Business Process modeling. 16
Figure 4 Service Oriented Architecture (SOA). 18
Figure 5 Orchestration and choreography. 20
Figure 6 Declarative vs. procedural style of modeling a simple choreography [146]. 24
Figure 7 Fragment of a chronic cough treatment guideline modeled in GLARE (from [33]). 27
Figure 8 The contrasting forces of compliance and flexibility. 29
Figure 9 Comparison of declarative and open vs procedural and closed interaction models in the space of execution trace. 30
Figure 10 Life cycle of declarative open interaction models. 31
Figure 11 The CLIMB framework. Each number identifies the part of the dissertation covering the corresponding portion of the schema. 33
Figure 12 ConDec diagram capturing the Customer-Seller-Warehouse example; for the sake of readability, the three interacting roles are shown as pools in the diagram. 46
Figure 13 Hidden dependencies in a ConDec model (shown as dashed connections). 48
Figure 14 Variants of the SCIFF language and their expressiveness. 54
Figure 15 Abstraction of a system execution in terms of occurring events. 55
Figure 16 Agent UML model for a simplified version of the *query-ref* FIPA interaction protocol. 71
Figure 17 The two-ways relationship between expectations and happened events, established by the CLIMB declarative semantics. 76
Figure 18 Compliance of the execution traces reported in Examples 4.11 and 4.13 with the query-ref protocol. 78
Figure 19 The set of compositional specifications compared to the variants of SCIFF. 82
Figure 20 A simple ConDec model containing a choice constraint. 90
Figure 21 Metric constraints in ConDec^{++}. 113

Figure 22 An MXML-like meta model for representing the traces produced by the execution of interaction instances (the original MXML diagram can be found in [199]). 114
Figure 23 The life cycle of ConDec activities (from [157]). 115
Figure 24 Atomic and non-atomic activities in ConDec^{++}. 117
Figure 25 Example of a ConDec^{++} model. 119
Figure 26 Conformance and replaceability of services in a choreography. 135
Figure 27 ConDec and verification of properties. 136
Figure 28 ConDec model of an order&payment protocol. 140
Figure 29 A-priori compliance verification. 142
Figure 30 Two complementary methods to compose local interaction models for realizing a global choreographic model. 143
Figure 31 Local models of a customer and of three candidate sellers. 146
Figure 32 Different possible errors in the realization of a choreography. 149
Figure 33 A simple payment choreography. 151
Figure 34 Three candidate local models (one customer and two sellers) for the payment choreography shown in Figure 33. 151
Figure 35 Representation of Query 8.4 – introduced in Section 8.3.4 – with extended ConDec. 175
Figure 36 Shifting the perspective when modeling the alternate response constraint. 184
Figure 37 Three ConDec models containing different kind of loops. 186
Figure 38 The three AND/OR forward graphs corresponding to the ConDec models shown in Figure 37. 189
Figure 39 Pre-processing analysis of ConDec models for detecting and handling the presence of loops. 193
Figure 40 Part of the infinite branching proof tree produced by `g-sciff` when the model contains a succession constraint. 196
Figure 41 Complete proof tree produced by `g-sciff` when the model contains a succession constraint, and the revised formalizations is adopted. The infinite branches anomaly is not experienced anymore. 198
Figure 42 The *branching responses* benchmark when 7 activities and 8 constraints are employed. 202
Figure 43 Trend of `g-sciff` when reasoning upon the *branching responses* benchmark. 202
Figure 44 The *alternate responses* benchmark, parametrized on N and K. 204
Figure 45 Trend of `g-sciff` when reasoning upon the *alternate responses* benchmark. 204

Figure 46 The *chain responses* benchmark, parametrized on N and K. 206
Figure 47 Trend of g-sciff when reasoning upon the *chain responses* benchmark by adopting a qualitative notion of time. 206
Figure 48 Trend of g-sciff when reasoning upon the *chain responses* benchmark by adopting a quantitative notion of time. 207
Figure 49 Kripke structure of a micro-wave oven (from [57]). 209
Figure 50 A non-deterministic Büchi automaton representing the LTL formula $\square(start \Rightarrow \Diamond heat)$, produced by the LTL2BA algorithm[83]. 210
Figure 51 Parametric extension to the model presented in Figure 59. 216
Figure 52 Charts showing the ratio NuSMV/g-sciff runtime, in Log scale. 217
Figure 53 ConDec and run-time verification. 234
Figure 54 SOCS-SI architecture. 240
Figure 55 An order management ConDec model with hidden dependencies. 242
Figure 56 Run-time verification vs monitoring. 246
Figure 57 Fluents tracking with REC. 254
Figure 58 ConDec run-time verification and monitoring. 261
Figure 59 A ConDec choreography fragment, including a deadline and a compensation. 267
Figure 60 Fluents trend generated by REC when monitoring a specific interaction w.r.t. the diagram of Figure 59. The verification time spent for reacting to each happened event is also reported. 268
Figure 61 Enactment of ConDec models. 269
Figure 62 Enactment of the ConDec model shown in Figure 55, after the execution of activity empty stock. 272
Figure 63 Some process mining techniques (from [197]). 276
Figure 64 Grounding of the process mining general schema on declarative technologies. 277
Figure 65 An excerpt of the CLIMB textual rules grammar. 279
Figure 66 A methodology for building, configuring and applying business rules. 281
Figure 67 Basic hierarchy of string and time constraints. 282
Figure 68 A screenshot of the main SCIFF Checker window. 283
Figure 69 Compliance chart produced by SCIFF Checker at the end of verification. 284
Figure 70 Basic Think3 workflow for the managament of manufacturing products. 286
Figure 71 Architecture of an intelligent monitoring and quality assessment framework for wastewater treatment plants (from [123]). 290
Figure 72 DecMiner plug-in: trace classification. 293

Figure 73 DecMiner plug-in: ConDec template selection. 294
Figure 74 Integration of SCIFF Checker and DecMiner, realizing a checking-discovery cycle. 295

LIST OF TABLES

Table 1 Some similarities between Multi-Agent and Service-Oriented systems. 24
Table 2 Interplay between CG's prescriptions and the Basic Medical Knowledge (from [32]). 28
Table 3 Examples of interaction models. 29
Table 4 ConDec existence constraints. 38
Table 5 ConDec choice constraints. 39
Table 6 ConDec relation constraints. 40
Table 7 ConDec negation constraints. 42
Table 8 Mapping the statements of the Customer-Seller-Warehouse in ConDec. 44
Table 9 Implicit ConDec constraints involved in the Customer-Seller-Warehouse example. 45
Table 10 Some cognitive dimensions. 46
Table 11 Core ConDec graphical elements and their corresponding meaning. 47
Table 12 LTL temporal operators. 50
Table 13 Common syntax of SCIFF integrity constraints. 60
Table 14 CLIMB specialization of the integrity constraints syntax reported in Table 13. 61
Table 15 Common syntax of a SCIFF knowledge base. 64
Table 16 CLIMB specialization of the knowledge base syntax reported in Table 15. 65
Table 17 SCIFF-lite specialization of the syntax reported in Tables 13 and 15 respectively. 66
Table 18 Translation of ConDec existence constraints to CLIMB. 88
Table 19 Translation of ConDec choice constraints to CLIMB. 89
Table 20 Guidelines followed for translating ConDec relation and negation constraints to CLIMB. 92
Table 21 Translation of ConDec relation constraints to CLIMB. 93
Table 22 Translation of ConDec negation constraints to CLIMB. 94
Table 23 Equivalence of ConDec negation constraints[186]. 96
Table 24 Strength of "forward" relation ConDec constraints. 184
Table 25 Timings employed by `g-sciff` to verify the *branching responses* benchmark. 202
Table 26 Timings employed by `g-sciff` to verify the *alternate responses* benchmark. 205

Table 27 Timings employed by `g-sciff` to verify the *chain responses* benchmark. 207
Table 28 Results of the benchmarks (SCIFF/NuSMV), in seconds [144]. 217
Table 29 The EC ontology. 247
Table 30 Representing some optional ConDec constraint in REC. 263
Table 31 REC formalization of the choreography fragment shown in Figure 59. 267
Table 32 Enactment of a ConDec model. 270

ACRONYMS

ALP Abuctive Logic Programming

B2B Business-To-Business

BDD ordered Binary Decision Diagram

BP Business Process

BPM Business Process Management

BPMN Business Process Modeling Notation

CEC Cached Event Calculus

CEP Complex Event Processing

CG Clinical Guideline

CHR Constraint Handling Rules

CLIMB Computational Logic for the verIfication and Modeling of Business processes and choreographies

CLP Constraint Logic Programming

EBS Event-Based System

EC Event Calculus

FOL First Order Logic

KB Knowledge Base

IC Integrity Constraint

LP Logic Programming

LTL propositional Linear Temporal Logic

MAS	Multi-Agent Systems
MC	Model Checking
MTL	Metric Temporal Logic
NAF	Negation As Failure
MTL	Metric Temporal Logic
REC	Reactive Event Calculus
SCIFF	Social Constrained IFF Framework
`sciff`	SCIFF Proof Procedure
`g-sciff`	g-SCIFF Proof Procedure
REC	Reactive Event Calculus
SOA	Service Oriented Architecture
SOC	Service Oriented Computing
SNF	Separated Normal Form
TPTL	Timed Propositional Temporal Logic
WfMS	Workflow Management System
WS	Web Service

1

INTRODUCTION

Contents

1.1 Contributions of the Dissertation 3
1.1.1 Specification of Interaction Models 3
1.1.2 Static Verification of Interaction Models 4
1.1.3 Run-time Verification, Monitoring and Enactment Facilities 4
1.1.4 A-Posteriori Verification of Execution Traces 5
1.2 Organization of the Dissertation 5
1.2.1 Part I: Specification 6
1.2.2 Part II: Static Verification 6
1.2.3 Part III: Run-Time and A-Posteriori Verification 7

The thesis defended in this dissertation is that

> *Declarativeness* and *openess* are needed to deal with different emerging settings, where systems are composed by several autonomous entities which collaborate and *interact* to the aim of achieving complex strategic goals, impossible to be accomplished on their own. Computational Logic is a suitable framework to support the entire life cycle of such systems, ranging from their specification and static verification to their execution, monitoring and analysis.

The advent of distributed and heterogeneous systems has laid the foundation for the birth of new architectural paradigms, which enable to attack the complexity of a targeted domain by splitting it up into several interacting components and entities. In the field of software engineering, the difference of this kind of paradigms w.r.t. to classical centralized and monolithic ones has been identified since 1975, when DeRemer and Kron conied the two opposite terms of *programming-in-the-small* vs *programming-in-the-large* [66].

A non exhaustive list of systems targeted by programming-in-the-large paradigms includes Business Process Management (BPM), Clinical Guidelines and Care-flow Protocols, Service-Oriented and Multi-Agent Systemss (MASs). For example, BPM systems help organizations in the process of decomposing the work into sub-units, whose execution is then delegated to different parties. The behaviour of these parties must be disciplined so as to obtain, from their cooperation, the

achievement of strategic business goals, such as supplying a service to the customer or deliver a product.

In this settings, the focus of the engineering process is on the *interaction* between the entities composing the system. It is largely recognized that engineering interaction requires novel modeling techniques. In particular, many authors are claiming that an open, declarative perspective is needed to complement the closed, procedural nature of the state of the art specification languages [194, 19, 189, 158, 87, 43, 100, 206, 7, 146].

The main drawback of procedural, closed approaches is that they impose to explicitly tackle all the ordering constraints among the activities carried out by the interacting entities, leading to synthesize a single pre-defined way for accomplishing the desired strategic goals. They force the modeler to produce a rigid scheme defining one fixed algorithm for disciplining the work, ruling out many acceptable possibilities. This is unreasonable when the system must be able to cope with an unpredictable, changing and dynamic environment, where the expertise of workers must be exploited at best [157] or the autonomy of interacting entities and the possibility of exploiting new opportunities must be preserved as much as possible [206].

In this respect, open and declarative approaches are gaining consensus as a way to capture requirements, regulations, policies and best practices involved in the domain under study, but by guaranteeing at the same time *flexibility* and, in turn, usability. In order to enable the effective adoption of such novel modeling approaches, two fundamental requirements must be satisfied:

- on the one hand, they must provide an appealing, intuitive graphical front-end, for the sake of usability by non-IT savvy;
- on the other hand, they must be prone to verification, in order to guarantee the trustworthiness and reliability of the developed model, as well as to ensure that the actual executions of the system effectively comply with it.

In this dissertation, we describe an integrated framework, called CLIMB, able to meet both these fundamental requirements. The CLIMB framework relies on the ConDec graphical language for the high-level, open and declarative specification of interaction models. ConDec has been recently proposed by Pesic and van der Aalst [157, 158] for tackling the flexible, declarative and open modeling of Business Process models. Instead of rigidly defining the flow of interaction, ConDec focuses on the (minimal) set of external regulations, internal policies, best practices which must be satisfied in order to correctly carry out the collaboration. Differently from procedural specifications, the ConDec language provides a number of control-independent abstractions to constrain activities, alongside the more traditional ones.

To enable reasoning and verification capabilities, we propose a complete and automatic translation of all the ConDec constructs to the CLIMB formal language. The CLIMB language is a sub-set of the SCIFF

Computational Logic-based language, which has been originally developed for specifying the rules of engagement regulating interaction protocols in open MAS [7]. The language features a clear declarative semantics which formally captures the notion of *compliance* of the system's executions with the prescribed rules. The SCIFF framework provides the `sciff` and `g-sciff` proof procedures to reason about the developed models during the design phase as well as at run-time and after the execution. We show how the two proof procedures can be combined to provide complete support to the targeted systems, spanning from static verification to execution support, run-time verification&monitoring and a-posteriori verification. We demonstrate the feasibility and effectiveness of the approach by means of experimental evaluations, and by reporting the application of some of the reasoning techniques to real industrial case studies.

1.1 CONTRIBUTIONS OF THE DISSERTATION

The contributions of this dissertation to advancing research in the specification and verification of interaction models can be grouped into four categories:

- specification of interaction models;
- static verification of interaction models;
- run-time verification, monitoring and enactment facilities;
- a-posteriori verification of execution traces.

1.1.1 *Specification of Interaction Models*

We provide an overview of different emerging settings in which openness and declarativeness are of fundamental importance to deal with their complexity at the right level of abstraction, and guaranteeing flexibility. In particular, we focus our attention on Business Process Management systems, Clinical Guidelines, Service Composition/Choreography, and Multi-Agent Systems, pointing out the limits of state of the art modeling languages. We then propose the combination of the ConDec graphical notation and the CLIMB Computational-Logic based language to obtain a suitable modeling framework. We then introduce a complete automatic translation from the ConDec constructs to CLIMB rules. The translation has a twofold advantage: on the one hand, non IT-savvy have the possibility of developing CLIMB specifications by working at the graphical level of ConDec, avoiding the direct manipulation of CLIMB formulae; on the other hand, ConDec is equipped with an underlying formal representation, which can be obtained in an automatic, transparent way. In particular, the translation enables:

- to exploit the expressiveness of the CLIMB language for extending the ConDec notation with new interesting capabilities, such as data-related aspects, a non-atomic model for activities, and metric temporal constraints;

- the application of the `sciff` and `g-sciff` proof procedures for reasoning about (extended) ConDec models.

Finally, we provide a comparison between the CLIMB formalization and the original formalization of ConDec, which has been given in terms of propositional Linear Temporal Logic (LTL). We demonstrate that our formalization is sound w.r.t. the LTL one, and investigate a more general theoretical comparison between LTL and SCIFF, proving that SCIFF is strictly more expressive than LTL.

1.1.2 *Static Verification of Interaction Models*

Although declarative technologies improve readability and modifiability, and help reducing programming errors, what makes systems trustworthy and reliable is formal verification. Static verification aims at verifying the model during the design phase, *before* the execution. It provides support for guaranteeing a-priori that the model will behave, during the execution, in a consistent manner, enabling the premature identification of errors and undesired situations which, if encountered at run-time, could be costly to repair or could even compromise the entire system. When the model under study is part of a complex system, static verification can be employed to check whether it fits with the other components of the system, dealing with interoperability and compatibility/composability issues.

We deal with all these issues, introducing different static verification tasks that can be applied on ConDec models and their composition. We then show how `g-sciff` is able to accomplish such verification tasks. Since CLIMB belongs to a first-order setting, it suffers of semi-decidability issues, and therefore `g-sciff` is not guaranteed to terminate in all the possible cases. To overcome termination issues, we propose a pre-processing technique which is able to analyze a ConDec model and identify the possible sources of non-termination, allowing to take suitable counter-measures. We provide an extensive quantitative evaluation aimed at assessing the performance and scalability of our verification framework, presenting a comparison with state of the art explicit and symbolic model checking techniques.

1.1.3 *Run-time Verification, Monitoring and Enactment Facilities*

After the design phase, an interaction model is instantiated and executed inside a system. Each instance involves a set of concrete interacting entities, whose behaviour must comply with the prescriptions of the model.

In this respect, two different scenarios arise, depending on whether the interacting entities belong to the same system in which the model has been developed or not. If so, they must be supported during the execution, informing them about how their actions impact on the prescriptions of the model, in terms of the currently enabled and forbidden activities. This task is called *enactment*. If, instead, interacting

entities work within a third party system, or act in a completely autonomous, uncontrollable and possibly untrusted manner, then they must be monitored during the execution, in order to assess whether they are effectively behaving as expected and to report potential violations as soon as possible. *Run-time verification* and *monitoring* deal with this issue.

In the dissertation, we describe how a combination of `sciff` and `g-sciff` can tackle the run-time verification task, guaranteeing that possible violations are identified as soon as possible. We then accommodate a reactive version of the Event Calculus (EC) [114] in SCIFF, showing how it can be exploited for monitoring ConDec constraints and reifying the detected violations, enabling the possibility of handling corresponding compensation mechanisms. We finally present how all these reasoning techniques can be combined to support the enactment of ConDec models, preventing the execution of activities which would surely lead to eventually encounter a violation.

1.1.4 *A-Posteriori Verification of Execution Traces*

In many different settings, when an instance of the system completes its execution, the trace composed by all its occurred events is stored inside an information system, enabling their retrieval and *a-posteriori* analysis. The set of all the stored execution traces provides a detailed insight about the *real* behaviour of the system. It is therefore of key importance to provide suitable technologies for analyzing such execution traces, in order to compare the real system with the model, and to assess if business goals have been effectively achieved or not. In the Business Process Management field, this task is commonly called *process mining*; however, its application is not restricted to Business Process models, but it can seamlessly target any Event-Based System equipped with logging facilities.

We present two tools for *declarative process mining*. The tools have been concretely implemented as part of the ProM framework [190], one of the most popular process mining softwares. The first tool, called SCIFF Checker, verifies whether a set of execution traces complies with a business rule, specified using a textual pseugo-natural notation which resembles the CLIMB language. Beside describing the main features offered by the tool, we also report its application on three different industrial case studies: a company working in the Product Life cycle Management market, the Care-flow Protocol realizing a Clinical Guideline in the Emilia Romagna region of Italy, and a system for the intelligent monitoring of Wastewater Treatment Plants.

1.2 ORGANIZATION OF THE DISSERTATION

The dissertation is organized in four parts:

1. Specification;
2. Static Verification;

3. Run-Time and A-Posteriori Verification;
4. Conclusions and Future Work.

Each one of the three central parts is organized as follows.

1.2.1 *Part I: Specification*

- Chapter 2 introduces declarative open interaction models, providing an overview of different settings (Business Process Management, Service Oriented Computing, Clinical Guidelines and Multi-Agent Systems) in which declarativeness and openness are needed to overcome the limits of the state of the art approaches.
- Chapter 3 presents the ConDec graphical language, originally developed by Pesic and van der Aalst [158, 157] for the flexibile specification of Business Processes with a declarative and open flavor.
- Chapter 4 describes the CLIMB rule-based language, a subset of the SCIFF Computational Logic-based language originally thought for addressing the formal specification of interaction in open Multi-Agent Systems; the chapter overviews syntax and declarative semantics of the language, and prove some interesting formal properties related to the CLIMB fragment.
- Chapter 5 investigates how ConDec models can be automatically translated to CLIMB specifications, obtaining a unified framework for the specification of declarative open interaction models, which covers both graphical and formal aspects. The proposed translation is compared with the original formalization of ConDec, based on propositional Linear Temporal Logic (LTL), to establish if the two formalizations are equivalent and, more in general, to carry out a theoretical investigation about the expressiveness of the two formalisms.
- Chapter 6 shows how the expressiveness of the CLIMB language can be exploited to extend the ConDec notation with novel interesting capabilities, maintaining a complete and valid CLIMB representation.
- Related work and a summary conclude this part.

1.2.2 *Part II: Static Verification*

- In Chapter 8 we discuss the different kind of verifications that can be applied to a ConDec model at design-time, taking into account the case of a single model as well as the one in which many ConDec local models are composed to obtain a more complex global model.

- In Chapter 9 two proof procedures able to reason upon CLIMB specification are recalled, presenting their implementation as well as their formal properties.
- One of these two proof procedure, namely the `g-sciff` proof procedure, is then adopted in Chapter 10, showing how it can deal with the static verification of ConDec models.
- Chapter 11 aims at evaluating the effectiveness of the approach presented in Chapter 10, by conducting a quantitative evaluation of `g-sciff` on different ConDec benchmarks. The chapter shows how the static verification of ConDec models can be also formalized as a model checking problem, and compares performance and timings of `g-sciff` with that of state of the art model checkers.
- Related work and a summary conclude this part.

1.2.3 *Part III: Run-Time and A-Posteriori Verification*

- Chapter 13 introduces the run-time verification problem, showing how the `sciff` and `g-sciff` proof procedures can be successfully exploited for the run-time verification of interacting entities w.r.t. ConDec models.
- Chapter 14 discusses how a reactive form of Event Calculus can be axiomatized on top of `sciff`, augmenting the CLIMB framework with the possibility of dynamically reasoning upon the effects of actions and the state of affairs. This added features are then exploited for monitoring ConDec optional constraints, reifying violations and dealing with compensation mechanisms. The combination of monitoring and run-time verification techniques is finally employed to enact the ConDec models.
- In Chapter 15, we present how CLIMB can be applied to the a-posteriori verification of event logs, to the aim of realizing *declarative process mining* techniques. We describe two implemented tools developed for the log-based verification and process discovery, reporting our experience related to their application on some real case studies.
- Related work and a summary conclude this part.

Part I

SPECIFICATION

2

DECLARATIVE OPEN INTERACTION MODELS

The limits of my language
mean the limits of my world

— Ludwig Wittgenstein

Contents

2.1 Open Declarative Interaction Models: an Intuitive Characterization **12**
2.1.1 Activities, Events and Execution Traces 12
2.1.2 Characterization of Time 13
2.1.3 Procedural Vs Declarative Interaction Models 14
2.1.4 Open Vs Closed Interaction Models 14
2.2 Business Process Management **14**
2.2.1 Limits of Procedural Business Process Modeling 15
2.2.2 The Need For Flexibility 17
2.3 Service Oriented Computing **18**
2.3.1 Service Oriented Architecture 18
2.3.2 Orchestration and Choreography 19
2.3.3 Limits of Procedural Choreography Modeling 21
2.4 Multi-Agent Systems **23**
2.5 Clinical Guidelines **25**
2.5.1 The Role of Basic Medical Knowledge in Clinical Guidelines 26
2.5.2 Semi-openness of Clinical Guidelines 27
2.6 Lessons Learnt **28**
2.7 Challenges in Declarative Open Interaction Models **30**
2.8 Grounding the Framework **32**

In this Chapter, we provide an informal overview of different systems in which interaction plays a central role. We discuss the limits of procedural and closed approaches when modeling such settings, motivating why we claim that also an *open* and *declarative* perspective is needed to compensate these limits and to capture such systems at the right level of abstraction.

We then give a precise characterization to open declarative interaction models, describing a generic framework for managing their life cycle, spanning from their design to their execution, monitoring and

a-posteriori verification. We conclude the Chapter by introducing how we propose to fill the building blocks of such a generic framework.

2.1 OPEN DECLARATIVE INTERACTION MODELS: AN INTUITIVE CHARACTERIZATION

The complexity of today's systems poses important engineering challenges. Thanks to the increasing pervasiveness of networks, computers and communication modalities, this complexity is often faced by decomposing the system under study, and delegating the management of its sub-parts to a set of autonomous, possibly distributed entities. Such entities *interact* and collaborate to the aim of realizing the overall goal of the system. Models developed to describe these systems are called, in this dissertation, *interaction models*.

2.1.1 *Activities, Events and Execution Traces*

An interaction model focuses on the *dynamics* of a system rather than on its static organization: its purpose is to capture the overall behaviour that must be guaranteed by the interacting entities in order to fruitfully accomplish the strategic goals of the system while respecting, at the same time, its requirements and *constraints*.

For example, the items involved in an auction must be sold by strictly adhering to the regulations which express how offers can be placed by bidders, and how the winner is finally determined.

Instances and execution traces

The dynamics of a concrete interaction (called *instance* of the system), manifests itself by means of *events* which occur in the environment due to relevant *activities* performed by the interacting entities, attesting that the instance is evolving. Hence, an instance is completely characterized by the set of events which occurred during its execution. A set of correlated occurred events will be referred to as *execution trace*[1]. For example, in a sanitary structure the sequence of treatments, exams, drug administrations related to a single patient constitute an instance of the system.

Activities

Activities are the basic building block upon which interaction models are built: they represent unit of work that must be accomplished during the execution. They encompass both actions made by a single participants without involving the other ones (such as for example the revision of a mechanical part) as well as interaction between different participants (e.g., communicating to the customer that a given item is out of stock).

Event-based systems

Since the core first-class object of interaction models is the concept of event, in the remainder of this dissertation we will refer to the targeted systems as Event-Based Systems (EBSs).

1 An execution trace will be called *partial* if it covers only a part of the complete evolution of an instance, *complete* otherwise.

2.1.2 *Characterization of Time*

Since interaction models focus on the dynamics of a system, the time dimension plays a central role in them. From a foundational point of view, time can be characterized along many different axes. We cite here only the most relevant for our purposes; the interested reader may refer to [74] for a comprehensive survey on the topic.

QUALITATIVE VS QUANTITATIVE TIME Qualitative time expresses relative positions of temporal objects (i.e., events), while quantitative time supports the definition of distances between time points[107, 140]. For example, "the white rabbit arrived at work *before* Alice" is a qualitative time constraint, whereas "the rabbit took 20 minutes to get to work" is a quantitative time constraint.

DENSE VS DISCRETE The time structure is dense if we assume that the system evolves in a continuous manner, while it is discrete if the evolution of the system is sampled.

POINT VS INTERVAL-BASED A point-based algebra of time associates event occurrences with single time-points, whereas an interval-based algebra provides intervals and their durations as first-class objects.

BOUNDED VS UNBOUNDED Bounded time states that there exists an initial (final) time point which does not have predecessors (successors); if it is not the case, then time is unbounded. Hybrid approaches combine the two notions, stating that time is bounded w.r.t. the initial time but not w.r.t. the final time or vice-versa.

In the remainder of this dissertation, we will characterize time as

- point-based, because each occurred event belonging to an execution trace of the system is usually associated to a single time value [199];
- bounded, because an execution trace has a fixed starting time (corresponding to the time at which the interacting entities started the instance), and a final time, whose value is not known a-priori, at which the interaction will terminate.

Boundedness on the final time is needed because interaction models are developed to accomplish a strategic goal, and it is desirable that such a goal is reached in finite time or, in the negative case, that the impossibility of reaching the goal is detected in finite time. In this respect, all the execution traces characterizing instances of an interaction model will always be finite.

On the contrary, we do not fix whether time is dense or discrete nor whether it has a qualitative or a quantitative characterization. We leave this choices open discussing their impact on interaction models throughout the dissertation.

2.1.3 *Procedural Vs Declarative Interaction Models*

Generally speaking, given a problem...

- a *procedural* approach aims at synthesizing a specific solution to the problem;
- a *declarative* approach focuses on the problem itself, eliciting and capturing its constraint and requirements, and delegating the search of a solution to a general-purpose algorithm.

Differently from procedural approaches, declarative paradigms aim at the separation of logic aspects from control aspects, as long advocated by Kowalski [111].

In the context of this dissertation, a procedural interaction model provides a complete and explicit description of all the steps that must be followed by the interacting entities, while a declarative interaction model adopts a more general and high-level view of interaction specification, focusing on what is the (minimal) set of *constraints* to be fulfilled in order to successfully interact.

2.1.4 *Open Vs Closed Interaction Models*

Once an interaction model has been specified, there exist two possibility concerning how it relates with all the information that has not been explicitly captured in the model itself:

- a *closed* interaction model makes the implicit assumption that all that has not been explicitly captured in the model is forbidden;
- an *open* interaction model makes the opposite assumption, stating that interacting entities can freely behave where not explicitly constrained.

In the next part of this Chapter, we provide an overview of different systems which can be specified by means of interaction models, arguing that declarativeness and openness are fundamental requirements to deal with them.

2.2 BUSINESS PROCESS MANAGEMENT

Companies can reach their business goals and effectively produce a specific service or product that is of value the the customer only if all the involved participants and activities are well organized and regulated. To achieve this goal, a collection of related activities aimed at producing a service or product is organized in a Business Process (BP).

As clearly defined in [202]:

> A BP consists of a set of activities that are performed in coordination in an organizational and technical environments. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with BPs performed by other organizations.

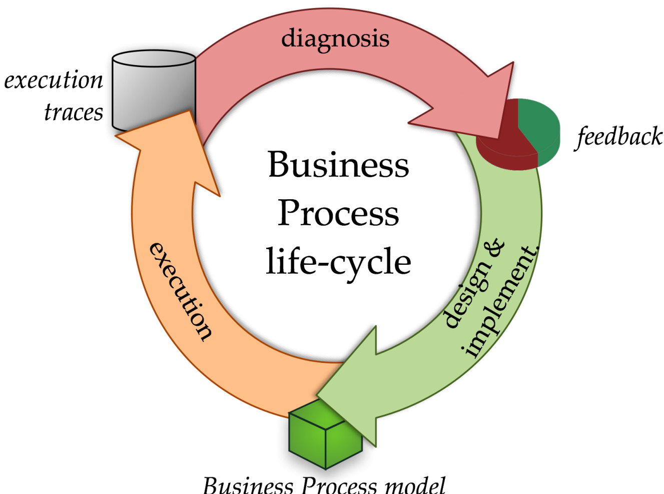


Figure 1: The BP life cycle.

Business Process Management (BPM) is a collection of methods and techniques to assist business practitioners and employees in the management of BPs along their entire life cycle [71], depicted in Figure 1.

The BPM life cycle

The first phase of the BPM life cycle is the *design*, in which the BP models of the company are analyzed and modeled. To develop BP models, process modeling languages are employed. Such languages typically provide a graphical notation, for the sake of readability and user-friendliness. The obtained model is then *implemented* in a BPM system, an information system able to provide support for the execution of process models.

Thanks to this support, different instances of the BP model are then enacted, guiding the work of all the participants involved in their *execution*. Each performed activity is tracked by the BPM system, in order to store all the information characterizing the evolution of each instance (i.e., all execution traces).

In the last *diagnosis* phase, the information recorded in the collected execution traces is analyzed to evaluate whether business goals have been effectively achieved, taking new strategic decisions accordingly. The outcome of this last phase is then used as a feedback for the original BP model, in order to improve it and better align it with business goals and customer's needs.

2.2.1 *Limits of Procedural Business Process Modeling*

By focusing on the control-flow perspective of a BP, that aims at capturing the way in which activities must be performed over time, we will find that all the state of the art BP modeling/implementation languages (such as BPMN [203], BPEL [12] and YAWL [187]), adopt a procedural style. In other words, they impose to explicitly tackle all the ordering constraints among activities, synthesizing one pre-defined way for ac-

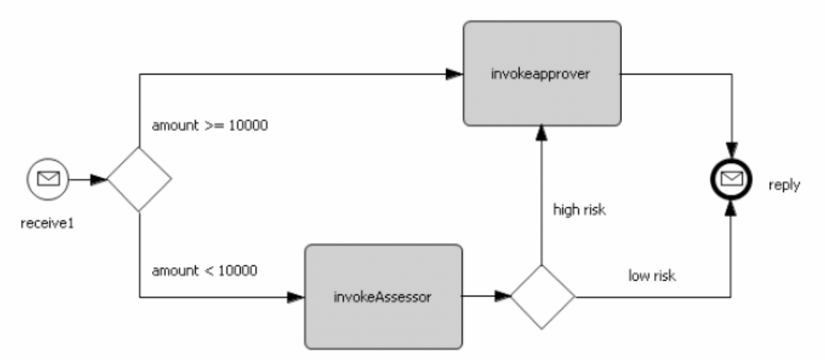


Figure 2: A procedural Business Process (BP) modeled in BPMN.

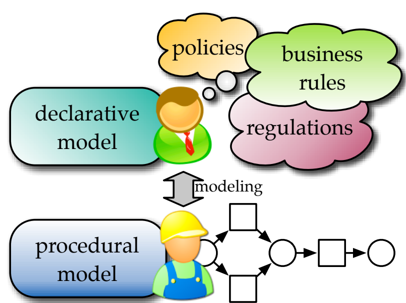


Figure 3: Procedural vs declarative perspective in Business Process modeling.

complishing the desired business goals. In this respect, BPs can be considered as a rigid scheme defining one fixed algorithm for disciplining the work, ruling out other acceptable possibilities [158]. Figure 2 shows an example of this kind of process, using the BPMN language.

While this assumption is reasonable in BPs showing a high degree of predictability and repetitiveness (such as classical production workflows [120]), it is unreasonable when the company must deal with unpredictable and frequently changing processes, where the expertise of workers must be exploited at best. A procedural style of modeling clashes with the unstructured declarative knowledge of business practitioners, who perceive a BP as a "container" of many different aspects, including business goals, business rules, external regulations, internal policies and best practices.

In [157], Pesic points out that the procedural nature of contemporary BPM approaches tends to determine the way companies organize work, forcing them to adjust their business processes to the system: instead of providing support for modeling and implementing the BPs at the level of abstraction of business practitioners and workers, they force to adapt BPs in a way that they fit the system (see Figure 3). Pesic argues that this drawback has a twofold impact:

- due to the gap between the preferred way of work and the system's way of work, the risk of implementing inappropriate BPs, supporting undesired behaviors, increases;
- people perceive the system as an obstacle, performing their concrete work outside the system and recording what has been done in a separate moment.

In [186], van der Aalst and Pesic support this claim by considering a simple yet significative example: expressing a "not coexistence" constraint in the process. Such a constraint could be adopted by a business practitioner to state that two activities are incompatible, i.e., they cannot be executed both in the same instance. Trying to capture such a constraint in a procedural setting forces the modeler to explicitly enumerate all the possible executions which respect it, introducing ambiguous decision points and obtaining a complex, unreadable model. A more detailed discussion of these issues is provided in Section 2.3.3; in fact, the limits of procedural approaches in a BPM setting strictly resemble the case of service choreographies.

2.2.2 *The Need For Flexibility*

The inadequacy of current state-of-the-art BP modeling languages have pointed out by many authors [158, 201, 165], who argue that to be successfully adopted, BPM technologies must make a trade-off between controlling the way workers do their business and turning them loose to exploit their expertise during the execution. In other words, they claim that BPM systems should be *flexible*, w.r.t. the work of practitioners as well as the ability of adapting to environmental changes and to the acquisition of new knowledge.

In [176], Schonenberg et al. introduce four types of flexibility in BPM:

Taxonomy of flexibility

FLEXIBILITY BY DESIGN The ability to model BPs by leaving many possible alternatives to the users, letting them free to choose for the best way of execution.

FLEXIBILITY BY UNDERSPECIFICATION The ability of leaving BPs partially unspecified until the execution.

FLEXIBILITY BY CHANGE The ability of re-organizing the BP model at run-time, selecting which running instances must be migrated to the revised model.

FLEXIBILITY BY DEVIATION The ability of deviating from the prescriptions contained in the BP model, during a specific execution.

In this dissertation, we will mainly focus on the first type of flexibility, by adopting, formalizing and extending a modeling language proposed by Pesic and van der Aalst, called ConDec, able to supports flexibility by design thanks to its declarative style of modeling.

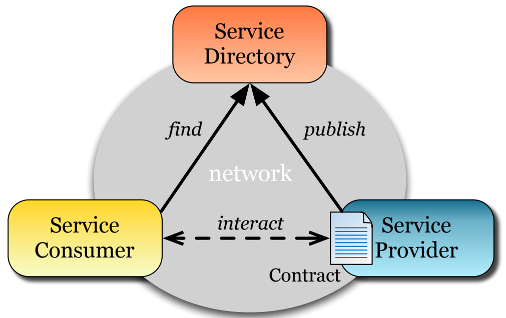


Figure 4: Service Oriented Architecture (SOA).

2.3 SERVICE ORIENTED COMPUTING

Service Oriented Computing (SOC) is an emerging computing paradigm which employs *services* as the basic constructs to support the development of large-scaled distributed applications in heterogeneous environments. Its underlying enabling technologies and languages are reaching a good level of maturity and a widespread adoption, drawing the interest of both software vendors and scientists [194]. As stated in [153]:

> The visionary promise of Service-Oriented Computing is a world of cooperating services where application components are assembled with little effort into a network of services that can be loosely coupled to create flexible dynamic business processes and agile applications that may span organizations and computing platforms.

Roughly speaking, services are self-contained software components which incapsulate computing functionalities, exposing their core business competencies over a network in the form of an usage contract/interface. Third-parties may utilize an exposed functionality through the use of standardized languages and protocols, without accessing to its concrete implementation.

2.3.1 *Service Oriented Architecture*

The basic Service Oriented Architecture (SOA) is depicted in Figure 4. The architecture relies upon three roles and three fundamental operations:

SERVICE REGISTRY maintains a list of known service contracts together with their location, and provides support for searching for a certain functionality, returning the location of a service able to accomplish it.

SERVICE PROVIDER encapsulates a business function (or a set of business functions), exposing a corresponding usage contract; It can interact with the service registry in order to *publish* its contract.

SERVICE CONSUMER is the party which requires a certain functionality. If it does not know where a service able to accomplish the requested functionality is located, it has the possibility to consult the service registry in order to *find* a suitable service. After having retrieved the contract and location of a suitable service, it can finally *interact* with it by invoking the desired functionality, and obtaining a result (if any).

In order to enable the possibility to effectively making heterogeneous service providers and consumers interact, the three fundamental operations as well as the published service contracts require standardized languages and protocols. Each concrete SOA implementation must therefore define its own standards and protocols, tailored to the underlying network in which the implementation is deployed.

Web services

Thanks to the widespread diffusion of the Internet, the web implementation of SOA is nowadays the most popular. In the web implementation, the functionalities provided by business applications are encapsulated within *web services*, which can be retrieved and invoked by service consumers through a stack of Internet standards including HTTP, XML, SOAP [34], WSDL [54] and UDDI [22].

2.3.2 *Orchestration and Choreography*

The role of service consumer could be played by a human user but also by another service provider; in this way, the functionalities realized by single services can be composed, giving birth to complex collaborative distributed business applications. Different organizations can mutually benefit from each other by exposing their own functionalities and exploiting external services.

Two different complementary approaches can be followed to realize/perceive a service composition[156]:

ORCHESTRATION perceives the collaboration by the point of view of a single party. The party synthesizes an executable BP which can interact with internal and external services, delegating the realization of some activities to such services. The execution is then controlled by the single party as well: it acts as an *orchestrator*, coordinating the other services and correlating the results obtained from them. The invoked external services are unaware that they are participating in a collaboration. The orchestration itself could be exposed as a single functionality on the network, making it possible of realizing even more complex "nested" orchestrations.

CHOREOGRAPHY models the collaboration from a global, objective point of view, independently from the perception of the single interacting services. Differently from orchestration, its focus is not on executability, but rather on capturing the public contract which provides the necessary rules of engagement for making all the interacting parties correctly collaborate.

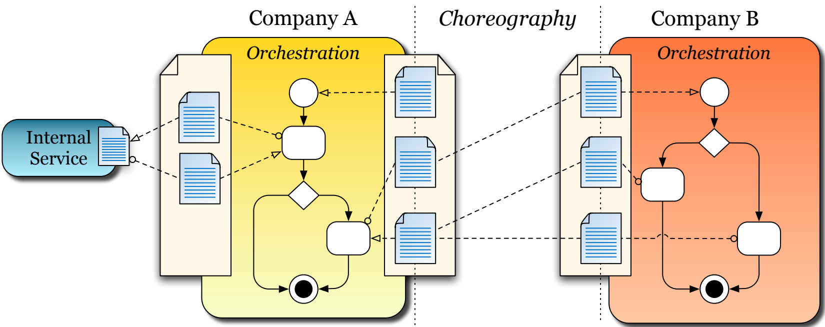


Figure 5: Orchestration and choreography.

The choreography vs orchestration perspective of a service composition are shown in Figure 5. While orchestration is useful when a single party is interested in aggregating the capabilities of a set of internal/external services, choreography helps when the collaboration must be achieved by taking into account the mutual requirements of interacting parties, without assuming a unique center of control.

Choreographies in a B2B setting

Let us for example consider a Business-To-Business (B2B) setting, in which different organizations share their own services to mutually benefit from each other, trying to accomplish a complex strategic goal impossible to be pursued autonomously. In this context, it is often impossible to make the assumption that one of the involved organizations will take the lead during the interaction, acting as an orchestrator. As clearly pointed out in the WS-CDL specification [108]:

> In real-world scenarios, corporate entities are often unwilling to delegate control of their business processes to their integration partners. Choreography offers a means by which the rules of participation within a collaboration can be clearly defined and agreed to, jointly. Each entity may then implement its portion of the choreography as determined by the common or global view.

In a B2B setting, the birth of a service choreography is often determined by putting together external norms/regulations and internal policies, requirements, best practises, business goals of each participating organization. All these different contributions have the effect of constraining the possible allowed interactions.

Compliance vs flexibility in choreographies

The obtained global model should suitably mediate between *compliance* and *flexibility*: on the one hand, all interacting services must respect the agreed constraints; on the other hand, each party should be able to execute the business processes which cover its part of the choreography as free as possible, preserving interoperability and replaceability of services. In other words, we claim that a service choreography should play the role of a public global contract containing the rules that must be respected to correctly interact, without stating how such a collaboration is concretely carried out. This kind of knowledge is therefore inherently *open* and *declarative*.

As pointed out in [19, 189], while the technologies for implementing and interconnecting basic services are reaching a good level of maturity, modeling service interaction from a global viewpoint, i.e., representing service choreographies, is still an open challenge: the leading current proposals for modeling service interaction, such as WS-BPEL [12] and WS-CDL [108], fail to tackle a suitable balance between compliance and flexibility. The main problem is that, despite the complementarity of choreography and orchestration, current mainstream approaches propose very similar languages and methodologies to model them both. All the state-of-the-art approaches focus on procedural aspects, aimed at completely specifying control and message flow of the interacting services. This leads to loose the declarative nature of the knowledge involved in the choreography definition, forcing the modeler to capture it at a procedural level.

Behavioral interfaces

Finally, it is worth noting that when a service is internally modeled as an orchestration of other services, it could publish an extended usage contract, which partially discloses its internal organization. Such an extended contract is usually called *behavioral interface*. Behavioral interfaces may be used to enhance a service registry with more advanced capabilities, such as extracting services which do not only expose the requested functionalities, but also satisfy policies and regulations required by the service consumer[109, 6]. For example, a customer could express that she is looking for an electronic bookshop which supports credit card payment, and it is also able to provide a guarantee that it relies on a secure payment method. In this respect, publishing a procedural behavioral interface would be too restrictive, ruling out the selection of a service even if a fruitful collaboration could be established.

2.3.3 *Limits of Procedural Choreography Modeling*

To illustrate the difficulty of handling even simple choreography constraints with classical procedural approaches, let us consider a fragment of a purchase choreography, regulating the decision of the seller for what concerns an order confirmation or rejection. The seller could freely decide whether to confirm or refuse customer's order, but must obey to the following constraints:

- if the warehouse cannot ship the order, then the seller must refuse it;
- the seller can accept the order only if the warehouse has previously accepted its shipment;
- both the seller and the warehouse cannot accept and reject the same order, i.e., answers are mutually exclusive.

By considering these global rules, many different compliant interactions can be established by a concrete seller and a concrete warehouse. For example, when and how the warehouse is contacted is not specified, and there could be different choreography executions in which

the warehouse is not contacted at all: an execution in which the seller autonomously decides to reject the order, without asking warehouse's opinion, is foreseen by the choreography. This execution trace clearly attests that many different compliant ways to interact are not explicitly mentioned in the choreography, but are instead implicitly supported. We argue that this is due to the fact that choreography rules constitute a form of declarative and open knowledge, which states what is forbidden and mandatory in services without giving details about how to carry out the interaction, nor by making the unreasonable assumption that all the allowed behaviors can be explicitly captured.

Limits of procedural approaches

When the user tries to model this kind of knowledge using a classical procedural specification language such as BPEL or WS-CDL, she is forced to explicitly enumerate all the implicitly supported executions, and to introduce further unnecessary details. Consider for example the adoption of BPMN [203] collaborative diagrams as a modeling language to capture the above described choreography fragment. All candidate diagrams will surely contain two separated pools, one for the seller and one for the warehouse, each containing the set of activitites pertaining to the corresponding role. The problems arise when the modeler tries to interconnet these activities by means of control and message flows: which activities must be executed in sequence? How to deal with negative information such as "the seller cannot accept and reject the same order"? How to deal with non-ordered constraints, such as the one stating that "if the warehouse refuses the order, then the seller must also refuse (or have refused) it"? Who is in charge to contact the warehouse? And when?

The difficuly of providing an answer to these question by adopting a procedural style of modeling is threefold:

LACK OF PROPER ABSTRACTIONS Activities can be inter-connected only by means of positive temporally-ordered relationships (sequence patterns, mixed with constructs aimed at splitting/merging the control or the message flow). Modeling other kind of constraints forces the user to complicate the model. For example, capturing temporally-unordered relationships leads either to choose one ordering and impose it in the model, compromising flexibility, or to explicitly capture all the possible orderings, introducing ambiguous decision points to combine them [186].

CLOSED NATURE Procedural models makes the implicit assumption that "all that is not explicitly modeled is forbidden", and must therefore enumerate all the allowed executions. Therefore, when a negative requirement (such as forbidding a certain activity or stating that two activities must never co-exist in the same execution) must be considered, it cannot be made explicit in the model; instead, it is responsability of the user to check whether the produced model implicitly entails the negative requirement or not. This is a difficult task, especially when the complexity of the model increases.

PREMATURE COMMITMENT Since procedural approaches have a close nature and do not provide proper abstractions, they force the modeler to prematurely take decisions and make assumptions about the interaction. For example, even if the considered choreography fragment does not specify how and when the warehouse must be contacted, a choice must indeed be taken during the modeling phase.

The combination of these drawbacks has the effect that choreographies become *over-specified* and *over-constrained*: unnecessary activities and constraints are introduced, and acceptable interactions are dropped out. As a consequence, while compliance is respected, flexibility becomes sacrificed: potential partners are discarded, fruiful interactions are rejected and, at last, the choreography becomes unusable.

Furthermore, when the modeler tries to get back flexibility by relaxing the imposed constraints and reducing premature commitments, the lack of proper abstraction and the closed nature of procedural approaches lead to further stress over-specification: the resulting choreography tends to become a tangled, unintellegible spaghetti-like model, and, at the same time, the risk of supporting undesirable behaviors increases.

Figure 6(a) shows how the choreography fragment described above can be easily specified with a declarative constaint-based language such as ConDec. Figure 6(b)-(c) instead depict two possible BPMN collaborative diagrams which try to model the described constraints by mediating between compliance and flexibility. The result is that unnecessary activities are introduced (such as the contact warehouse activity) and that some acceptable execution traces are not supported. For example, both diagrams do not support the possibility that the warehouse refuses the shipment after the refusal of the seller; even if the refusal of the warehouse seems to be, in this case, insignificant, it could be involved in other constraints of the choreography, and should therefore be supported. Adding this behaviour would require to complicate the model, replicating execution paths and activities, and introducing ambiguous decision points. And, obviously, this issue would be even more difficult to handle when the modeled fragment has to be composed with other constraints to capture the whole choreography.

2.4 MULTI-AGENT SYSTEMS

The issue about what information should be captured or left out by the global view of interaction has been (and is still) matter of discussion also in the Multi-Agent Systems (MAS) research community. Here, the issue is how to specify and to properly constrain agent interactions, intended as uttered speech-acts or exchanged messages. The problem is two-faced: on one side, there is the need for a semantics of the exchanged messages, on the other it is necessary to allow, constrain and forbid messages and sequences of messages. The concept of choreography is here referred with the term *interaction protocol*. It is not surprising, then, that Multi-Agent and Service-Oriented systems share many

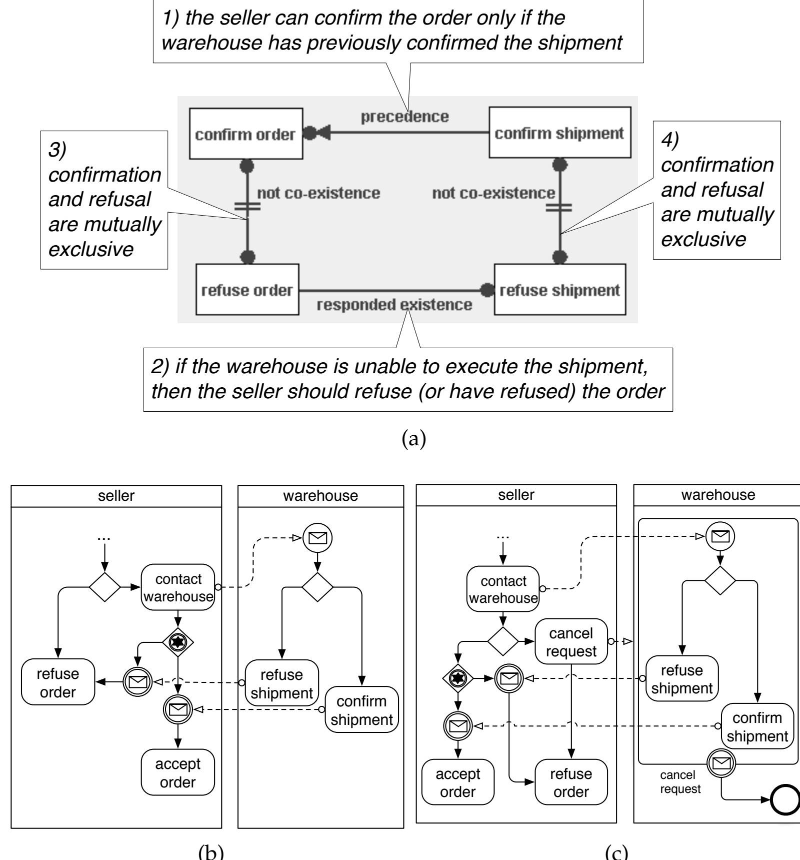


Figure 6: Declarative vs. procedural style of modeling a simple choreography [146].

	MAS	SOC
INTERACTING PARTIES	autonomous heterogeneous agents	autonomous heterougeneous services
COMMUNICATION	speech-acts	messages
LOCAL VIEW OF INTERACTION	(external) agents policies	behavioral interfaces
GLOBAL VIEW OF INTERACTION	global interaction protocols	choreographies

Table 1: Some similarities between Multi-Agent and Service-Oriented systems.

similarities [16, 100], and that similar solutions have been proposed (see Table 1).

As in SOA, also in the MAS community several approaches have been proposed for modeling interaction, spanning from more procedural to more declarative flavor. Approaches like, e.g., AUML [21], aim to exactly specify how the interaction protocols should be executed by the

interacting agents. Other proposals, instead, consider MAS as open societies and model interaction protocols by means of declarative constrains. Social approaches abstract away from the nature of interacting entities, supporting heterogeneity, and adopt an open perspective. Furthermore, their aim is not only to support the specification task, but also to define a precise semantics of interaction, enabling different verification capabilities.

Multi-Agent Systems as open societies

In [206], the authors propose to adopt the notion of *commitment* to provide a semantics to the interaction protocols: an agent (the debtor) makes a commitment to another agent (the creditor) to bring about a certain property. Commitments capture and handle mutual obligations which relate interacting agents, giving a meaning to the exchanged messages in terms of their impact on commitments. A variant of Event Calculus (EC) is used to specify how events (e.g., an exchanged message) affect the evolution (creation, discharge, release, etc.) of the commitments. Noticeably, Singh et al. have recently applied commitment-based protocols also in the context of BPM and SOC, by addressing the problem of BP adaptability [67] and of protocols composition [131].

Another social-based approach has been developed within the SOCS EU Project[2], where global interactions protocols are specified by means of SCIFF.Protocols are specified only by considering the external observable behavior of interacting entities, and by the concept of *expectation* about desired events and interactions; occurred events and positive/negative expectations are linked by means of forward rules. The framework is equipped with a clear declarative semantics and with two proof procedures able to verify the developed models along their entire life cycle.

We strongly believe that the BPM/SOC and MAS research areas can mutually benefit from each other. This dissertation goes in this direction: we propose to formalize the ConDec language by means of SCIFF specifications, aiming at realizing a unifying framework for the specification and verification of declarative open interaction models.

2.5 CLINICAL GUIDELINES

In this Section, we discuss the need of integrating procedural and declarative knowledge to capture Clinical Guidelines (henceforth CGs); the provided motivation and examples are taken from [32].

In the definition of the USA Institute of Medicine:

> Clinical Guidelines are, systematically developed statements to assist practitioner and patient decisions about appropriate health care in specific clinical circumstances.

They enforce evidence-based medicine, and they can be used to promote high-quality medicine and cost optimization, as well as to evaluate the quality of health service and its organization over a territory.

2 SOcieties of heterogeneous ComputeeS, IST-2001-32530 (home page `http://lia.deis.unibo.it/research/SOCS/`).

One of the main goals of CGs is to put evidence into practice. However, from one side, evidence is essentially a form of statistical knowledge, capturing the generalities of classes of patients, rather than the peculiarities of a specific patient. From the other side, demanding to expert committees to characterize all possible executions of a CG on any possible specific patient in any possible clinical condition is an unfeasible task.

Implicit assumptions in the definition of CGs

Thus, several conditions are usually implicitly assumed when building a CG. In particular, CGs are developed by assuming ideality of context and patients, i.e., by hypothesizing their application in a healthcare structure able to provide all the requested resources, and on patients experiencing only the targeted disease. The first assumption is needed because guidelines are developed at an abstract level, without focusing on a specific context of execution. The second assumption is needed because the variety of possible patients is potentially infinite. When the CG is applied on a specific patient, healthcare professionals implicitly employ their *Basic Medical Knowledge* (BMK henceforth) for adapting the generic guidelines prescription to her.

2.5.1 *The Role of Basic Medical Knowledge in Clinical Guidelines*

In the last decade, many different approaches and projects have been developed to create domain-independent computer-assisted tools for managing, acquiring, representing and executing clinical guidelines [61, 155, 182, 184].

CGs are usually specified by adopting a procedural style of modeling. Computer-assisted tools provide graphical languages sharing many similarities with the ones used in the BPM field: actions are inter-connected by constructs aimed at defining a fixed schedule for their execution. Figure 7 depicts a fragment of a CG specified using the GLARE language [184].

As far as now, all this tools assume that CGs must be strictly followed, i.e., they make the assumption that all the involved actions are *must do* actions, and that all the possible treatments are explicitly contained in the CG model (closed approach). Only recently, however, some approaches has started to consider that CGs cannot be interpreted (and executed) in isolation, but that their challenging integration with the BMK of healthcare professionals is of fundamental importance. In fact, actions recommended by a CGs could be prohibited by the BMK, as well as a CG could force some actions despite the BMK discouraging them.

Table 2 summarizes some real examples in which the complex interplay between the CG and the BMK gives raise to different "hybrid" situations, showing that the CG cannot be simply interpreted as a strict, normative procedure. In particular, the examples points out three different integration modalities between the two kind of knowledge:

A. The CG supports the possibility of choosing among two different treatments, and the BMK expresses what choice should be taken when certain circumstances hold.

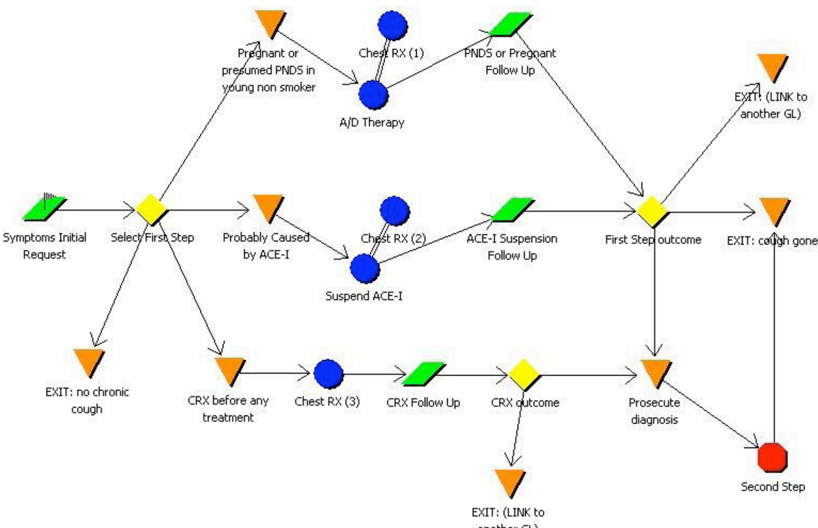


Figure 7: Fragment of a chronic cough treatment guideline modeled in GLARE (from [33]).

B. The BMK emends the CG by stating how to threat a patient when the CG's prescriptions are not applicable.

C. The CG defeats the BMK, stating that even if the treatment could be dangerous for the patient, it has the priority.

2.5.2 *Semi-openness of Clinical Guidelines*

We have seen that CGs are developed making some simplifications, in particular hypothesizing that they are applied on "ideal" patients. The BMK should be used to complement the prescriptions of the CGs, bridging (at least in part) the gap between the ideal and the actual application cases.

While CGs have a procedural and closed nature, the BMK is:

DECLARATIVE, because its aim is not to define rigid procedures that must be followed by health-care professionals, but is rather focused on capturing their best practices and expertise, as well as the underlying common medical knowledge they exploit.

OPEN, because it is unreasonable to assume that it captures the whole knowledge of health-care professionals, and consequently it should not preclude unforeseen actions.

The integration of a CG and the declarative BMK gives rise to a hybrid "semi-open" knowledge, making it possible to apply the CG on each patient by taking into account her specific characteristics.

A deep understanding concerning the nature of this hybrid knowledge, and how its building components actually interact, is still far to

	CG	BMK	CG+BMK
A.	Patients suffering from bacterial pneumonia must be treated with penicillin or macrolid.	Do not administer drugs to which a patient is allergic.	Admister macrolid to a patient with bacterial pneumonia if she is allergic to penicillin.
B.	Patients with post-hemorragic shock require blood transfusion.	Do not apply therapies that are not accepted by patients. Plasma expander is a valid alternative to blood transfusion, under the hypothesis that . . . *(omitted)*	If the patient refuses blood transfusion, in case of post-hemorragic shock treat her with plasma expander.
C.	In patients affected by unstable angina, coronary angiography is mandatory.	A patient affected by advanced predialytic renal failure should not be subject to coronary angioraphy, because the contrast media administration may cause a further deterioration of the renal functions.	Even in case of a predialytic renal failure, perform coronary angiography if the patient is affected by unstable angina.

Table 2: Interplay between CG's prescriptions and the Basic Medical Knowledge (from [32]).

be reached. The first necessary step towards such an understanding is to find suitable specification languages and effective technologies for capturing the BMK, respecting its open and declarative nature. We hope that this dissertation could contribute to the investigation of such a first step.

2.6 LESSONS LEARNT

The presented systems can be considered as interaction models, differing for what concerns the nature of participants and the relevant events occurring during the executions of the system (see Table 3).

The state of the art proposals share a closed and procedural philosophy for modeling such systems. However, as we have pointed out in our discussion, there is an increasing demand for incorporating also a declarative and open perspective, to better fit with the nature of this systems and the requirements and needs of the involved users.

Compliance and flexibility as contrasting forces

By abstracting away from the specific features of each discussed domain, we can identify two complementary aspects which need to be suitably captured by a modeling framework [87, 157][3]:

COMPLIANCE constraints the behavior of interacting entities so as to guarantee that all the external regulations, internal policies, best practices are met.

3 [157] uses the term *support* instead of *compliance*.

INTERACTING ENTITIES	EVENTS	INTERACTION MODEL
Employees, stakeholders, customers, devices, ...	Starting/Completion of activities	Business Process
Distributed heterogeneous services	Exchanged Messages	Service Composition/ Choreography
Agents, other interacting entities	Speech acts, actions on the environment	Multi-Agent System
Healthcare professionals, patients, administrative staff	actions, treatments,	Clinical Guideline + Basic Medical Knowledge

Table 3: Examples of interaction models.

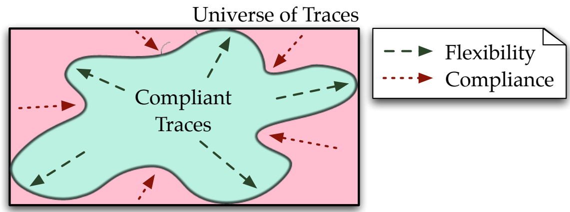
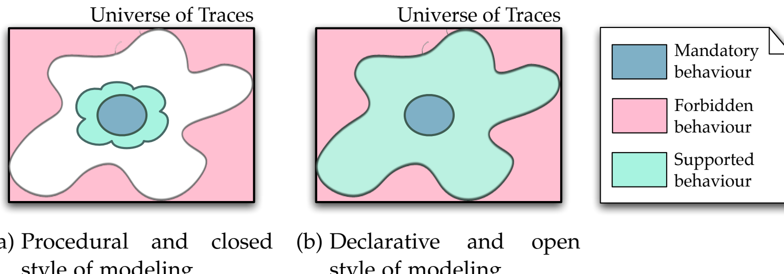


Figure 8: The contrasting forces of compliance and flexibility.

FLEXIBILITY leaves the behavior of interacting entities as unconstrained as possible, allowing them to exploit their own expertise to take strategic decisions and evaluate the best ways to interact.

If we refer these two complementary concepts to the space of possible execution traces that can be generated by the interacting entities, we can see them as two contrasting forces. On the one hand, compliance tends to *restrict* the number of allowed traces, forbidding (mis)behaviors that would lead to violate one of the imposed prescriptions. On the other hand, flexibility tends to *widen* the number of allowed traces, trying to include many different possibilities among which participants can choose. As we have seen, finding the right balance between this two contrasting forces is of key importance: compliance must be preserved for "normative" reasons, while flexibility must be guaranteed to ensure usability and adaptability.

In this respect, procedural and closed approaches provide good support for ensuring that compliance is guaranteed, but they sacrifice flexibility, being prone to over-constrain and over-specify the interaction. Declarative and open approaches, instead, focus on the elicitation of mandatory and forbidden behaviors imposed by compliance, but avoid the insertion of further unnecessary constraints, preserving both compliance and flexibility. Figure 9 gives an intuitive idea of this difference.



(a) Procedural and closed style of modeling.

(b) Declarative and open style of modeling.

Figure 9: Comparison of declarative and open vs procedural and closed interaction models in the space of execution trace.

2.7 CHALLENGES IN DECLARATIVE OPEN INTERACTION MODELS

Having acknowledged the importance of declarative open interaction models, this dissertation aims at providing a framework supporting the entire life cycle of such specifications. Beside design support, to make the developed models and their executions actually trustworthy and reliable, the framework must be able to *formally verify* them along the whole life cycle.

Support on the internal life cycle

Thanks to formal verification, it is possible to guarantee the correctness and consistency of a designed model, as well as effectively establishing whether the behavior of interacting entities really adhere to the prescriptions of the model. In particular, we identify the following requirements and desiderata about a supporting framework (see Figure 10, top part):

DESIGN/IMPLEMENTATION PHASE The framework must provide suitable novel specification languages, modeling the targeted EBSs by following a declarative and open approach; since the designed models are declarative, they can be directly executed, if a general-purpose execution engine actually exists. During the design phase, a model must be verified to guarantee a-priori that it will meet, during the execution, certain desired *properties*, as well as to ensure that it effectively *complies* with external regulations and internal policies.

EXECUTION PHASE The enactment of declarative open interaction models is a challenging task. While procedural closed models are executed by means of a process engine, which shows at each moment what are the next steps to be executed (the so-called *work list*), declarative open models are enacted by presenting, instant by instant, what are the expected and forbidden behaviours.

DIAGNOSIS PHASE In the diagnosis phase, the execution traces of the system are *analyzed* to study if business goals have been effectively achieved and to assess system's trends, or to extract a new model from the concrete traces, in order to better identify the

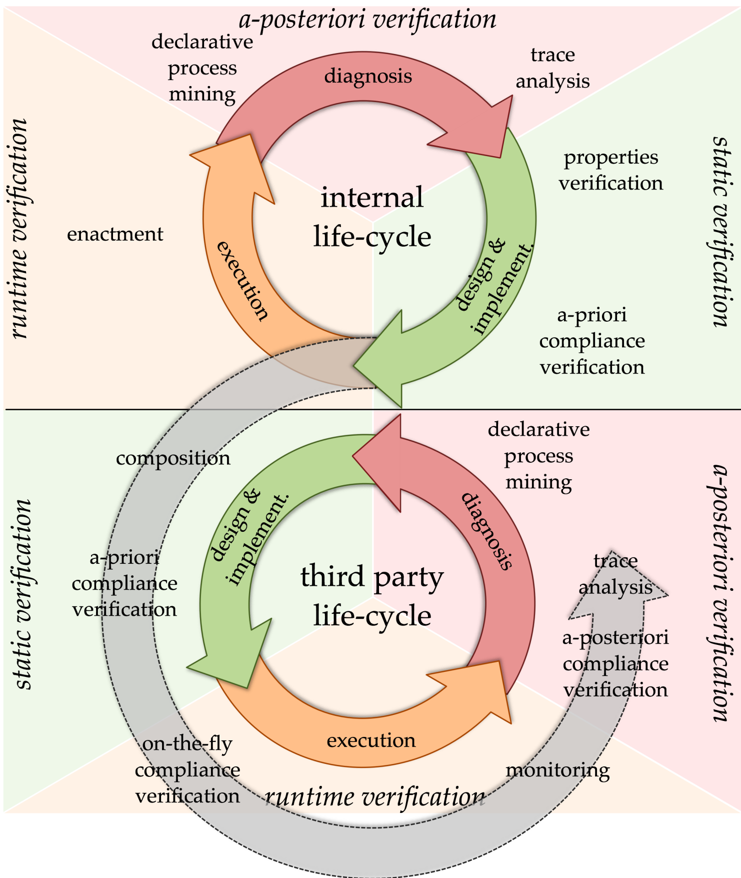


Figure 10: Life cycle of declarative open interaction models.

real behaviour exhibited by the interacting entities. This latter task is commonly known, in the BPM field, as *process discovery* [192].

Support on a third party life cycle

As pointed out in the bottom part of Figure 10, an interaction model could also take part to a third party life cycle. For example, the internal life cycle could be focused on the development of a choreography model, which is then not enacted, but it is rather used to verify if local models of services, possibly provided by external organizations, could be employed to concretely realize the choreography. In this respect, our desired framework should support the following task ($\mathcal{I}$ denotes the internal model which takes part to the life cycle of another specification):

DESIGN/IMPLEMENTATION PHASE $\mathcal{I}$ could represent a choreography or a regulatory/prescriptive model, used in a third party life cycle to assess compliance/suitability of the designed model. Another case is the one in which $\mathcal{I}$ represents a partial/local model, which must be combined with other local/partial models for achieving a complex strategic goal. Our framework should be

able to deal with the *a-priori compliance verification* of third party models, as well as with the composition of models developed by different organizations.

EXECUTION PHASE Suppose we need to carry out the same tasks involved in the design/implementation phase, but the third party model is unknown, unaccessible, or untrusted. This is e.g. the case of a service choreography, in which a set of suitable services is selected to build the underlying composition. Each service is selected by looking at its behavioural interface, and there is no guarantee that the exposed interface corresponds to the internal implementation. Potential mismatches between a behavioural interface and the real implementation of a service may lead to unexpected/undesired interactions. In this case, verification can be accomplished by looking at the actual execution of the external system, checking if the involved interacting entities are behaving as $\mathcal{I}$ prescribes and reporting violations as soon as possible. In other words, *monitoring* and *run-time verification* facilities must be provided by our framework.

DIAGNOSIS PHASE Since the diagnosis phase is carried out by taking into account only the execution traces provided by the different running instances of the system, then the same reasoning mechanisms used for the internal life cycle can be seamlessly applied to third party models, if they make their execution traces available. In this respect, $\mathcal{I}$ could play the role of a regulatory model used to analyze the execution traces of a third party.

2.8 GROUNDING THE FRAMEWORK

We propose to grounding a generic framework for the specification and verification of declarative open interaction models as depicted in Figure 11. We call our grounded framework Computational Logic for the verIfication and Modeling of Business processes and choreographies (CLIMB).

The CLIMB framework faces the management of open interaction models by integrating the ConDec language [158, 157], with the SCIFF formal framework [7], based on Computational Logic.

ConDec is a graphical language which adopts a declarative and open modeling style. It has been originally developed by Pesic and van der Aalst to model flexible BPs, but two variants of ConDec have been proposed also for modeling service flows/choreographies [186, 146] and clinical guidelines [149].

SCIFF is a framework based on computational logic originally thought for the formal specification and verification of open heterogeneous MAS. It gives a clear declarative semantics to interaction, and provides two corresponding proof procedures, which enable static as well as dynamic reasoning capabilities.

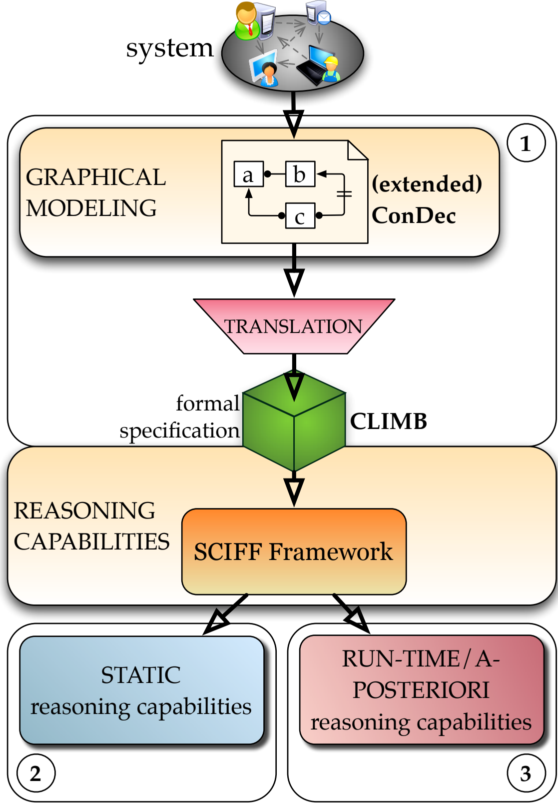


Figure 11: The CLIMB framework. Each number identifies the part of the dissertation covering the corresponding portion of the schema.

In the remainder of the dissertation, we show how the ConDec notation can be formalized by using a subset of the SCIFF language (called CLIMB language), and how the expressiveness of this language can be used to extend ConDec with new interesting features. We then show how the two proof procedures of the SCIFF framework could be jointly adopted (and adapted) to tackle all the desiderata pointed out in Section 2.7.

3

THE CONDEC GRAPHICAL LANGUAGE

Contents

3.1 ConDec in a Nutshell **35**
3.2 ConDec models **36**
3.3 Constraints **37**
3.3.1 Existence Constraints 37
3.3.2 Choice Constraints 38
3.3.3 Relation Constraints 39
3.3.4 Negation Constraints 41
3.3.5 Branching Constraints 43
3.4 A ConDec Choreography **43**
3.5 Usability of the Language **46**
3.6 Linear Temporal Logic **48**
3.6.1 LTL Models 49
3.6.2 Syntax of LTL 49
3.6.3 Semantics of LTL 50
3.7 Translation of ConDec to LTL **51**

ConDec is a graphical language proposed by Pesic and van der Aalst for the flexible, declarative and open modeling of Business Process (BP) models [157, 158]. They claim that these features are needed to fit with complex, unpredictable processes, where a good balance between support and flexibility must be found (see the discussion provided in Chapter 2). Variants of the ConDec language have been developed for dealing with service flows and choreographies [186, 146] and for capturing clinical guidelines [149].

In this Chapter, we provide a critical overview of the language, recalling its constructs and discussing its style of modeling in the light of the *cognitive dimensions* framework [92]. We then introduce the framework of propositional Linear Temporal Logic (LTL), summarizing how it has been exploited in [157, 158, 186, 146] to provide an underlying semantics for the ConDec constructs.

3.1 CONDEC IN A NUTSHELL

The term "ConDec" stands at the intersection of the words *constraint* and *declarative*.

ConDec is constraint-based

In fact, ConDec is a declarative, constraint-based language: instead of rigidly defining rigid the flow of interaction, it focuses on the (minimal) set of rules which must be satisfied in order to correctly carry out

the collaboration. Since rules constrain the way activities can be executed, they are referred to as constraints. Constraints reflect different kind of business knowledge: external regulations and norms, internal policies and best practices, business objectives, and so on.

ConDec is declarative

Differently from procedural specifications, in which activities can be inter-connected only by means of sequence patterns, mixed with constructs supporting the splitting/merging of control flows [191], the ConDec language provides a number of control-independent abstractions to constrain activities, alongside the more traditional ones. It is possible to insert past-oriented constraints, as well as constraints that do not impose any ordering among activities.

ConDec is open

Furthermore, while procedural specifications are closed, i.e., all what is not explicitly modeled is forbidden, ConDec models are open: activities can be freely executed, unless they are subject to constraints. This choice has two impacts. First, a ConDec model accommodates many different possible executions, improving flexibility. Second, the language provides abstractions to explicitly capture not only what is indispensable, but also what is forbidden. In this way, the set of possible executions does not need to be expressed extensionally and models remain compact: models specify the desired and undesired courses of interaction while leaving undefined other possibilities of interaction that are neither desired nor undesired.

3.2 CONDEC MODELS

A ConDec model is composed by a set of activities, which represent *atomic* units of work (i.e., units of work associated to single time points[1]), and relations among activities, used to specify constraints on their execution. Optional constraints are also supported, to express preferable ways to interact, but allowing the possibility to violate them.

Definition 3.1 (ConDec model). A ConDec model is a triple $\langle \mathcal{A}, \mathcal{C}_m, \mathcal{C}_o \rangle$, where:

- $\mathcal{A}$ is a set of *activities*, represented as boxes containing their name;
- $\mathcal{C}_m$ is a set of *mandatory* constraints;
- $\mathcal{C}_o$ is a set of *optional* constraints.

Given a ConDec model $\mathcal{CM}$, notations $\mathcal{A}^{\mathcal{CM}}$, $\mathcal{C}_m^{\mathcal{CM}}$ and $\mathcal{C}_o^{\mathcal{CM}}$ respectively denote the set of activities, mandatory and optional constraints of $\mathcal{CM}$.

Development of ConDec models

A ConDec model is developed in two steps:

1 In [157], a non-atomic model of activities is presented. However, it cannot be formalized by means of LTL [157]. We therefore make the assumption that the basic version of the language deals only with atomic activities, delegating the discussion about non-atomic ones in Section 6.2.2.

A. The relevant activities of the system are identified and placed in the model. At this stage, the model is completely unconstrained, and therefore each activity can be performed an arbitrary number of times, in whatever order.

B. ConDec constraints are added to reflect the different constraints of the system, restricting the set of allowed executions.

Support of execution traces

In the following, we provide a natural language description of all the ConDec constraints. Note that each constraint is defined in terms of requirements that it imposes on a given execution trace in order to evaluate it as compliant. An execution trace is then *supported* by the overall model iff each mandatory constraint of the model is respected.

DEFINITION 3.2 (Supported execution trace). Given an execution trace $\mathcal{T}$ and a ConDec model $\mathcal{CM}$, $\mathcal{T}$ is *supported* by $\mathcal{CM}$ iff $\mathcal{T}$ complies with each constraint belonging to $\mathcal{C}_m^{\mathcal{CM}}$.

3.3 CONSTRAINTS

ConDec constraints are grouped into four families:

EXISTENCE CONSTRAINTS are *unary cardinality* constraints expressing how many times an activity can/should be executed;

CHOICE CONSTRAINTS are *n-ary* constraints expressing the *necessity* to execute some activities between a set of possible *choices*, independently from the rest of the model;

RELATION CONSTRAINTS are *binary* constraints which impose the *presence* of a certain activity when some other activity is performed, possibly imposing also *temporal constraints* on such a presence;

NEGATION CONSTRAINTS are the negative version of relation constraints, and are employed to explicitly *forbid* the execution of a certain activity when some other activity is performed.

ConDec has a qualitative notion of time

Note that, in ConDec, the characterization of time is qualitative (see Section 2.1.2): temporal constraints could express the required relative positions among the occurrence of two activities, but they are not able to deal with metric distances between them.

3.3.1 *Existence Constraints*

Existence constraints are unary cardinality constraints predicating on the number of possible executions of an activity. They graphically resemble the UML cardinality constraints, and represent either the minimal, the exact or the maximum executions of an activity. The init constraint has a slightly different meaning: it is used to identify the starting activity of the model.

NAME	GRAPHICAL	MEANING
`absence(a)`	0 [a]	Activity a cannot be executed.
`absence(n+1,a)`	0..n [a]	Activity a can be executed *at most* n times, i.e., the execution trace cannot contain $n+1$ occurrences of a.
`existence(n,a)`	n..∗ [a]	Activity a must be executed *at least* n times.
`exactly(n,a)`	n [a]	Activity a must be executed *exactly* n times.
`init(a)`	init [a]	Activity a must always be the first activity to be executed.

Table 4: ConDec existence constraints.

Table 4 summarizes the different existence constraints, showing their graphical representation.

An existenceN constraint could be employed to state that *at least one* item must be ordered by the customer during the interaction:

1..∗
order item

On the contrary, an absenceN constraint could be adopted to state that at most three payment attempts can be made by the customer:

0..3
pay

3.3.2 *Choice Constraints*

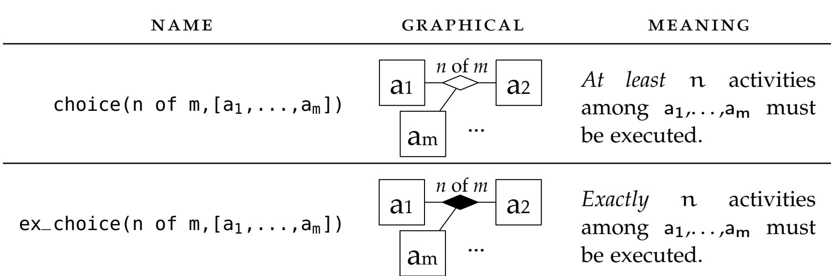
Choice constraints are shown in Table 5. They are n-ary constraints asserting that the interacting entities must necessarily perform certain activities, selecting them among a set of possible choices. They can be therefore seen as an extension of existenceN/exactlyN constraints, whose single involved activity is replaced with a set of activities. Choice constraints are very useful when the model contains different possible activities, which actually accomplish the same business goal: interacting entities are free to choose the most suitable on their own.

An *1 of 2* choice could be used to state that a contributing author must submit her paper either by uploading it through the web site of the conference or by sending it via e-mail[2]:

upload paper —◇— e-mail paper

Note that the constraint supports an execution trace containing three submissions of the paper (via e-mail and/or by using the web site), while it evaluates an execution with no submission as non-compliant.

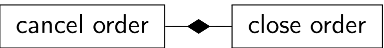
2 We suppose that when $n = 1$, the *1 of m* notation can be omitted.



NAME	GRAPHICAL	MEANING
`choice(n of m,[a1,...,am])`	a1, n of m, a2, am, ...	*At least* n activities among $a_1,\ldots,a_m$ must be executed.
`ex_choice(n of m,[a1,...,am])`	a1, n of m, a2, am, ...	*Exactly* n activities among $a_1,\ldots,a_m$ must be executed.

Table 5: ConDec choice constraints.

An *1 of 2* exclusive choice can be instead adopted to state that the interacting entities must choose between two alternatives, but such alternatives exclude each other. For example, it could be used to state that, within an instance of the system, the customer is always committed to cancel or successfully close the order, but not both.



Goal-oriented constructs

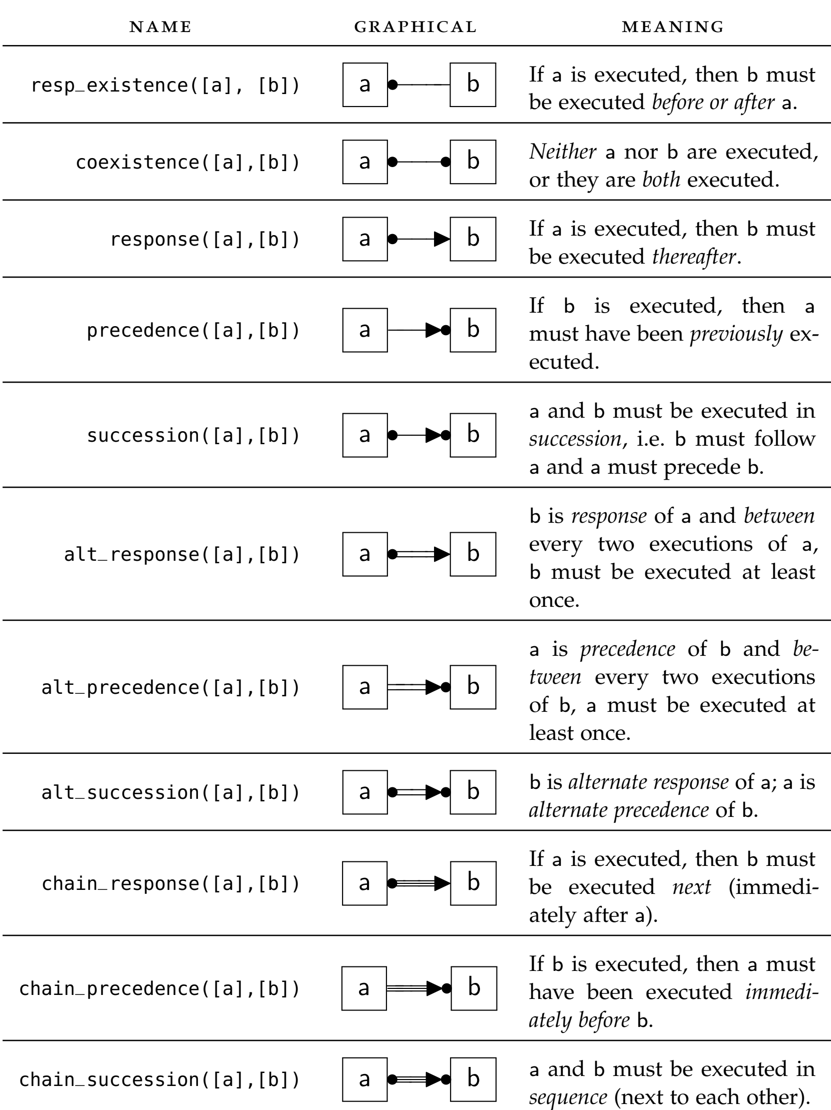
In conclusion, existenceN, exactlyN and choice constraints could be considered as *goal-oriented* constructs: they impose the presence of (some of) the involved activities in any possible execution trace, independently from the other parts of the model.

3.3.3 *Relation Constraints*

Differently from goal-oriented constraints, relation constraints express positive dependencies among activities; in their simplest form, they interconnect two activities (the generalized case will be discussed in Section 3.3.5). As depicted in Table 6, the graphical representation of the connection communicates to the user the meaning of the relationship, depending on the number of lines, on where • elements are positioned, and on the presence/absence of an arrow. In particular, what changes moving from one relationship to another are the qualitative time constraints involved in the relationship. Let us discuss such an issue in more detail.

Triggering of relation constraints

When a relation constraint associates a • to an activity, then each occurrence of that activity (let us call it the constraint's source) has the ability to *trigger* the constraint, imposing that also the activity connected on the other side must be executed (let us call this latter activity the constraint's target). In other words, relation constraints are *reactive*. Starting from this basic schema, each relation constraint then characterizes *when* the occurrence of b is expected to happen. The responded existence and coexistence relation constraints do not impose any temporal ordering, but rather let the interacting entities completely free to decide when executing the target. For example, let us consider the case of a seller which relies on a warehouse to deliver the sold goods. When the warehouse ships a good, then it must also notify the seller that the amount of stored goods has changed; the notification could be sent



NAME	GRAPHICAL	MEANING
resp_existence([a], [b])		If a is executed, then b must be executed *before or after* a.
coexistence([a],[b])		*Neither* a nor b are executed, or they are *both* executed.
response([a],[b])		If a is executed, then b must be executed *thereafter*.
precedence([a],[b])		If b is executed, then a must have been *previously* executed.
succession([a],[b])		a and b must be executed in *succession*, i.e. b must follow a and a must precede b.
alt_response([a],[b])		b is *response* of a and *between* every two executions of a, b must be executed at least once.
alt_precedence([a],[b])		a is *precedence* of b and *between* every two executions of b, a must be executed at least once.
alt_succession([a],[b])		b is *alternate response* of a; a is *alternate precedence* of b.
chain_response([a],[b])		If a is executed, then b must be executed *next* (immediately after a).
chain_precedence([a],[b])		If b is executed, then a must have been executed *immediately before* b.
chain_succession([a],[b])		a and b must be executed in *sequence* (next to each other).

Table 6: ConDec relation constraints.

either before or after the actual shipment. In this setting, a responded existence could be used to connect the shipment and the notification activities:

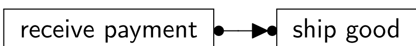
ship good —— notify seller

If at the time at which the good is shipped, the seller has been already notified, then the constraint is automatically fulfilled, otherwise it "waits" for the notification.

Response and precedence instead impose the two fundamental qualitative temporal orderings, i.e., *after* and *before*. Alternate constraints restrict the time window inside the target must occur; for example, the alternate response constraint states that the target must be executed after the occurrence of the source, but before a new occurrence of the source. Chain constraints impose even a more strict ordering relations among the two occurrences: they must be next to each other. Summarizing, relation constraints range from the completely open approach of the responded existence constraint to the completely closed perspective of the chain succession, which resembles the *sequence* connection of classical procedural languages.

Finally, it is worth noting that succession constraints are a shortcut for representing the combination of the corresponding response and precedence relations, as well reflected by their graphical presentation. In this case, each activity is at the same time source and target of the constraint.

For example, a "normal" succession constraint could be employed to state that when a payment is received, then the warehouse must eventually deliver the paid good, and, conversely, the warehouse will ship the good only if a payment has been previously done:

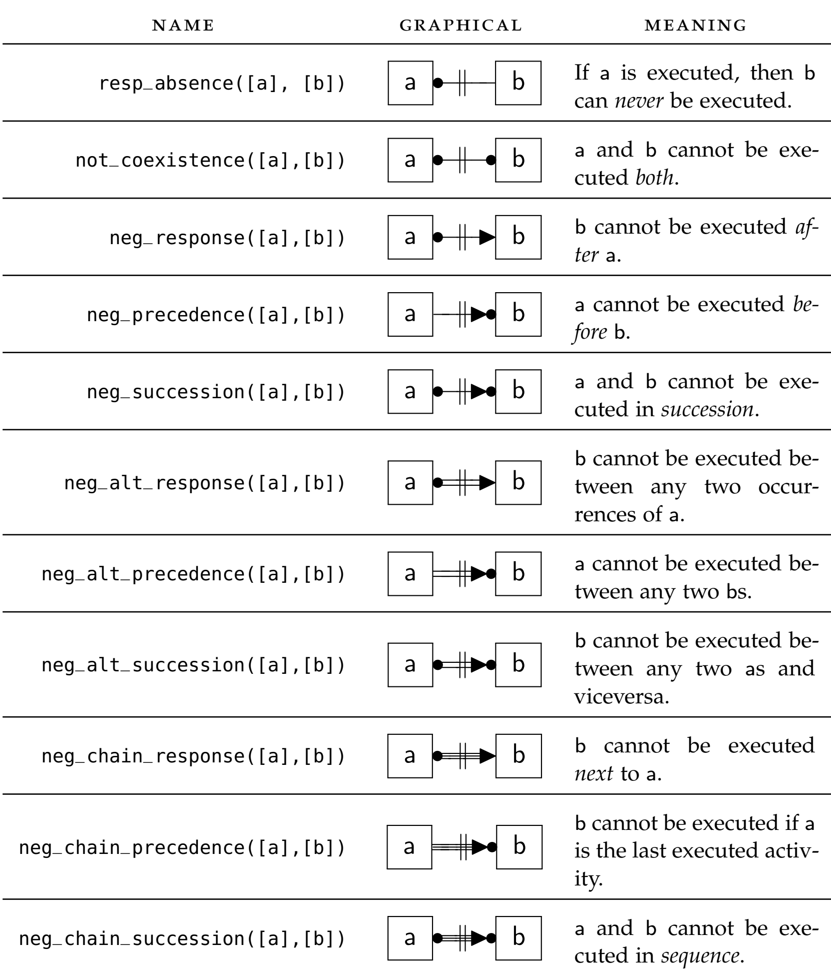


Since the succession constraint imposes a loosely-coupled time ordering among the involved occurrences, other activities can be performed by the warehouse between the two.

3.3.4 *Negation Constraints*

As shown in Table 7, negation constraints are specular to relation constraints: when a negation constraint is triggered by its source activity, then it imposes that the target activity cannot be executed between certain time bounds, determined by the the specific nature of the constraint. As a consequence, negation constraints "invert" the open and close nature of relation constraints: a —— b states that when activity a is executed, b can be executed if at least one intermediate occurrence of another activity exists; on the contrary, a —— b forbids the presence of b along the whole instance, if a is performed inside that instance.

Negation constraints are a peculiar feature reflecting the open nature of ConDec. Since by default all activities can be executed in any order,

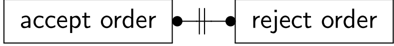


NAME	GRAPHICAL	MEANING
resp_absence([a], [b])		If a is executed, then b can *never* be executed.
not_coexistence([a],[b])		a and b cannot be executed *both*.
neg_response([a],[b])		b cannot be executed *after* a.
neg_precedence([a],[b])		a cannot be executed *before* b.
neg_succession([a],[b])		a and b cannot be executed in *succession*.
neg_alt_response([a],[b])		b cannot be executed between any two occurrences of a.
neg_alt_precedence([a],[b])		a cannot be executed between any two bs.
neg_alt_succession([a],[b])		b cannot be executed between any two as and viceversa.
neg_chain_response([a],[b])		b cannot be executed *next* to a.
neg_chain_precedence([a],[b])		b cannot be executed if a is the last executed activity.
neg_chain_succession([a],[b])		a and b cannot be executed in *sequence*.

Table 7: ConDec negation constraints.

it would be unfeasible to assume that "positive" constraints suffice to express significative interaction models: it is sometimes much more easy and compact to capture what are the undesired behaviours rather than the allowed ones.

For example, the not coexistence constraint, which is very difficult to be captured in a closed procedural setting (as the other negation constraints)[3], is a first-class object in ConDec. It could be employed to express that two activities are incompatible, i.e., for example, that a seller cannot accept and reject an incoming order:

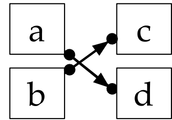


3 See the discussion made in Section 2.2.1, or refer to [186].

3.3.5 *Branching Constraints*

As far as now, we have considered only relation and negation constraints involving two activities. Such constraints can be generalized, by interconnecting multiple activities. When a constraint branches on multiple activities, the intended interpretation is *disjunctive*:

- In case of *multiple sources*, the constraint is triggered by either one of the source activities.
- In case of *multiple targets*, when the relation (negation) constraint is triggered, it can be satisfied by (not) executing either one of the target activities;
- Succession constraints maintain their decomposition into a response and precedence part; for example, in constraint

  the activity a plays the role of a disjunctive source w.r.t. the response part of the succession, and it plays the role of a disjunctive target w.r.t. the precedence part of the succession.

The only exception is constituted by the coexistence constraint, for which sources and targets are indistinguishable when looking at its graphical representation. The n-ary interpretation of such a constraint is: whenever one among the n interconnected activities occur, then at least one of the other $n-1$ involved activities must be executed as well.

Summarizing, branching constraints enable the possibility to combine the reactive nature of relation and negation constraints with the flexibility of choice constraints.

3.4 a condec choreography

We illustrate the power of ConDec in modeling a simple yet significative service choreography, which extends the fragment discussed in Section 2.3.3.

Let us consider a choreography that aims to define/constrain the behaviour of some parties when submitting an "order" for a set of items. The choreography includes three different roles: a *customer* which interacts with a *seller* to place an order of a set of items, and a *warehouse* which could participate to the interaction by communicating to the seller if it is able to ship the ordered items or not. We distinguish different interactions among many services that follow the choreography by assuming that there is a unique identifier (frequently called *instance identifier*) for each interaction: e.g., in this choreography we can assume that all the executed activities related to the same interaction are identified by means of an order number, intuitively referring to the

SOURCE	CONSTRAINT NAME		TARGET	DESCRIPTION
cancel order	C_1	negation response	choose item	in case of cancelation, the user cannot choose other items [...] anymore
	C_2	not coexistence	commit order	a canceled order cannot be committed (and vice-versa)
	C_3	precedence	choose item	an order is made up by at least one chosen item
commit order	C_4	response	refuse or confirm order	after having committed an order, the customer expects a positive or negative answer from the seller
	C_5	precedence	confirm shipment	the seller could confirm the order only if the warehouse has previously confirmed the shipment
	C_6	precedence	choose item	an order is made up by at least one chosen item
refuse shipment	C_7	responded existence	refuse order	if the warehouse is unable to execute the shipment, then the seller should refuse (or have refused) the order
	C_8	absence(1)	receipt delivery	the seller will deliver a single receipt

Table 8: Mapping the statements of the Customer-Seller-Warehouse in ConDec.

concept of "order". Each interaction among services that aim to follow the choreography represents an instance.

EXAMPLE 3.1 (The Customer-Seller-Warehouse Scenario). *The customer makes up an order by choosing one or more items from the seller list. Before committing an order, it is always possible to cancel it; in this case, the user cannot choose other items within the same instance anymore, and the choreography instance terminates. After having committed an order, the customer expects a positive or negative answer from the seller. The seller could freely decide whether to confirm or refuse customer's order, but sometimes it has also to consider the opinion of the warehouse about the shipment. In particular:*

- *the seller can confirm the order only if the warehouse has previously confirmed the shipment;*
- *if the warehouse states that it is unable to execute the shipment, then the seller should refuse (or have refused) the order.*

When the order is committed, in case of a confirmation from the seller the order becomes established*, and the customer is in charge to pay for it. Conversely, when the seller receives the payment for an established order, she must deliver a single corresponding receipt. However, the seller is free to change her*

SOURCE	CONSTRAINT NAME		TARGET	DESCRIPTION
refuse order	C_9	precedence	commit order	An answer from the seller is valid only if it is performed after having committed the order
confirm order	C_{10}	precedence	commit order	
payment	C_{11}	precedence	confirm order	A valid payment should be preceded by the confirmation of the order
deliver receipt	C_{12}	precedence	payment	The receipt should be delivered only if the order has been paid
refuse order	C_{13}	negation response	confirm order	While a confirmed order can be refused, the opposite should not happen (nothing is said in the example)
TARGET	CONSTRAINT NAME		TARGET	DESCRIPTION
confirm shipment	C_{14}	not co-existence	refuse shipment	Possible warehouse's answers are mutually exclusive
commit order	C_{15}	cardinality 0..1		the choreography centres around the concept of a single order, which could possibly be canceled
commit order	C_{16}	1 of 2 choice	cancel order	

Table 9: Implicit ConDec constraints involved in the Customer-Seller-Warehouse example.

mind at any time: she can refuse an already established order. Clearly, if the payment has already been done, this exceptional situation must be handled by the seller, which is committed to refund the customer.

Methodology

To model the example in ConDec, a first step is to rearrange all the involved constraints in a table, making the activities explicit and linking the natural language description to a corresponding ConDec construct. Table 8 shows the result of this process.

By analyzing Example 3.1, we could infer some further useful constraints, reflecting common-sense or implicit knowledge, to properly complete Table 8. The added constraints are shown in Table 9, while in Figure 12 the whole set of constraints is shown using the ConDec notation.

Instance identifier

Among the added constraints, C_{15} and C_{16} deal with the core concept of the choreography, which is actually the commitment of one order. Since the order could also be canceled, we attach a `0..1` constraint (i.e., C_{15}) to the commit order activity (to express that at most

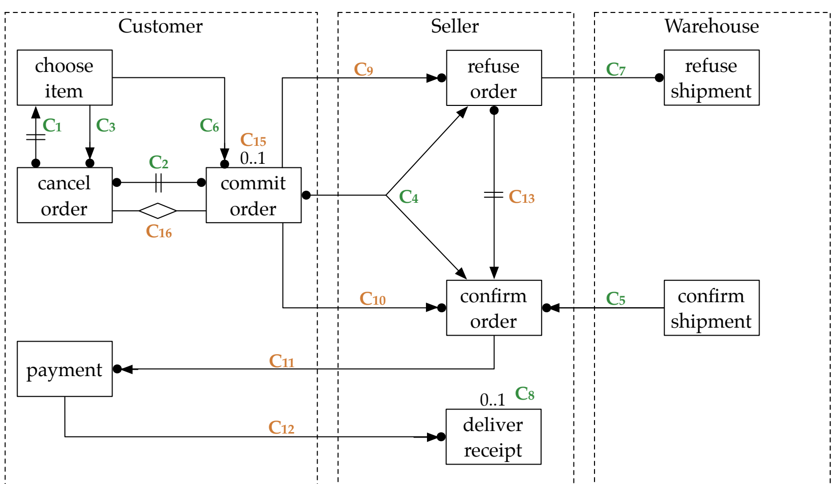


Figure 12: ConDec diagram capturing the Customer-Seller-Warehouse example; for the sake of readability, the three interacting roles are shown as pools in the diagram.

CLOSENESS OF MAPPING	Closeness of representation to domain
ABSTRACTION	Types and availability of abstraction mechanisms
CONSISTENCY	Similar semantics are expressed in similar syntactic forms
HIDDEN DEPENDENCIES	Important links between entities are not visible
PREMATURE COMMITMENT	Constraints on the order of doing things
PROGRESSIVE EVALUATION	Work-to-date can be checked at any time

Table 10: Some cognitive dimensions.

one order can be committed for each choreography instance), and bind the cancelation and the commitment with a choice ConDec relation (i.e., constraint C_{16}), which states that at least one of the two connected activities has to be executed: an order should be committed or canceled.

3.5 USABILITY OF THE LANGUAGE

Cognitive dimensions

Although a deep and extensive analysis of ConDec from the end-users viewpoint has not yet been carried out[157], we will try to briefly review its usability in terms of some cognitive dimensions, whose definition is briefly listed in Table 10[4]. Cognitive dimensions [92] are a useful tool to subjectively assess the usability of languages and notations in an easy-to-comprehend way. They have been applied to a broad range of programming languages and environments/editors, also visual [93].

4 See http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf

concept	notation
unary constraints	cardinality constraints a lá UML
source of a relation/negation constraint	•
negation	‖
temporal ordering	—►
relationship's strength	– (normal) = (alternate) ≡ (chain)
succession representation	response + precedence part

Table 11: Core ConDec graphical elements and their corresponding meaning.

Closeness of mapping

The main strength of ConDec relies on the *closeness of mapping* between the notation and the problem of capturing constraints: it provides a wide range of *abstractions* to constrain the execution of activities in many different ways, overcoming both over-constraining and over- specification issues. ConDec diagrams can range from classical procedural models (by only using the *chain succession* relation) to loosely-coupled ones. As shown in Section 3.4, many different natural language statements can be straightforwardly represented as ConDec constraints.

Consistency

Another valuable feature of ConDec is the *consistency* of its constraints: they have a representation which coherently combines only the few basic intuitive visual elements shown in Table 11. For example, the representation of succession relationships can be easily inferred from other constructs: both semantics and representation of this kind of constraint is determined by combining the semantics and representation of the corresponding response and precedence ones.

Hidden dependencies

Even though ConDec combines simple concepts, rendered in a consistent way, when the complexity of models grows, their readability would quickly be compromised. The semantics of a ConDec model is determined by the combination of all its constraints: the user is driven to adopt a non-structured approach to modeling. While avoiding *premature commitments*, this methodology has the main drawback that the overall meaning tends to become unclear; because of the interplay between the different constraints, many *hidden dependencies* among activities are implicitly introduced.

To better clarify the problem, let us consider the simple example of Figure 13.

Suppose that activity c is executed; constraint [c with cardinality 0..1] forbids further executions of c in the same instance. However, such a forbidding implicitly impacts also on the execution of activities a and b. In fact, the

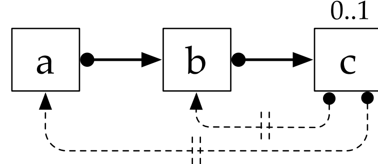


Figure 13: Hidden dependencies in a ConDec model (shown as dashed connections).

response constraints state that the execution of a or b imposes that c is executed afterwards as well; this would be impossible. In other words, the interplay between the absence and response constraints introduces two hidden negation responses relating activity c with a and b respectively. In complex cases, such an interplay could lead to produce inconsistent or incorrect models.

Progressive evaluation

In this respect, ConDec models demand a constant support along their entire life cycle, ranging from their static verification to their enactment and a-posteriori analysis, for the sake of *progressive evaluation*. Only if a suitable support is provided, the hidden dependencies implicitly introduced during the design will not undermine the correctness and consistency of the model.

3.6 LINEAR TEMPORAL LOGIC

We introduce the framework of propositional Linear Temporal Logic (LTL), which has been exploited by Pesic and van der Aalst to provide an underlying semantics to ConDec [157, 158].

Temporal logics are a special class of Modal Logics where modalities are interpreted as temporal operators, used to describe and reason about how the truth values of assertions vary with time [73].

Classification of temporal logics

According to [73], temporal logics can be classified along a number of axes[5]:

A. *propositional* vs *first-order*

B. *global* vs *compositional*

C. *linear* vs *branching*

D. *point* vs *interval-based*

E. *discrete* vs *continuous*

F. *past* vs *tense*

Where not explicitly stated, in the following we will refer to propositional Linear Temporal Logic (LTL) with future-tense operators only. Hence, in our setting LTL is:

5 Obviously, such axes resembles the different aspects aiming at characterize the notion of time within a system – see Section 2.1.2.

QUALITATIVE Temporal operators are used to express qualitative time relations between events. Metric distances are not part of the logic.

PROPOSITIONAL Constraints are built up from *atomic propositions*, whose truth values change with the flow of time.

LINEAR A *linear* course of time means that there is only one possible future moment for each current situation, and modalities are exploited for describing the truth of propositions along a single timeline.

POINT-BASED Temporal operators and propositions are evaluated over *points* in time.

DISCRETE Time is *discrete*; the present moment correspond to the current state of the system and the next moment to the immediate successor state (i.e., the first state in the future at which some event occur, making the system to evolve).

FUTURE-TENSE Temporal operators are used to describe the occurrence of events in the future. This is reasonable for systems which have a definite starting time. All the systems considered in this thesis show this feature.

3.6.1 *LTL Models*

In accordance with the described properties, in LTL the *time structure* (also called *frame*) $\mathcal{F}$ models a single, linear timeline; formally, $\mathcal{F}$ is a totally ordered set $(\mathcal{S}, \prec)$.

Time structure

Let $\mathcal{P}$ be the set of all atomic propositions in the system. An LTL model $\mathcal{M}$ is a triple $(\mathcal{S}, \prec, v)$ where $v : \mathcal{P} \rightarrow 2^{\mathcal{S}}$ is a *valuation function* which maps each proposition in $\mathcal{P}$ to the set of time instances at which the proposition holds.

LTL model

Valuation function

In the context of ConDec, propositions represent the possible events that can happen in the system. A proposition $e \in \mathcal{P}$ is true in a certain state if at that state the event denoted by e occurs.

LTL models as execution traces

DEFINITION 3.3 (LTL execution trace). An LTL execution trace $\mathcal{T}_{\mathcal{L}}$ is defined as an LTL model having $\mathbb{N}$ as time structure and $\mathcal{E}$, the set of all events which can potentially occur during the execution of the system, as the set of atomic propositions: $\mathcal{T}_{\mathcal{L}} = (\mathbb{N}, <, \nu_{occ})$, where $\nu_{occ} : \mathcal{E} \rightarrow 2^{\mathbb{N}}$ is a valuation function mapping each event $e \in \mathcal{E}$ to the set of all timestamps at which e occurred.

For convenience, in the following $\mathcal{T}_{\mathcal{L}}(i)$ will denote the $i-th$ state of $\mathcal{T}_{\mathcal{L}}$; $e \in \mathcal{T}_{\mathcal{L}}(i)$ will be adopted as a shortcut for $i \in \nu_{occ}(e)$.

3.6.2 *Syntax of LTL*

LTL formulae are defined by using atomic propositions (i.e., events and

LTL operators

OP.	NAME	MEANING	INTUITION
$\bigcirc\phi$	Next time	ϕ will hold in the next future moment	
$\Diamond\phi$	Eventually	ϕ will hold sometimes in the future	
$\Box\phi$	Globally	ϕ will always hold in the future	
$\psi\mathcal{U}\phi$	Until	ϕ will hold in a future moment, and ψ holds until that moment	

Table 12: LTL temporal operators.

the two special constants $\mathtt{true}$ and $\mathtt{false}$), classical propositional connectives ($\neg$, $\wedge$, $\wedge$ and $\Rightarrow$) and temporal operators ($\bigcirc$, $\mathcal{U}$, $\Diamond$, $\Box$ and $\mathcal{W}$).

An LTL formula is recursively defined as follows:

- each atomic proposition (i.e., event) $e \in \mathcal{E}$ is a formula;
- if ϕ is a formula, then $\neg\phi$ is a formula;
- if ϕ and ψ are formulae, then $\phi \wedge \psi$ is a formula;
- if ψ is a formula, then $\bigcirc\psi$ is a formula;
- if ϕ and ψ are formulae, then $\phi\mathcal{U}\psi$ is a formula.

Other LTL formulae can be defined as abbreviations:

- $\phi \vee \psi \triangleq \neg(\neg\phi \wedge \neg\psi)$ and $\phi \Rightarrow \psi \triangleq \neg\phi \vee \psi$;
- $\mathtt{true} \triangleq \neg\phi \vee \phi$ and $\mathtt{false} \triangleq \neg\mathtt{true}$;
- $\Diamond\phi \triangleq \mathtt{true}\mathcal{U}\phi$, $\Box\phi \triangleq \neg\Diamond\neg\phi$ and $\psi\mathcal{W}\phi \triangleq \psi\mathcal{U}\phi \vee \Box\psi$.

Priority of operators

When parentheses are omitted, the following priority of operators holds: first all the temporal operators, followed by $\neg$, $\wedge$, $\vee$ and finally $\Rightarrow$.

3.6.3 *Semantics of LTL*

The semantics of LTL is given w.r.t. an LTL model, in a given state. We will use $\models_{\mathcal{L}}$ to denote entailment in the LTL setting. $\mathcal{M}, i \models_{\mathcal{L}} \phi$ means that ϕ is true at time i in $\mathcal{M}$. The intuitive semantics of each temporal operator is described in Table 12. $\models_{\mathcal{L}}$ is defined by induction on the structure of the formulae[6]:

- $(\mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \phi)$ iff $(\mathcal{T}_{\mathcal{L}}, 0 \models_{\mathcal{L}} \phi)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} e)$ iff $e \in \mathcal{T}_{\mathcal{L}}(i)$ (i.e., $i \in \nu_{occ}(e)$);

6 For the sake of readability, we explicitly denote the semantics of $\Diamond$, $\Box$ and $\mathcal{W}$, even if their meaning can be obtained from the semantics of $\mathcal{U}$.

- $(\mathcal{T}_{\mathcal{L}}, i \not\models_{\mathcal{L}} e)$ iff $e \notin \mathcal{T}_{\mathcal{L}}(i)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \phi \wedge \psi)$ iff $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \phi)$ and $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \psi)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \phi \vee \psi)$ iff $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \phi)$ or $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \psi)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \phi \Rightarrow \psi)$ iff $(\mathcal{T}_{\mathcal{L}}, i \not\models_{\mathcal{L}} \phi)$ or $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \psi)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \bigcirc\phi)$ iff $(\mathcal{T}_{\mathcal{L}}, i+1 \models_{\mathcal{L}} \phi)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \psi \mathcal{U} \phi)$ iff $\exists k \geqslant i$ s.t. $(\mathcal{T}_{\mathcal{L}}, k \models_{\mathcal{L}} \phi)$ and $\forall i \leqslant j < k$ $(\mathcal{T}_{\mathcal{L}}, j \models_{\mathcal{L}} \psi)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \Diamond\phi)$ iff $\exists j \geqslant i$ s.t. $(\mathcal{T}_{\mathcal{L}}, j \models_{\mathcal{L}} \phi)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \Box\phi)$ iff $\forall j \geqslant i$ $(\mathcal{T}_{\mathcal{L}}, j \models_{\mathcal{L}} \phi)$;
- $(\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \psi W \phi)$ iff either $\exists k \geqslant i$ s.t. $(\mathcal{T}_{\mathcal{L}}, k \models_{\mathcal{L}} \phi)$ and $\forall i \leqslant j < k$ $(\mathcal{T}_{\mathcal{L}}, j \models_{\mathcal{L}} \psi)$, or $\forall j \geqslant i$ $(\mathcal{T}_{\mathcal{L}}, j \models_{\mathcal{L}} \phi)$.

3.7 TRANSLATION OF CONDEC TO LTL

As pointed out in [157, 158], all the ConDec constraints can be formalized in LTL. In fact, the ConDec language itself has been developed starting from the work of Dwyer et al. [72], in which a pattern-based analysis of the most diffused LTL formulae is presented. In this light, ConDec can be seen as a graphical front-end, supporting non-IT savvy in the rigorous formal specification of interaction models avoiding the direct manipulation of logical formulae.

We suppose that the translation of an arbitrary ConDec model to the underlying LTL formalization is embedded in a translation function called t_{LTL}.

DEFINITION 3.4 (ConDec to LTL translation). t_{LTL} is a function which translates a ConDec model $\mathcal{CM} = \langle \mathcal{A}, \mathcal{C}_m, \mathcal{C}_o \rangle$ to an LTL *conjunction* formula as follows:

$$t_{LTL} : \qquad \mathcal{CM} \longmapsto t_{LTL}(\mathcal{CM}) = \bigwedge_{C_i \mid C_i \in \mathcal{C}_m} t_{LTL}(C_i)$$

where the application of t_{LTL} to a ConDec mandatory constraint follows the guidelines provided in [157, 158].

As pointed out in the definition, the translation of the whole model is a *conjunction formula* embracing the translation of each single constraint. Indeed, an execution trace supported by the model must comply with all the constraints of the model.

Having provided a characterization of execution traces and ConDec model in the LTL setting, the LTL entailment can be used to give a formal account to the notion of *compliance* of a trace w.r.t. a set of constraints, which in turn defines the concept of *support* provided by a ConDec model w.r.t. a given trace (see Definition 3.2).

DEFINITION 3.5 (Supported execution trace in the LTL setting). Given an LTL execution trace $\mathcal{T}_{\mathcal{L}}$ and a ConDec model $\mathcal{CM}$, $\mathcal{T}_{\mathcal{L}}$ is *supported* by $\mathcal{CM}$ iff:

$$\mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} t_{LTL}(\mathcal{CM})$$

EXAMPLE 3.2 (Translation of a ConDec constraint to LTL and trace evaluation). *Let us consider a ConDec model $\mathcal{CM}$ containing only a single response constraint between the two* query *and* answer *activities. It is translated to LTL as follows:*

$$t_{LTL}(\mathcal{CM}) \triangleq t_{LTL}\left(\boxed{\text{query}} \bullet\!\!\longrightarrow \boxed{\text{answer}}\right) \triangleq \Box(query \Rightarrow \Diamond answer)$$

In accordance with response's intuitive meaning, the LTL formalization states that in every state of an instance of the system, it must be true that if query *is performed, then there must be a following state in which the* answer *is provided. Formally, by applying the LTL declarative semantics (reported in Section 3.6.3) and Definition 3.5, an LTL execution trace $\mathcal{T}_{\mathcal{L}}$ is supported by such a response constraint iff*

$$\begin{aligned} \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \Box(query \Rightarrow \Diamond answer) &\Leftrightarrow \\ \forall\, i \geqslant 0,\ (\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} query \Rightarrow \Diamond answer) &\Leftrightarrow \\ \forall\, i \geqslant 0,\ query \notin \mathcal{T}_{\mathcal{L}}(i) \vee (\mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \Diamond answer) &\Leftrightarrow \\ \forall\, i \geqslant 0,\ query \notin \mathcal{T}_{\mathcal{L}}(i) \vee (\exists\, j \geqslant i,\ answer \in \mathcal{T}_{\mathcal{L}}(j)) & \end{aligned}$$

Therefore, it holds that:

- *The empty trace is supported by $\mathcal{CM}$, because all its states do not contain the execution of the* query *activity.*
- *The trace containing only an execution of an "external activity" (i.e., an activity different than* query *and* answer*) is supported as well, because of the same reason; this example attests the openness of ConDec.*
- *The trace described by $\nu_{occ}(query) = \{0\}$ is not supported, because the response constraint requires the presence of an* answer *in at least one state following 0.*
- *The trace described by $\nu_{occ}(query) = \{0, 1\}$, $\nu_{occ}(answer) = \{2\}$ is supported, because for both $i = 0$ and $i = 1$ there exists a following j ($j = 2$ in particular) contained in the valuation function of the* answer *activity.*

4

THE CLIMB RULE-BASED LANGUAGE

Contents

4.1 The CLIMB Language in a Nutshell **54**
4.2 The CLIMB Syntax **55**
4.2.1 Event Occurrences and Execution Traces 55
4.2.2 Constraint Logic Programming 57
4.2.3 Expectations 58
4.2.4 Integrity Constraints 60
4.2.5 The Static Knowledge Base 64
4.2.6 SCIFF-lite and Composite Events 66
4.3 CLIMB Declarative Semantics **67**
4.3.1 Abduction 67
4.3.2 Abductive Logic Programming 69
4.3.3 Representing a system and its executions 70
4.3.4 SCIFF-lite Specifications 72
4.3.5 A Declarative Notion of Compliance 73
4.4 Equivalence and Compositionality **77**
4.4.1 Equivalence w.r.t. Compliance 78
4.4.2 Compositionality w.r.t. compliance 79

The CLIMB (Computational Logic for the verIfication and Modeling of Business processes and choreographies) language is a declarative rule-based language for the specification of Event-Based Systems (EBSs). This chapter describes syntax and features of the language, showing how it can suitably deal with static as well as dynamic aspects of EBSs; the declarative semantics of the language is presented, providing a formal characterization of *compliance*; finally, some interesting properties, such as *compositionality* of CLIMB specifications, are investigated.

The CLIMB language is a first-order logic-based language; quantification is left implicit in the language, in order to facilitate readability and usability. We will briefly discuss in an informal way and by means of example how variables are quantified. For an exhaustive description about the quantification of variables and the restrictions imposed on the language the interested reader may refer to [7, 44].

In the remainder of the dissertation, we assume that the reader is familiar with First Order Logic (FOL), Logic Programming (LP) and Prolog. A good introduction to LP is the book by Lloyd [122]; good introductions to Prolog are the books by Sterling and Shapiro [181] and by Bratko [35].

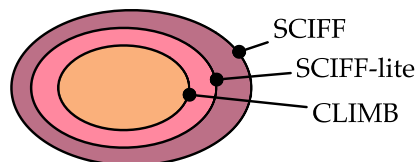


Figure 14: Variants of the SCIFF language and their expressiveness.

4.1 THE CLIMB LANGUAGE IN A NUTSHELL

As shown in Figure 14, the CLIMB syntax is a subset of the Social Constrained IFF Framework (SCIFF) language [7], which has been developed in the context of the SOCS European Project[1].

The SOCS EU Project

The purpose of the SOCS EU Project was to develop a logic-based framework for the specification and verification of open and hetorogeneous Multi-Agent Systems (MAS), ranging from the implementation of autonomous intelligent agents (called *computees*) to the engineering of their interaction. Within the project, a large part was focused on the *inter-agent* perspective, i.e., on the interaction protocols regulating the exchange of communicative acts among computees and other external, unknown entities.

Being these systems open and heterogeneous, no assumption about the number of interacting agents nor about their internal implementation can be made. Hence, the problem of specifying and verifying interaction requires an abstraction step, to identify what pieces of information could be effectively observed and, consequently, targeted by the specification language. The idea is to characterize (an execution instance of) the system in terms of *events* that are generated by the involved entities when acting on the environment or interacting with each other (see Figure 15). Thus, in general SCIFF can be applied to any dynamic context whose executions can be described in terms of events.

CLIMB specifications

CLIMB specifications are mainly composed by two parts:

A. a set of *rules* which relate events occurring during the execution with expectations describing the ideal behaviour of the system, and that are used to characterize the *dynamic aspects* of the system;

B. a *knowledge base* formalizing the *static aspects* of the system, i.e. the background knowledge independent of the specific executions of the system.

1 Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. IST-2001-32530, 2002-2005.
Web-page: `http://lia.deis.unibo.it/research/socs/`

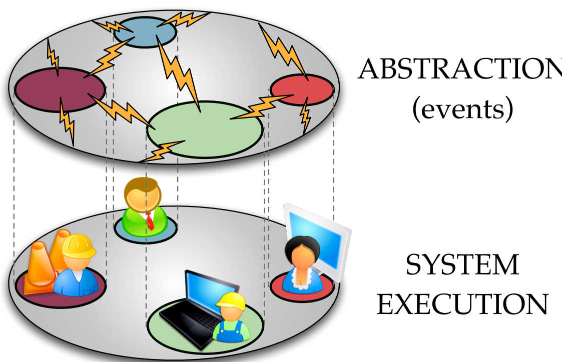


Figure 15: Abstraction of a system execution in terms of occurring events.

4.2 THE CLIMB SYNTAX

4.2.1 *Event Occurrences and Execution Traces*

Deciding what has to be considered an event strictly depends on the application domain. In a service-oriented setting, events may represent messages exchange, whereas in BPM an event may represent the fact that a certain activity has been started, completed, or reassigned. Furthermore, even if the application domain is fixed, there could be several different notions of events, because of the assumed perspective, the granularity, and so on. For example, one could adopt a coarse-grained approach by considering that each activity in a Business Process is atomic, thus linking the concept of event to the generic one of "activity execution".

Representation of events

Events are represented in CLIMB as LP terms. In this way, the language completely abstracts away from the problem of deciding "what an event is", and rather lets the developers decide what are the important events for modeling the domain, at the desired abstraction level. For example, the delivery of a message sent by a web service ws_1 to another web service ws_2, with a certain content can be simply denoted by $\mathtt{send}(ws_1, ws_2, \mathtt{content}(\ldots))$.

Happened events

The evolution of an instance of the system is characterized by events which *occur* when entities act and interact. Therefore, the language must provide a symbol denoting the occurrence of an event at a certain time must be provided by the language. Differently from other approaches (such as for example temporal logics), for which time is implicitly and qualitatively recalled by means of dedicated operators, CLIMB adopts an explicit and quantitative notion of time: happened events are associated with a corresponding numerical timestamp. More specifically, the occurrence (happening) of an event Ev at a (discrete) time point T is denoted by the atom

$$\mathbf{H}(Ev, T)$$

where Ev is a term and T is a numerical variable, which can assume real or integer values, depending on the chosen underlying time structure.

DEFINITION 4.1 (Time structure). A *time structure* $\mathfrak{T}$ is a totally ordered set $(\mathcal{S}, \prec)$. If $\mathfrak{T} \equiv (\mathbb{N}, <)$ the structure is *discrete-time*; if $\mathfrak{T} \equiv (\mathbb{R}, <)$, the structure is *dense-time*.

The modeler exploits happened events to refer to (classes of) concrete events generated during an execution. Happened events can be completely or partially specified.

CLIMB execution traces

A completely specified happened event is ground, and is therefore used to identify a specific occurrence of the involved event, at a given time. In particular, completely specified happened events represent the occurrences that dynamically fill the execution trace of an instance of the system; in the context of CLIMB, execution traces are then simply defined as sets of ground happened events.

DEFINITION 4.2 (Traces Herbrand base). The *traces Herbrand base* $\mathfrak{B}^{\mathbf{H}}$ is the Herbrand base built upon SCIFF happened events:

$$\mathfrak{B}^{\mathbf{H}} = \{\mathbf{H}(e, t) | e \in \mathfrak{U}, t \in \mathfrak{T}\}$$

where $\mathfrak{U}$ is the Herbrand universe of events.

DEFINITION 4.3 (CLIMB Execution trace). A CLIMB *execution trace* $\mathcal{T}$ is a set of ground happened events, i.e. a subset of the traces Herbrand base: $\mathcal{T} \subseteq \mathfrak{B}^{\mathbf{H}}$.

Partially specified happened events, instead, contain non-ground variables, and are used to identify and match with "classes" of concrete ground events. If variables contained in a happened event are completely free, then it will denote *any* occurrence of *any* event in the execution trace (i.e. it will match with all events in the trace); variables can then be subject to *constraints* in order to restrict the corresponding "matching" class. In particular, variables can be constrained by means of CLP constraints (see below) or Prolog predicates.

EXAMPLE 4.1 (Happened events). *Let us consider some examples of happened events in the context of a loan process. We assume that the handling of a loan is a non-atomic activity; the fact that the employee* Emp *starts (completes) to handle the loan identified by* Loan_{id} *can be represented by the term* $\mathsf{event}(\mathsf{Type}, \mathsf{handle_loan}, \mathsf{Emp}, \mathsf{Loan}_{id})$*, where* $\mathsf{Type}/\mathsf{start}$ *(*$\mathsf{Type}/\mathsf{complete}$*).*

$$\mathbf{H}(\mathsf{event}(\mathsf{start}, \mathsf{handle_loan}, \mathsf{alice}, \texttt{l-09-2008}), 14164) \qquad (\dagger)$$

represents that Alice started to handle the loan identified by `l-09-2008` *on October 13th 2008 (supposing that times are represented as the number of days from January 1st 1970).*

$\mathbf{H}(\mathsf{Ev}, \mathsf{T})$ *represents that something happens, and matches with any concrete occurring event. It obviously matches with the concrete occurrence* (†)*, with the following substitution:* $\mathsf{Ev}/\mathsf{event}(\mathsf{start}, \mathsf{handle_loan}, \mathsf{alice}, \mathtt{l\text{-}09\text{-}2008})$, $\mathsf{T}/14164$.

$\mathbf{H}(\mathsf{event}(\mathsf{Type}, \mathsf{handle_loan}, \mathsf{O}, \mathsf{Id}), \mathsf{T}) \wedge \mathsf{member}(\mathsf{Type}, [\mathsf{start}, \mathsf{com-plete}])$ *represents the happening of the start or the completion of the loan handling activity. It matches with the concrete occurrence* (†) *with the substitution* $\mathsf{Type}/\mathsf{start}$, $\mathsf{O}/\mathsf{alice}$, $\mathsf{Id}/\mathtt{l\text{-}09\text{-}2008}$ *and* $\mathsf{T}/14164$.

$\mathbf{H}(\mathsf{event}(\mathsf{start}, \mathsf{handle_loan}, \mathsf{O}, \mathsf{Id}), \mathsf{T}) \wedge \mathsf{T} \leqslant 13966$ *represents that the handling of a loan has been started by someone before the end of February 2008 (supposing that times are represented as the number of days from the 1st January 1970). It cannot match with the happened event* (†) *because* $14164 > 13966$.

$\mathbf{H}(\mathsf{event}(\mathsf{start}, \mathsf{handle_loan}, \mathsf{O}, \mathsf{Id}), \mathsf{T}) \wedge \mathsf{level}(\mathsf{O}, 1)$ *represents that a level 1 employee has started to handle a loan, supposing that* $\mathsf{level}(\mathsf{O}, \mathsf{L})$ *is a Prolog predicate denoting that the employee* O *belongs to the level* L. *It matches with the occurrence* (†) *only if Alice is a level-1 employee.*

$\mathbf{H}(\mathsf{event}(\mathsf{complete}, \mathsf{handle_loan}, \mathsf{alice}, \mathsf{Id}), \mathsf{T}) \wedge \mathsf{price}(\mathsf{Id}, \mathsf{P}) \wedge \mathsf{P} > 10^6$ *represents that Alice has completed to handle a loan involving an amount of 1K €, supposing that* $\mathsf{price}(\mathsf{Id}, \mathsf{P})$ *is a Prolog predicate denoting that the loan identified by* Id *involves the price* P. *It matches with the happened event* (†) *only if the loan* $\mathtt{l\text{-}09\text{-}2008}$ *involves more than 1K €.*

4.2.2 *Constraint Logic Programming*

Constraint Logic Programming (CLP) [101] is a class of programming languages that combines the advantages of LP with the efficiency of constraint solving. This is done by providing an interpretation to some of the symbols used in the program. In LP symbols are not interpreted, and therefore the term $2 + 3$ does not mean 5, but simply a structure whose functor is $+$ and whose terms are 2 and 3 ($2 + 3 \triangleq +(2, 3)$). Unification is a syntactic operation, so the term 5 will not unify with the term $3 + 2$, and the goal $5 = 3 + 2$ simply fails.

CLP is defined as a general scheme which can be specialized over particular domains. In particular, each language of the CLP class is identified by:

A. a *domain*, representing the set of values that a variable can assume;

B. the set of constraints;

C. the set of interpreted symbols.

For example, CLP($\mathcal{R}$) [102] is the instance of CLP that works on the reals; this means that a variable in CLP($\mathcal{R}$) can have a real value, and it can be subject to constraints on the reals. Current implementations

typically employ the simplex algorithm as constraint solver and feature a complete propagation of linear constraints. CLP($\mathcal{FD}$) instead is the specialisation of CLP on the Finite Domains [70].

These two CLP languages give an interpretation to numbers, their relation symbols $<$, $\leqslant$, $\neq$, ... and operations $+$, $-$, $*$, $/$. Therefore, in CLP the symbol 5 stands for the number *five* and the symbol + represent the *addition* operation. For these symbols, unification is extended with a more powerful constraint solving mechanism. For example, the goal $5 = A + 3$ succeeds, providing the answer $A = 2$. This behaviour is obtained by performing a syntactic identification of the interpreted atoms (called *constraints*), inserting them in a *constraint store* and delegating their evaluation to a *constraint solver* instead of applying classical resolution. CLP constraints have the effect of pruning the domain of the variables; the constraint solver propagates this information to other constraints, updating the domains of the involved variables and iterating the propagation. For example, if we consider times to be real variables, adopting a dense-time structure, the constraint $T \leqslant 13966$, cited in Example 4.1, restricts the domain of T to $(-\infty, 13966]$.

Various languages and efficient solvers have been developed to describe and handle constraints, such as ECL^{i}PS^e[2] and SICStus[3]. Such languages have been successfully exploited for hard combinatorial problems in many different application domains, such as scheduling [31], planning [164], and bioinformatics [152].

4.2.3 *Expectations*

The perhaps most important feature of CLIMB is that it focuses on the *dynamics* of the system by adopting a *dynamic* approach. In particular, beside the representation of "what" happened and "when", the language explicitly supports the modeling of "what" is expected (not) to happen, and "when". The notion of *expectation* is then used to represent, at every point in time, the (un)desired evolution of the execution, and plays a key role when defining the dynamic evolution of an Event-Based System: it is quite natural, in fact, to think of EBSs in terms of rules of the form *"if* e_1 *happened, then* e_2 *is expected (not) to happen"*.

Since CLIMB adopts an open approach, the prohibition of a certain event should be explicitly expressed in the model; for that reason, the language supports the concepts of positive and negative expectations.

Positive expectations

The fact that the event Ev is expected to happen at time T is modeled with a *positive expectation*, whose form is

$$\mathbf{E}(Ev, T)$$

Negative expectations

Conversely, the fact that the event Ev is expected not to happen (i.e., is forbidden) at time T is modeled with a *negative expectation*, whose form is

$$\mathbf{EN}(Ev, T)$$

2 http://www.eclipse-clp.org/
3 http://www.sics.se/isl/sicstuswww/site/

On the one hand, expectations typically refer to events which have not yet occurred, and is therefore impossible to have complete knowledge about them. Variables enable the modeler to deal with this source of incompleteness by modeling expectations on partially specified events and non-ground time variables. On the other hand, the modeler needs to put requirements on these expectations. Technically, this is again done by means of CLP constraints and Prolog predicates.

Roughly speaking, the quantification of variables involved in positive and negative expectations follows the intuitive meaning. Positive expectations are existentially quantified: they predicate on the presence of (at least) one corresponding occurrence; on the contrary, variables in negative expectations are universally quantified (unless they are also contained in a positive expectation): they forbid the presence of all the possible corresponding occurrences. The following example clarifies this aspect considering some concrete cases.

EXAMPLE 4.2 (Requirements on expectations). *A business manager in a loan company needs to specify that:*

A. *"someone must start to handle the loan `l-01-2008` by the end of February 2008".*

B. *"level-1 employees cannot handle loans involving an amount greater than 1K €";*

Being workers autonomous, these requirements cannot be enforced, but rather they are formulated as expectations that the business manager places on employees. In particular, the first requirement is a positive expectation, whereas the second requirement is a negative expectation (they respectively describe a desired and an undesired situation).

Following the approach of Example 4.1, we assume that an event concerning loan handling is represented by $\mathsf{event}(\mathsf{T}, \mathsf{handle_loan}, \mathsf{O}, \mathsf{Id})$*, where* T *is the event type (start or complete),* O *is the worker who handles the loan and* Id *is the loan identifier. Furthermore, we use a Prolog predicate* $\mathsf{price}(\mathsf{Id}, \mathsf{P})$ *to specify that the loan identified by* Id *involves an amount* P *and the predicate* $\mathsf{level}(\mathsf{O}, \mathsf{L})$ *to state that the employee* O *belongs to the level* L*.*

Under this assumptions, we can be express the first statement with the following positive expectation:

$$\mathbf{E}(\mathsf{event}(\mathsf{start}, \mathsf{handle_loan}, \mathsf{O}, \texttt{l-01-2008}), \mathsf{T}) \wedge \mathsf{T} \leqslant 13966$$

The expectation states that there must exist a worker O *and a time* $\mathsf{T} \leqslant 13966$ *at which* O *starts to handle the loan* `l-01-2008` *(*O *and* T *are existentially quantified).*

We can express instead the second statement with the following negative expectation:

$$\mathbf{EN}(\mathsf{event}(_, \mathsf{handle_loan}, \mathsf{O}, \mathsf{Id}), \mathsf{T}) \wedge \mathsf{level}(\mathsf{O}, 1) \\ \wedge \mathsf{price}(\mathsf{Id}, \mathsf{P}) \wedge \mathsf{P} > 10^6$$

The expectation states that it should never happen that a level-1 employee O *is responsible of an event related to loans which involve more than 1K €*

$\mathcal{IC}$	::=	[IC]*
IC	::=	Body " → " Head "."
Body	::=	"true" | BConjunction
BConjunction	::=	BConjunct [" ∧ " BConjunct]*
Head	::=	"⊥" | HDisjunction
HDisjunction	::=	HDisjunct [" ∨ " HDisjunct]*
HDisjunct	::=	HConjunct [" ∧ " HConjunct]*
HEvent	::=	"**H**(" Event "," Time ")"
Expectation	::=	ExpType "(" Event "," Time ")"
ExpType	::=	"**E**" | "**EN**"
Event	::=	Term
Time	::=	Number | Variable
Literal	::=	Atom | "not"Atom

Table 13: Common syntax of SCIFF integrity constraints. The definition of the BConjunct and HConjunct non-terminal symbols depends on the chosen variant (CLIMB, SCIFF-lite, full SCIFF).

*(*Type *and* T *are universally quantified on the entire domain,* O *is universally quantified over the domain of level-1 employees,* Id *is universally quantified on the domain of identifiers representing* $> 1K€$ *loans).*

4.2.4 *Integrity Constraints*

CLIMB *integrity constraints* are rules used to relate happened events and expectations[4]. They allow the user to specify what is the desired (expected) behavior of the system's executions, provided that a certain state of affairs, described by means of happened events, occurs.

Integrity constraints are represented as forward rules of the form Body → Head. Their syntax has a part which is common to the three variants of the SCIFF language (CLIMB, SCIFF-lite and full SCIFF, see Figure 14 – Page 54). The grammar of this part is reported in Table 13, where Atom, Term, Variable and CLPConstraint have the same structure as in LP and CLP, and not denote Negation As Failure (NAF) [56]. Roughly speaking, it states that the body of each integrity constraint is a conjunction of BConjunct symbols, whereas the head is a disjunction of conjunction of HConjunct symbols. These two non-terminal symbols are then defined in a different way by the three variants.

In particular, CLIMB integrity constraints are Body → Head rules where:

4 They are called *social integrity constraints* in the SCIFF setting, because they are used to regulate interaction from a global viewpoint, i.e. at the social level of the MAS.

BConjunct	::=	HEvent | Literal | CLPConstraint
HConjunct	::=	Expectation | Literal | CLPConstraint

Table 14: CLIMB specialization of the integrity constraints syntax reported in Table 13.

- Body contains (a conjunction of) happened events, together with constraints on their variables;
- Head contains (a disjunction of conjunctions of) positive and negative expectations, together with constraints on their variables (and/or on variables contained in the Body).

The corresponding grammar is reported in Table 14.

Each integrity constraint can be perceived as a business rule or policy which captures and constrains a specific behavioural aspect of the system. The set of all integrity constraints (which will be usually denoted by $\mathcal{IC}$) represents the regulatory model of the entire system. Such a regulatory model may contain internal policies as well as external regulations.

EXAMPLE 4.3 (Integrity constraints). *Let us consider the following business rules:*

(*Br1*) *"The person who takes the final decision about the acceptance or rejection of a paper cannot review that paper."*

(*Br2*) *"If a customer has closed an order and the shop has consequently accepted it, then the customer is bound to execute the corresponding payment."*

(*Br3*) *"If a premium customer pays for an order by credit card, then the seller should answer within 10 time units by delivering a corresponding receipt, or by communicating a payment failure."*

We adopt an atomic model for the activities, i.e. map their execution to single events.

Sentence Br1 is an example of the four eyes principle, *which states that two different activities cannot be executed by the same person. It can be represented by means of the following CLIMB integrity constraint:*

$$\begin{aligned}&\mathbf{H}(\mathsf{emit_decision}(\mathsf{Person}, \mathsf{Paper}, \mathsf{Decision}), \mathsf{T}_e)\\ \rightarrow&\mathbf{EN}(\mathsf{review}(\mathsf{Person}, \mathsf{Paper}), \mathsf{T}_r).\end{aligned} \tag{Br1-CLIMB}$$

Sentence Br2 describes a rules which is activated only when two different events (order closing and order acceptance) occur; in this case, the payment is expected:

$$\begin{aligned}&\mathbf{H}(\mathsf{close_order}(\mathsf{Customer}, \mathsf{Shop}, \mathsf{Order}), \mathsf{T}_c)\\ &\wedge \mathbf{H}(\mathsf{accept_order}(\mathsf{Shop}, \mathsf{Customer}, \mathsf{Order}), \mathsf{T}_a)\\ &\wedge \mathsf{T}_a > \mathsf{T}_c\\ \rightarrow&\mathbf{E}(\mathsf{pay}(\mathsf{Customer}, \mathsf{Shop}, \mathsf{Order}, \mathsf{Method}), \mathsf{T}_p)\\ &\wedge \mathsf{T}_p > \mathsf{T}_a.\end{aligned} \tag{Br2-CLIMB}$$

The CLP *constraint* $T_a > T_c$ *is used to identify a "legal" ordering of happened events: the shop can accept an order only if the customer has previously closed it (if this is not the case, then something was wrong, and the expectation about the payment is not placed). The second constraint (*$T_p > T_a$*) is used to identify the time range inside which the customer is bound to pay (i.e., after both the events in the body already occurred).*

Sentence Br3 is an example of disjunctive rule: it specifies that when a certain event happen, s.t. a corresponding requirement is satisfied (the customer is a premium one), one among two possible behaviors is expected. This can be captured in CLIMB *by using a disjunctive head:*

$$\begin{aligned}
&\mathbf{H}(\mathsf{pay}(\mathsf{Customer}, \mathsf{Shop}, \mathsf{Item}, \mathsf{credit_card}), T_p) \\
&\quad \wedge \mathsf{premium_customer}(\mathsf{Customer}, \mathsf{Shop}) \\
&\rightarrow \mathbf{E}(\mathsf{deliver}(\mathsf{Shop}, \mathsf{Customer}, \mathsf{receipt}(\mathsf{Item}, \mathsf{Info})), T_d) \\
&\quad \wedge T_d > T_p \wedge T_d < T_p + 10 \\
&\vee \mathbf{E}(\mathsf{tell}(\mathsf{Seller}, \mathsf{Customer}, \mathsf{failure}, \mathsf{Reason}), T_f) \\
&\quad \wedge T_f > T_p \wedge T_f < T_p + 10.
\end{aligned} \tag{Br3-CLIMB}$$

The concept of premium customer is formalized in the integrity constraint by means of the premium_customer/2 *Prolog predicate, and supposing that*

$$\mathsf{premium_customer}(C, S)$$

is true if C *is a premium customer for the shop* S*. To express mutual exclusion between the receipt delivery and the failure communication, we could also add a rule like*

$$\begin{aligned}
&\mathbf{H}(\mathsf{deliver}(\mathsf{Seller}, \mathsf{Customer}, \mathsf{receipt}(\mathsf{Item}, \mathsf{Info})), T_d) \\
&\rightarrow \mathbf{EN}(\mathsf{tell}(\mathsf{Seller}, \mathsf{Customer}, \mathsf{failure}, \mathsf{Reason}), T_f).
\end{aligned}$$

and viceversa.

Deadlines

An interesting aspect of Sentence Br3 in Example 4.3 is that it contains an example of *deadline*, i.e. of a maximum time delay by which an event (the delivery of a receipt or the communication of a failure) is expected to occur. Deadlines and other quantitative time constraints are treated in CLIMB by imposing CLP constraints on time variables. For example, the fact that a receipt must be delivered by 10 time units w.r.t. the payment is rephrased by imposing that the delivery time is constrained to be lower than the payment time plus 10 time units.

Triggering of integrity constraints

Integrity constraints are operationally interpreted in a forward, *reactive* manner. Occurring events match with the happened events contained in integrity constraints, grounding their variables to actual values. When the entire body of an integrity constraint becomes true, then the integrity constraint *triggers*, and the head must also be true. As we will see, this leads to the generation of the expectations contained in the head; if the head is a disjunction, then a choice point is opened. Variables contained in the body of integrity constraints are universally quantified with scope the entire integrity constraint: in this way, each integrity constraint triggers for any possible actual configuration that makes its body true.

EXAMPLE 4.4 (Triggering of integrity constraints). *Let us consider the integrity constraint* (Br2-CLIMB) *and the following integrity constraint, which formalizes the fact that a customer has the possibility to close at most one order at the same shop:*

$$\begin{aligned}&\mathbf{H}(\mathsf{close_order}(\mathsf{Customer}, \mathsf{Shop}, \mathsf{Order}_A), T_c)\\ \rightarrow &\mathbf{EN}(\mathsf{close_order}(\mathsf{Customer}, \mathsf{Shop}, \mathsf{Order}_B), T_{c2})\\ &\wedge T_{c2} \neq T_c.\end{aligned} \quad \text{(Br4-CLIMB)}$$

Let us now consider the following execution trace:

(h_1) $\mathbf{H}(\mathsf{close_order}(\mathsf{alice}, \mathsf{bookShop}, \mathsf{order}-1), 2)$.

(h_2) $\mathbf{H}(\mathsf{accept_order}(\mathsf{bookShop}, \mathsf{alice}, \mathsf{order}-1), 3)$.

(h_3) $\mathbf{H}(\mathsf{accept_order}(\mathsf{bookShop}, \mathsf{lewis}, \mathsf{order}-2), 5)$.

(h_4) $\mathbf{H}(\mathsf{close_order}(\mathsf{lewis}, \mathsf{bookShp}, \mathsf{order}-2), 8)$.

(h_5) $\mathbf{H}(\mathsf{close_order}(\mathsf{lewis}, \mathsf{bookShop}, \mathsf{order}-2), 13)$.

(h_6) $\mathbf{H}(\mathsf{accept_order}(\mathsf{bookShop}, \mathsf{lewis}, \mathsf{order}-2), 21)$.

Being the body of the integrity constraints universally quantified with scope the entire rule, every time that a group of events making the body true happens, then the rule triggers, generating the involved expectations. In this case:

- *Happened Events* h_1 *and* h_2 *unify with the two happened events contained in the body of rule* (Br2-CLIMB) *with substitution* Customer/alice, Shop/bookShop, Order/order − 1, T_c/2, T_a/3. *Since* $3 > 2$, *the constraint contained in the body is true, and the following expectation is generated:*

$$\mathbf{E}(\mathsf{pay}(\mathsf{alice}, \mathsf{bookShop}, \mathsf{order}-1, \mathsf{Method}), T_1) \wedge T_1 > 3$$

- *Happened Event* h_4 *unifies with the first happened event contained in the body of* (Br2-CLIMB), *involving the substitution* Shop/bookShp; *this substitution also propagates to the second happened event (the two events share the same variable), but the execution trace does not contain any acceptance order executed by a* Shop *called* bookShp. *Therefore, the rule does not trigger.*

- *Happened Events* h_3 *and* h_5 *unify with the two happened events contained in the body of rule* (Br2-CLIMB), *but the rule does not trigger, because the substitutions* $T_c/5$, $T_a/3$ *do not satisfy the constraint* $T_a > T_c$ *(the ordering of happened events is not the one required by the integrity constraint).*

- *Happened Events* h_5 *and* h_6 *unify with the two happened events contained in the body of rule* (Br2-CLIMB), *also satisfying the ordering constraint. The following expectation is generated:*

$$\mathbf{E}(\mathsf{pay}(\mathsf{lewis}, \mathsf{bookShop}, \mathsf{order}-2, \mathsf{Method}), T_2) \wedge T_2 > 21$$

$\mathcal{KB}$	::=	[Clause]*
Clause	::=	CHead [" ← " CBody] "."
CHead	::=	Atom

Table 15: Common syntax of a SCIFF knowledge base. The definition of the CBody non-terminal symbol depends on the chosen variant (CLIMB, SCIFF-lite, full SCIFF).

- *each one of Happened Events* h_1, h_4 *and* h_5 *makes the body of the integrity constraint* (Br4-CLIMB) *true, generating respectively the following negative expectation:*

$$\mathbf{EN}(close_order(alice, bookShop, _), T_3) \wedge T_3 \neq 2$$
$$\mathbf{EN}(close_order(alice, bookShp, _), T_4) \wedge T_4 \neq 8$$
$$\mathbf{EN}(close_order(alice, bookShop, _), T_5) \wedge T_5 \neq 13$$

4.2.5 *The Static Knowledge Base*

As far as now, the chapter has been focused on the dynamic of an EBS, which is captured in CLIMB by means of integrity constraints which relate occurring events with expectations. Happened events and expectations contain variables that can be subject to constraints. Among the possible employed constraints, there are Prolog predicates, which have been used for example to characterize properties of a certain entity (e.g., the fact that a customer is premium in rule (Br3-CLIMB) – Example 4.3), or to obtain related informations (e.g., the price of an order and the level of an employee in Example 4.2).

In order to characterize such a knowledge, i.e. to give a definition to Prolog predicates, CLIMB provides a Prolog knowledge base (which will be usually identified with $\mathcal{KB}$). Here the modeler can formalize all the "static" background knowledge about the system (i.e., the information independent from specific instances), completing the definition of integrity constraints. Examples of this kind of information are roles descriptions, list of participants, a database containing information about items in a shop, complex conditions and decisions involving data, ...

As for integrity constraints, CLIMB knowledge bases are a subclass of SCIFF knowledge bases. Each SCIFF knowledge base is a logic program, i.e. a set of clauses, as reported in the common syntax shown in Table 15, where Atom has the same form as in LP. The three variations of SCIFF provide different possibilities in the body of clauses, hence the symbols CBody is left unspecified in the common syntax.

In CLIMB, the body of clauses are literals or CLP constraints. The syntax is specified in Table 16, where CLPConstraint has the same form as in CLP.

Example 4.5 (A knowledge base expressing roles in a company). *Let us consider the characterization of the concept of* level *associated to each employee inside a company. The company states that:*

CBody	::=	Literal | CLPConstraint
Literal	::=	Atom | "not"Atom

Table 16: CLIMB specialization of the knowledge base syntax reported in Table 15.

- *each employee can play different roles, but has a unique "primary" role, used to determine her level;*
- *roles are organized in a hierarchy;*
- *the level of a role is the maximum depth of the role inside the hierarchy, starting from the leaves.*

This kind of knowledge is static: it does not depend on the dynamic of the system, but on its structural organization. Therefore, it can be suitably modeled in a knowledge base. This knowledge base will involve two parts:

A. *the formalization of the criterion used to evaluate the level of an employee (intensional part);*

B. *the database of all employees together with their corresponding roles and primary roles (extensional part).*

We assume that role(E, R) *and* primary_role(E, PR) *respectively identify the role(s)* R *and the primary role* PR *played by employee* E*. Furthermore, we model the hierarchy of roles with predicate* children(R, Children)*, which identifies the list of roles that are children of a role. Leaf roles do not have children. Under this assumptions, the intensional knowledge base can be realized as follows (*level(E, L) *states that the level of* E *is* L*):*

$$
\begin{array}{rcl}
\mathrm{level}(E, L) & \leftarrow & \mathrm{primary_role}(E, PR) \wedge \mathrm{r_level}(PR, L). \\
\mathrm{r_level}(R, L) & \leftarrow & \mathrm{get_children}(R, C) \wedge \mathrm{r_levels}(C, \mathrm{Levels}) \\
 & & \wedge \max(L_{max}, \mathrm{Levels}) \wedge L \text{ is } L_{max} + 1. \\
\mathrm{get_children}(R, C) & \leftarrow & \mathrm{children}(R, C). \\
\mathrm{get_children}(R, []) & \leftarrow & \mathrm{not}(\mathrm{children}(R, C)). \\
\mathrm{r_levels}([R|Rs], [L|Ls]) & \leftarrow & \mathrm{r_level}(R, L) \wedge \mathrm{r_levels}(Rs, Ls). \\
\mathrm{r_levels}([], [0]). & &
\end{array}
$$

where max(M, L) *is true if* M *is the maximum element of the list* L*.*

A possible extensional knowledge base could be

```
children(supervisor, [technical_leader, secretary]).
children(technical_leader, [programmer, analyst]).
primary_role(alice, programmer).
primary_role(hatter, technical_leader).
primary_role(lewis, supervisor).
role(lewis, analyst).
```

According to the whole knowledge base, the level of Lewis is 3.

BConjunct	::=	HEvent | Literal | CLPConstraint
HConjunct	::=	HEvent | Expectation | Literal | CLPConstraint
CBody	::=	HEvent | Literal

Table 17: SCIFF-lite specialization of the syntax reported in Tables 13 and 15 respectively.

4.2.6 *SCIFF-lite and Composite Events*

As depicted in Figure 14 (page 54), CLIMB is a subset of more expressive languages, SCIFF-lite and SCIFF in order of expressiveness. Here we introduce how the syntax of CLIMB integrity constraints and knowledge base is extended in the case of SCIFF-lite. The syntax of full SCIFF is outside of the scope of this dissertation; the interested reader may refer to [7, 44].

SCIFF-lite syntax

From the syntax point of view, SCIFF-lite provides two extensions to the CLIMB syntax: the head of integrity constraints can contain happened events, which can also be used to define a Prolog predicate (i.e., inside the body of clauses). The extended syntax is reported in Table 17.

Composite events

Happened events contained in the head of integrity constraints represent fictitious events that are generated by rules themselves; this open many possibilities, like for example dealing with *composite events*, as they are called in the Complex Event Processing (CEP) field [124]. Composite events are complex events which do not concretely occur during the execution (i.e., they do not appear inside the execution trace), but that are determined as a combination of concrete events (under certain circumstances). They can ease the definition of rules, providing the possibility of modeling integrity constraints by mixing different levels of abstractions.

EXAMPLE 4.6 (Composite Events). *Let us consider the following requirements:*

- *if the device encounters a connection problem, then an alert must be communicated to the user immediately (i.e., at the following moment);*
- *the device encounters a connection problem if*
  - *the user enters wrong credentials*
  - or *the cable status is "off" for two consecutive detections*

Let us now suppose that the only concrete events handled by the device are that the detection of credentials acceptance/rejection and the detection of the cable status. Under this assumption, the "problem detected" event is not directly provided by the device, but rather must be inferred from the occurrence of other events. Nevertheless, the first requirement can be mapped to an integrity constraint which includes the detection of a problem as a normal event:

$$\mathbf{H}(\mathsf{problem_detected}, T_p) \rightarrow \mathbf{E}(\mathsf{generate_alert}, T_a) \wedge T_a = T_p + 1.$$

The second requirement is instead used to characterize the (combination of) events which lead to the "abstract" occurrence of problem detection. Such a requirement can be captured in SCIFF*-lite by using rules which contain the occurrence of problem detection in the head. The first rule links the fact that the user enters wrong credentials with the detection of a problem:*

$$\mathbf{H}(\mathsf{wrong_credentials}, T) \rightarrow \mathbf{H}(\mathsf{problem_detected}, T).$$

The second rules states that a problem is encountered if two consecutive "off" cable status are detected:

$$\begin{array}{lll} & \mathbf{H}(\mathsf{status}(\mathsf{cable}, \mathsf{off}), T_1) & \\ \wedge & \mathbf{H}(\mathsf{status}(\mathsf{cable}, \mathsf{off}), T_2) & \\ \wedge & T_2 > T_1 & \rightarrow \mathbf{E}(\mathsf{status}(\mathsf{cable}, S), T_3) \\ & & \quad \wedge T_3 > T_1 \wedge T_3 < T_2 \\ & & \vee \mathbf{EN}(\mathsf{status}(\mathsf{cable}, S), T_3) \\ & & \quad \wedge T_3 > T_1 \wedge T_3 < T_2 \\ & & \quad \wedge \mathbf{H}(\mathsf{problem_detected}, T_2). \end{array}$$

The body of the integrity constraints contain the occurrence of two different "off" status detections. The head is instead modeled by considering the two possible alternatives that may arise:

- *The two "off" detections are not* consecutive*; this situation happens if another detection interposes between them.*
- *The two "off" detections are consecutive, and thus a problem is encountered; this situation happens if no detection interposes between them.*

4.3 CLIMB DECLARATIVE SEMANTICS

The declarative semantics of CLIMB resembles the one of SCIFF [7, 44]. CLIMB specifications are interpreted in terms of abductive logic programs, giving an abductive characterization to positive and negative expectations. The declarative semantics goes further: the gap between expectations and happened events is bridged through the definition of *fulfillment*. The basic idea is that when an integrity constraint triggers, the expectations contained in its head are hypothesized; an *hypotheses-confirmation* step is then performed to check if the actual behavior of the system (a specific execution trace) effectively adheres to the generated expectations. This, in turn, gives a formal account to the notion of *compliance* of an execution trace w.r.t. a CLIMB specification, which is a central aspect of the dissertation.

4.3.1 *Abduction*

The notion of abduction has been first introduced by Peirce [98], who identified three forms of reasoning:

DEDUCTION, an analytic process where a general rule is applied to a particular case, inferring a result;

INDUCTION, a synthetic process which infers the rule from the particular case and the result;

Peirce's definition of abduction

ABDUCTION, a synthetic process which infers a case (a possible explanation/cause) from the particular result and the rule.

Abductive reasoning focuses on the "probational adoption of a hypothesis" as a possible explanation for observed facts (results), according to known laws [106]. It is a form of *incorrect* reasoning: *"we cannot say that we believe in the truth of the explanation, by only that it may be true"* (Peirce, [98]).

A well-known introductory example of abductive reasoning [154, 106] is illustrated.

EXAMPLE 4.7 (Abductive reasoning). *Let us consider the following set of sentences (theory), known by the agent:*

- *If the grass is wet, then the shoes are wet as well.*
- *If the last night it rained, then the grass is wet.*
- *If the sprinkler was on, then the grass is wet.*

The agent observes that her shoes are wet and she wants to know why. To find a (possible) answer, she performs abductive reasoning: she tries to find a possible explanation for the observation, namely a set of hypotheses which, together with the known sentences, implies the given observation. In this case, different hypotheses *may be placed: for example, it could be the case that the sprinkler was on, that it rained, or both.*

Abduction and incomplete knowledge

Basic explanations

Abducibles

Example 4.7 clearly illustrates that abductive reasoning is suitable to deal with situation where knowledge is *incomplete*: the agent observes the world (i.e., the effects of unknown causes) and exploits its own knowledge to hypothesize possible explanations for what is being observed. Obviously, not all explanations are satisfactory for the agent; for example, it would be desirable to explain an effect in terms of a cause and not in terms of another effect, i.e., to take into account only *basic* explanations [106]. In Example 4.7, "sprinkler was on" is a basic explanation, whereas "the grass is wet" is not. More generally, it is important to distinguish between sentences that are known to the agent (sentences whose truth value can be established by the agent) and sentences whose validity is unknown by the agent; the latter are hypothesized when trying to find (basic) explanations for what is being observed. For this reason, sentences used in the explanations are restricted to belong to a special domain-dependent class of sentences called *abducibles*.

Integrity constraints

When performing abductive reasoning, an agent does not only make use of the observation and the known laws, but it also exploits rules which constrain the way possible acceptable explanations are formulated. These rules are employed to characterize what are the legal states of knowledge, rejecting unintended abductive explanations. They

formalize the *integrity* of abductive explanations, and are therefore called *integrity constraints*[5].

EXAMPLE 4.8 (Abductive reasoning with integrity constraints). *Let us extend Example 4.7 with the following integrity constraint:*

- *It is impossible that it rained last night if the moon was shining.*

This integrity constraint is a typical example of denial, *because it explicitly reject an unlegal state of knowledge. In particular, it forces the rejection of "it rained" as a possible explanation if the agent knows that the moon was shining. If it is the case, the only basic abductive explanation for having wet shoes is that the sprinkler was on.*

Example of a denial

4.3.2 *Abductive Logic Programming*

Abuctive Logic Programming (ALP) is the extension of LP to support abduction [106].

DEFINITION 4.4 (Abductive logic program). An *abductive logic program* is a triple $\langle \mathcal{T}, \mathcal{A}, \mathcal{IC} \rangle$ where:

- $\mathcal{T}$ is a logic program (representing the theory of the agent);
- $\mathcal{A}$ is the set of abducible predicates[6];
- $\mathcal{IC}$ is a set of Integrity Constraints.

In this setting, abductive explanations can be formalized as follows.

DEFINITION 4.5 (Abductive Explanation). Given an abductive logic program $\langle \mathcal{T}, \mathcal{A}, \mathcal{IC} \rangle$ and a formula γ (goal), the purpose of abduction is to find a set of ground atoms $\Delta \subseteq \mathcal{A}$ which, together with $\mathcal{T}$, entails[7] γ and satisfies $\mathcal{IC}$:

$$\mathcal{T} \cup \Delta \models \gamma$$
$$\mathcal{T} \cup \Delta \models \mathcal{IC}$$

In this case, Δ is an *abductive explanation* for G.

EXAMPLE 4.9 (An abductive logic program). *Let us formalize the concepts introduced in Example 4.8 as an abductive logic program* $\langle \mathcal{T}, \mathcal{A}, \mathcal{IC} \rangle$. *We consider the facts that sprinkler was on and that it rained as possible*

5 The term is taken from the field of databases, in which integrity constraints are used to ensure accuracy and consistency of data.
6 We will use the same symbol to indicate the set of abducible predicates and the set of all their ground instances. To distinguish abducible predicates, we will depict them in **bold**.
7 The notion of entailment depends on the declarative semantics associated with the theory.

(partial) explanations, hence $\mathcal{A} = \{\textbf{sprinkler_was_on}, \textbf{rained_last_night}\}$*.*
The agent's theory $\mathcal{T}$ *is*

$$\begin{aligned} \mathtt{shoes_are_wet} &\leftarrow \mathtt{grass_is_wet}. \\ \mathtt{grass_is_wet} &\leftarrow \textbf{rained_last_night}. \\ \mathtt{grass_is_wet} &\leftarrow \textbf{sprinkler_was_on}. \end{aligned}$$

whereas $\mathcal{IC}$ *contains*

$$\textbf{rained_last_night} \wedge \mathtt{moon_was_shining} \rightarrow \bot.$$

If the agent observes that her shoes are wet (i.e., $\gamma = \mathtt{shoes_are_wet}$*), then three possible explanations can be hypothesized:*

$$\begin{aligned} \Delta_1 &= \{\textbf{rained_last_night}\} \\ \Delta_2 &= \{\textbf{sprinkler_was_on}\} \\ \Delta_3 &= \{\textbf{rained_last_night, sprinkler_was_on}\} \end{aligned}$$

Let us now suppose that someone informs the agent that the moon was shining last night. $\mathcal{T}$ *is extended with* $\mathtt{moon_was_shining}$*, and* Δ_1 *becomes the only acceptable abductive explanation (with* Δ_2 *and* Δ_3 *the integrity constraint is not entailed anymore).*

Remark 4.1 (Goal reformulation). In the following, we will not consider an explicit goal but only the set of integrity constraints. If we are not interested in the computed answer substitution, this choice do not lead to loose generality: Definition 4.5 can be simply reformulated as

$$\mathcal{T} \cup \Delta \models \mathcal{IC} \cup \{\mathtt{true} \rightarrow \gamma\}$$

4.3.3 *Representing a system and its executions*

We have described CLIMB specifications as composed by two main parts: a static part representing the background knowledge of the system, modeled by means of a knowledge base; a dynamic part constraining the behaviour of the system during the execution, modeled by means of integrity constraints. We can then map a CLIMB specification to an abductive logic program, where expectations are considered as abducibles, i.e., as pieces of knowledge that are hypothesized (made true) when rules trigger. The set of hypothesized expectations models the ideal behaviour that the system must exhibit when a certain situation (a partial execution trace) occurs.

Definition 4.6 (CLIMB specification). A *CLIMB specification* is an abductive logic program $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ where:

- the theory is the knowledge base of the system and obeys to the syntax shown in Table 16;
- expectations are abducibles,i.e., $\mathcal{A} = \{\mathbf{E}/2, \mathbf{EN}/2\}$);
- integrity constraints obey to the syntax shown in Table 14.

Since all CLIMB specifications have a fixed set of abducibles, $\mathcal{A}$ will be omitted.

The following definition and remark explicitly stress that, due to their syntax, CLIMB integrity constraints do not contain expectations in the body.

Definition 4.7 (Body and head of an integrity constraint). Given an integrity constraint IC, $\mathsf{body}(IC)$ and $\mathsf{head}(IC)$ respectively identify its *body* and *head*. Given an atom a, $a \in \mathsf{body}(IC)$ iff a is a conjunct of the body of IC and $a \in \mathsf{head}(IC)$ iff a is a conjunct of the head of IC.

Remark 4.2 (Structure of CLIMB integrity constraints). Given a CLIMB specification $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$, no integrity constraint belonging to $\mathcal{IC}$ contains expectations in its body:

$$\forall IC \in \mathcal{IC},\ \forall e \in \mathfrak{U},\ \forall t \in \mathfrak{T},\ \mathbf{E}(e,t) \notin \mathsf{body}(IC) \wedge \mathbf{EN}(e,t) \notin \mathsf{body}(IC)$$

Example 4.10 (A CLIMB specification modeling the FIPA query-ref interaction protocol). *Let us consider the CLIMB specification of a simplified version of the FIPA*[8] *query-ref interaction protocol*[9]*. As described in the FIPA specification*[10]*,* "query-ref is the act of asking another agent to inform the requestor of the object identified by a descriptor"*. In particular, the initiator agent requests the receiver to perform an* inform *act containing the object that corresponds to the descriptor; the receiver can return the requested information or refuse the request. The Agent UML*[11] *diagram of the protocol is shown in Figure 16. The query-ref protocol can be suitably represented with*

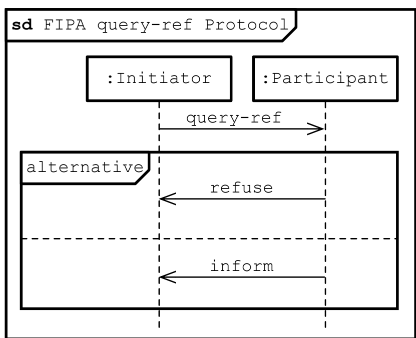


Figure 16: Agent UML model for a simplified version of the *query-ref* FIPA interaction protocol.

8 http://www.fipa.org
9 The full version supports two different speech acts to start the protocol, and it envisages the possibility that the receiving agent does not understand the message or experiences an exception.
10 Experimental specification XC00037H, available from http://www.fipa.org.
11 http://www.auml.org/

the CLIMB *specification* $\mathcal{Q} = \langle \mathcal{KB}, \{(QR_1), (QR_2), (QR_3)\} \rangle$ *where:*

- (QR_1) *specifies that when an initiator agent* I *sends a query-ref to another agent* P*, then* P *is expected to answer by uttering an inform or a refuse. We extend the specification by introducing a (parametric) deadline on the maximum time by which the answer is expected, and by accepting executions in which the answer is received before the request – the goal of the protocol is to obtain the information, not to obtain the information after the request*[12].

$$\begin{aligned} &\mathbf{H}(\mathtt{tell}(I, P, \mathtt{query-ref}(\mathtt{Info})), T_q) \wedge \mathtt{qr_deadline}(D) \\ &\rightarrow \mathbf{E}(\mathtt{tell}(P, I, \mathtt{inform}(\mathtt{Info}, \mathtt{Answer})), T_i) \wedge T_i < T_q + D \\ &\vee \mathbf{E}(\mathtt{tell}(P, I, \mathtt{refuse}(\mathtt{Info})), T_r) \wedge T_r < T_q + D. \end{aligned} \quad (QR_1)$$

- (QR_2) *imposes mutual exclusion between the two answers, stating that if an agent* P *has accepted a query-ref from an agent* I *about the information* Info*, then* P *cannot send a refuse message for the same information to the same agent* I[13].

$$\begin{aligned} &\mathbf{H}(\mathtt{tell}(P, I, \mathtt{inform}(\mathtt{Info}, \mathtt{Answer})), T_i) \\ &\rightarrow \mathbf{EN}(\mathtt{tell}(P, I, \mathtt{refuse}(\mathtt{Info})), T_r). \end{aligned} \quad (QR_2)$$

- (QR_3) *expects that the interaction protocol is started by some agent* I.

$$\mathtt{true} \rightarrow \mathbf{E}(\mathtt{tell}(P, I, \mathtt{query-ref}(\mathtt{Info})), T). \quad (QR_3)$$

- $\mathcal{KB}$ *specifies the value of the deadline, i.e., contains a fact like*

$$\mathtt{qr_deadline}(10). \quad (QR_{KB})$$

Instances

A specific execution of the system, called *instance*, is formally identified by the CLIMB specification that models the system and the execution trace produced by the instance.

Definition 4.8 (Instance). Given a CLIMB specification $\mathcal{S}$ and a trace $\mathcal{T}$, $\mathcal{S}_{\mathcal{T}} = \langle \mathcal{S}, \mathcal{T} \rangle$ is the $\mathcal{T}$-*instance* of $\mathcal{S}$.

4.3.4 *SCIFF-lite Specifications*

Beside syntactic differences, SCIFF-lite specifications extend CLIMB specifications by supporting arbitrary sets of abducible predicates, containining also:

(SOME) HAPPENED EVENTS In this way, happened events contained in the head of integrity constraints are hypothesized (i.e., generated), simulating their occurrence.

12 To impose that only "proper" answers are acceptable, the rule must be simply modified by introducing further temporal constraints on expectations.

13 Even if it is not apparent from the rule itself, it suffices for expressing "full" mutual exclusions between the two speech acts; see Example 4.14 on page 78 for a proof.

OTHER USER-DEFINED PREDICATES The modeler is equipped with the possibility of using integrity constraints to perform abductive reasoning in general. For example, a rule may contain a predicate constraining the data of an event, but whose definition is not known; it can be declared as abducible and subject to abductive reasoning.

DEFINITION 4.9 (SCIFF-lite specification). A *SCIFF-lite specification* is an abductive logic program $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ where:

- $\mathcal{A} \supseteq \{\mathbf{E}/2, \mathbf{EN}/2\}$;
- $\mathcal{KB}$ and $\mathcal{IC}$ obey to the syntax shown in Table 17, with the additional condition that *abducible predicates cannot be negated*.

4.3.5 *A Declarative Notion of Compliance*

CLIMB abductive explanations

The declarative semantics of CLIMB is given in two steps[14]. Being CLIMB specifications mapped to abductive logic programs, the first step is to characterize their abductive explanations.

DEFINITION 4.10 (Abductive Explanation). Given a CLIMB specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ and a trace $\mathcal{T}$, the abducible set $\Delta \subseteq \mathcal{A}$ is an *abductive explanation* for $\mathcal{S}_{\mathcal{T}}$ iff

$$Comp\,(\mathcal{KB} \cup \mathcal{T} \cup \Delta) \cup \mathsf{CET} \cup \mathsf{T}_{\mathcal{X}} \models \mathcal{IC}$$

where *Comp* is the three-valued completion of a theory [115], CET stands for Clark Equational Theory [56] and $\mathsf{T}_{\mathcal{X}}$ is the constraint theory [101] (parametrized by $\mathcal{X}$).

Fixing a desired CLP corresponds to instantiating the parameter $\mathcal{X}$. Therefore, different time structures (see Definition 4.1 on page 56) can be chosen without affecting the definition of abductive explanation.

The symbol $\models$ is interpreted in three valued logics. In this way, we can rely upon a three-valued model-theoretic semantics, as done, for instance, in a different context, by Fung and Kowalski [82] and by Denecker and De Schreye [65]. A three-valued semantics is a suitable choice when dealing with systems under incomplete knowledge; this is the case of open and heterogeneous EBSs, in which unpredictable events dynamically occur over time.

Note that a set of integrity constraints is entailed if each one is entailed. The following remark clarifies the impact of this observation on abductive explanations.

REMARK 4.3 (Abductive Explanations on Sub-sets). If Δ is an abductive explanation for $\langle \mathcal{KB}, \mathcal{IC} \rangle_{\mathcal{T}}$, then Δ is an abductive explanation for $\langle \mathcal{KB}, \mathcal{IC}' \rangle_{\mathcal{T}}$, where $\mathcal{IC}' \subseteq \mathcal{IC}$.

Intensional abductive explanations

Abductive explanations only contain ground abducibles; when vari-

14 In the following, we will focus on CLIMB specifications, but the same notion of declarative semantics holds for SCIFF-lite specifications as well.

ables are present, they are used as a shortcut to intensionally represent the set of all corresponding ground versions. For example, if we adopt a dense-time structure, $\mathbf{EN}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{refuse}(\mathsf{loc}(\mathsf{rabbit}))), \mathsf{T})$ is used to intensionally represent the set

$$\{\mathbf{EN}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{refuse}(\mathsf{loc}(\mathsf{rabbit}))), t) \mid t \in \mathbb{R}\}$$

whereas $\mathbf{E}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{inform}(\mathsf{loc}(\mathsf{rabbit}), X), T) \wedge T > 10$ is a shortcut for representing a ground expectation e s.t.

$$e \in \{\mathbf{E}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{inform}(\mathsf{loc}(\mathsf{rabbit}), r), t) \mid r \in \mathfrak{U}, t \in (10, \infty)\}$$

Another example, with sharing of variables between a positive and a negative expectation, is the following: $\mathbf{E}(a, T) \wedge T \geqslant 4 \wedge T < 7 \wedge \mathbf{EN}(b, T)$ is a shortcut for representing one of the following three sets (supposing that the time structure is $(\mathbb{N}, <)$:

$$\{\ \mathbf{E}(a, 4), \mathbf{EN}(b, 4)\ \}$$
$$\{\ \mathbf{E}(a, 5), \mathbf{EN}(b, 5)\ \}$$
$$\{\ \mathbf{E}(a, 6), \mathbf{EN}(b, 6)\ \}$$

Example 4.11 (Abductive explanations). *Let us consider the CLIMB specification $\mathcal{Q}$, described in Example 4.10, together with the execution trace*

$$\begin{aligned}\mathcal{T} = \{&\mathbf{H}(\mathsf{tell}(\mathsf{alice}, \mathsf{hatter}, \mathsf{query}-\mathsf{ref}(\mathsf{loc}(\mathsf{rabbit})))), 5),\\ &\mathbf{H}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{refuse}(\mathsf{loc}(\mathsf{rabbit})), 10)\ \}\end{aligned}$$

The abducible sets

$$\begin{aligned}\Delta_0 &= \emptyset\\ \Delta_1 &= \{\ \mathbf{E}(\mathsf{tell}(\mathsf{alice}, \mathsf{hatter}, \mathsf{query}-\mathsf{ref}(\mathsf{loc}(\mathsf{rabbit})))), 5)\ \}\end{aligned}$$

are not abductive explanations for $\mathcal{Q}_{\mathcal{T}}$ (due to integrity constraint (QR_3) *and* (QR_1) *respectively). Instead, the abducible sets*

$$\begin{aligned}\Delta_3 &= \Delta_1 \cup\\ &\quad \{\ \mathbf{E}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{inform}(\mathsf{loc}(\mathsf{rabbit}), \mathsf{garden})), 10),\\ &\quad \mathbf{EN}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{refuse}(\mathsf{loc}(\mathsf{rabbit})), T)\ \}\\ \Delta_4 &= \Delta_1 \cup \{\ \mathbf{E}(\mathsf{tell}(\mathsf{lewis}, \mathsf{alice}, \mathsf{refuse}(\mathsf{loc}(\mathsf{rabbit})), 10)\ \}\\ \Delta_5 &= \Delta_3 \cup \Delta_4\end{aligned}$$

are all abductive explanations for $\mathcal{Q}_{\mathcal{T}}$.

Consistency of CLIMB abductive explanations

The second step is to formally characterize the meaning of expectations, focusing on the relationship between positive and negative expectations and on the relationship between expectations and happened event. The first relationship is captured by the notion of **E**-consistency, which states that positive and negative expectations are conflicting concepts: the same event cannot be expected to happen and not to happen at the same time.

DEFINITION 4.11 (**E**-consistency). An abducible set Δ is **E**-*consistent* iff $\forall e \in \mathfrak{U}$ and $\forall t \in \mathfrak{T}$:

$$\{\mathbf{E}(e,t), \mathbf{EN}(e,t)\} \nsubseteq \Delta$$

EXAMPLE 4.12 (**E**-consistency and intensional representations). *Let us consider the abductive explanations of Example 4.11. The sets Δ_3 and Δ_4 are* **E**-*consistent, while Δ_5 is not* **E**-*consistent, because the same refusal is expected to happen and not to happen at time 10.*

Indeed, remember that $\mathbf{EN}(\mathtt{tell}(\mathtt{hatter}, \mathtt{alice}, \mathtt{refuse}(\mathtt{loc}(\mathtt{rabbit})), T)$ *is used to intensionally represent the (infinite) set of negative expectations on each ground time, 10 included.*

The intensional representation of expectations may sometimes cause confusion w.r.t. **E**-*consistency. For example, the set*

$$\{\ \mathbf{E}(ev, T_1) \wedge T_1 \geqslant 6 \wedge T_1 \leqslant 10, \mathbf{EN}(ev, T_2) \wedge T_2 > 7 \wedge T_2 < 10\ \}$$

is **E**-*consistent. If we adopt a discrete time structure, it intensionally represents* one *of the sets*

$$\{\ \mathbf{E}(ev, 6), \mathbf{EN}(ev, 8), \mathbf{EN}(ev, 9)\ \}$$
$$\{\ \mathbf{E}(ev, 7), \mathbf{EN}(ev, 8), \mathbf{EN}(ev, 9)\ \}$$
$$\{\ \mathbf{E}(ev, 10), \mathbf{EN}(ev, 8), \mathbf{EN}(ev, 9)\ \}$$

which are all **E**-*consistent.*

Fulfillment

The relationship between expectations and happened events is captured by the concept of *fulfillment*, which formalizes how expectations are satisfied by a given instance of the system. More specifically, it formalizes the intuitive notion that:

- positive expectations must have a corresponding matching event in the execution trace of the instance;
- negative expectations must have no corresponding matching event in the execution trace of the instance.

DEFINITION 4.12 (Fulfillment). Given a trace $\mathcal{T}$, the abducible set Δ is $\mathcal{T}$-*fulfilled* iff $\forall e \in \mathfrak{U}, \forall t \in \mathfrak{T}$:

$$\mathbf{E}(e,t) \in \Delta \Longrightarrow \mathbf{H}(e,t) \in \mathcal{T}$$
$$\mathbf{EN}(e,t) \in \Delta \Longrightarrow \mathbf{H}(e,t) \notin \mathcal{T}$$

By considering the abductive nature of expectations, fulfillment provides a notion of *hypotheses confirmation*:

- Expectations are hypothesized according to the running instance of the system (i.e. a partial execution trace) and the CLIMB specification; this leads to the formulation of abductive explanations, according to Definition 4.10.

- The execution of the system follows or deviates from the expected behaviour; in other words, the execution trace confirms or disconfirms the formulated expectations, according to Definition 4.12.

The overall picture depicted in Figure 17 shows how this two-ways relationship is established by the declarative semantics of CLIMB.

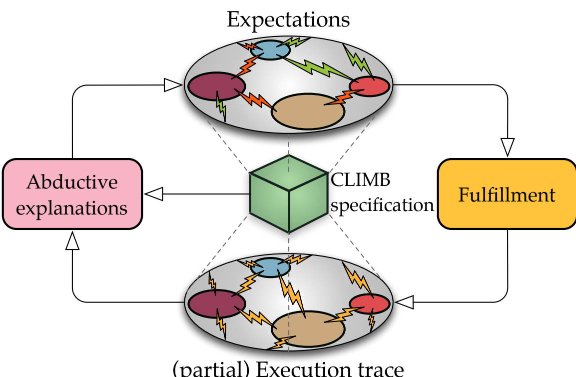


Figure 17: The two-ways relationship between expectations and happened events, established by the CLIMB declarative semantics.

Compliance

The most important aspect is that fulfillment can be interpreted as a way to isolate legal/correct executions from the wrong ones: the former satisfy all the expectations, while the latter omit an expected event or contain the occurrence of a forbidden event. Under this interpretation, the declarative semantics of CLIMB can be considered as a way to formally capture if an execution trace *complies with* a CLIMB specification or not. Through the notion of compliance, it is possible to identify, in the space of all execution traces, those traces that satisfy the integrity constraints of the specification under study.

DEFINITION 4.13 (Compliance). A trace $\mathcal{T}$ is *compliant* with a CLIMB specification $\mathcal{S}$ iff there exists an abducible set Δ s.t.:

A. Δ is an abductive explanation for $\mathcal{S}_{\mathcal{T}}$;

B. Δ is **E**-consistent;

C. Δ is $\mathcal{T}$-fulfilled.

In this case, we write $\text{COMPLIANT}_{\Delta}(\mathcal{S}_{\mathcal{T}})$ or simply $\text{COMPLIANT}(\mathcal{S}_{\mathcal{T}})$. Otherwise, we say that $\mathcal{S}$ is *violated* by $\mathcal{T}$ or that $\mathcal{T}$ is *not compliant* with $\mathcal{S}$, written $\neg\text{COMPLIANT}(\mathcal{S}_{\mathcal{T}})$.

EXAMPLE 4.13 (Compliant and non compliant execution traces). *Let us consider the CLIMB specification $\mathcal{Q}$ described in Example 4.10. The execution*

trace $\mathcal{T}$ introduced in Example 4.11 is compliant with $\mathcal{Q}$, because the expectations set Δ_4 defined in Example 4.11 is an **E***-consistent and $\mathcal{T}$-fulfilled abductive explanation for $\mathcal{Q}$.*

The execution traces

$$\begin{aligned}
\mathcal{T}_2 = \quad & \emptyset \\
\mathcal{T}_3 = \quad & \{\mathbf{H}(\mathsf{tell}(\mathsf{alice}, \mathsf{hatter}, \mathsf{query-ref}(\mathsf{loc}(\mathsf{rabbit}))), 5), \\
& \mathbf{H}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{refuse}(\mathsf{loc}(\mathsf{rabbit})), 10), \\
& \mathbf{H}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{inform}(\mathsf{loc}(\mathsf{rabbit}), \mathsf{garden})), 12)\} \\
\mathcal{T}_4 = \quad & \{\mathbf{H}(\mathsf{tell}(\mathsf{alice}, \mathsf{hatter}, \mathsf{query-ref}(\mathsf{loc}(\mathsf{rabbit}))), 5), \\
& \mathbf{H}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{inform}(\mathsf{loc}(\mathsf{rabbit}), \mathsf{garden})), 20)\}
\end{aligned}$$

are instead not compliant with $\mathcal{Q}$.

Due to rule (QR_3), each abductive explanation must contain a positive expectation about a query-ref event. Therefore, from the definition of fulfillment we have that $\mathcal{T}_2$ is not compliant with $\mathcal{Q}$: no happened event concerning the query-ref is contained in the trace, hence the positive expectation is violated.

By combining rule (QR_2) with the second happened event of $\mathcal{T}_3$, we have that each abductive explanation Δ_i for $\mathcal{Q}$ must contain the forbidding of the corresponding inform event:

$$\forall \Delta_i,\ \Delta_i \supseteq \{\ \mathbf{EN}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{inform}(\mathsf{loc}(\mathsf{rabbit}), \mathsf{garden})), T\ \}$$

Since this negative expectation has a corresponding matching occurrence in $\mathcal{T}_3$ (with $T/12$), there does not exist a Δ_i which is $\mathcal{T}_2$-fulfilled, and therefore $\mathcal{Q}$ is violated by $\mathcal{T}_3$. In particular, $\mathcal{T}_3$ violates rule (QR_2).

The first event of trace $\mathcal{T}_4$ satisfies rule (QR_3) and triggers rule (QR_1), generating a positive expectation concerning the answer. Abductive explanations must therefore contain an expectation e s.t.

$$e \in \{\mathbf{E}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{refuse}(\mathsf{loc}(\mathsf{rabbit}))), T) \mid T < 15\}$$

or

$$e \in \{\mathbf{E}(\mathsf{tell}(\mathsf{hatter}, \mathsf{alice}, \mathsf{inform}(\mathsf{loc}(\mathsf{rabbit})), L), T) \mid T < 15 \land L \in \mathfrak{U}\}$$

From the definition of fulfillment, e must have a corresponding matching event in $\mathcal{T}_4$, but this is not the case: no refusal is contained in $\mathcal{T}_4$, and the inform speech act is uttered by hatter *after the expiration of the (actual) deadline of 15.*

A graphical representation of the considered execution traces and their compliance w.r.t. $\mathcal{Q}$ is shown in Figure 18.

4.4 EQUIVALENCE AND COMPOSITIONALITY

Two interesting properties of SCIFF specifications are now introduced, namely equivalence and compositionality w.r.t. compliance, proving that a certain class of SCIFF specifications (which covers all the CLIMB specifications) is compositional.

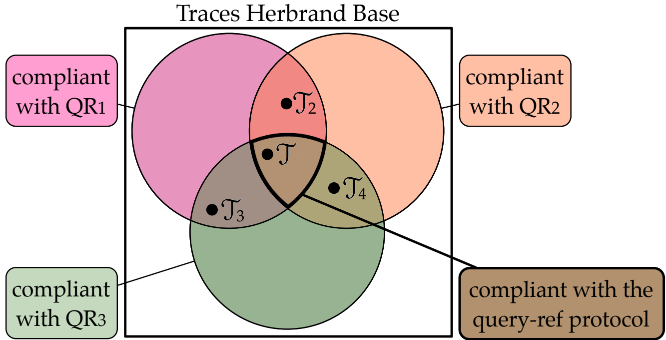


Figure 18: Compliance of the execution traces reported in Examples 4.11 and 4.13 with the query-ref protocol.

4.4.1 *Equivalence w.r.t. Compliance*

There are many different ways to characterize the same system, or, to be more precise, the same set of compliant execution traces. In other words, there are specifications different for what concerns their structure and their corresponding abductive explanations, but, at the same time, equivalent from an "external" point of view: given an execution trace, they will always agree when evaluating if the trace is compliant or not compliant. We say that these specifications are equivalent w.r.t. compliance.

Definition 4.14 (Equivalence w.r.t. compliance (between two SCIFF specifications)). Given two SCIFF specifications $\mathcal{S}^1$ and $\mathcal{S}^2$, $\mathcal{S}^1$ is *equivalent w.r.t. compliance* to $\mathcal{S}^2$ ($\mathcal{S}^1 \stackrel{c}{\sim} \mathcal{S}^2$) iff:

$$\forall \mathcal{T} \left(\textsc{compliant}(\mathcal{S}^1_{\mathcal{T}}) \Leftrightarrow \textsc{compliant}(\mathcal{S}^2_{\mathcal{T}}) \right)$$

This notion of equivalence is important because it provides a way to identify if two different specifications describe the same set of legal executions, ensuring that one specification can be seamlessly replaced with the other without affecting compliance. This opens the possibility of selecting the "best" among the two specifications w.r.t. a certain criterion, such as for example readability or compactness of the integrity constraints[15].

Example 4.14 (Equivalence w.r.t. compliance when modeling the not coexistence between two events). *Let us suppose that, in a given EBS, two events* e_1 *and* e_2 *cannot both occur within the same instance. To express the not coexistence between these two events, the modeler could rephrase the sentence as follows: if* e_1 *occurs, then* e_2 *must not occur, and if* e_2 *occurs,*

15 As we will see, an important criterion will be the impact of integrity constraints to the underlying verification technique.

then e_1 *must not occur. This formulation can be translated into* CLIMB *in a straightforward way:*

$$\mathbf{H}(e_1, T_a) \rightarrow \mathbf{EN}(e_2, T_b). \qquad (\otimes^{12})$$

$$\mathbf{H}(e_2, T_c) \rightarrow \mathbf{EN}(e_1, T_d). \qquad (\otimes^{21})$$

However, it is interesting to note that the two integrity constraints are redundant for what regards compliance: it is sufficient to use only one of them to express the not coexistence between e_1 *and* e_2*. In particular, it can be proven that*

$$\mathcal{S}^1 = \langle \emptyset, \{(\otimes^{12})\} \rangle \stackrel{c}{\sim} \mathcal{S}^2 = \langle \emptyset, \{(\otimes^{21})\} \rangle$$

Let us suppose, by reductio ad absurdum, that there exists an execution trace $\mathcal{T}$ *s.t.* COMPLIANT$(\mathcal{S}^1_{\mathcal{T}})$ *and* $\neg$COMPLIANT$(\mathcal{S}^2_{\mathcal{T}})$*. To violate* $\mathcal{S}^2$*,* $\mathcal{T}$ *must contain the occurrence of both* e_2 *and* e_1*. However, rule* $(\otimes^{21})$ *states that if* $\mathcal{T}$ *contains* e_1*, then it cannot contain also* e_2*, and therefore it is impossible that* $\mathcal{T}$ *is compliant with* $\mathcal{S}^1$*. The opposite case can be proven in the same way.*

4.4.2 *Compositionality w.r.t. compliance*

An interesting point now is to understand if and to what extent the integrity constraints of a SCIFF specification can be partitioned obtaining two separated specifications which are, together, equivalent w.r.t. compliance to the original one. This intuitive idea is formalized by the notion of *compositionality* w.r.t. compliance.

DEFINITION 4.15 (Compositionality). A SCIFF specification $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ is *compositional* iff $\forall \mathcal{T}, \forall \mathcal{IC}_1$ and $\forall \mathcal{IC}_2$ s.t. $\mathcal{IC} = \mathcal{IC}_1 \cup \mathcal{IC}_2$:

$$\begin{aligned} &\text{COMPLIANT}\,(\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC}_1 \rangle_{\mathcal{T}}) \\ \wedge &\text{COMPLIANT}\,(\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC}_2 \rangle_{\mathcal{T}}) \Leftrightarrow \text{COMPLIANT}\,(\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle_{\mathcal{T}}) \end{aligned}$$

A compositional specification $\mathcal{S}$ can be split up into simpler sub-specifications $\mathcal{S}_1, \ldots, \mathcal{S}_n$, each one covering a partition of its integrity constraints set, and compliance can be evaluated separately by each sub-specification: a trace is compliant with $\mathcal{S}$ iff it is compliant with each $\mathcal{S}_i$.

Replaceability of integrity constraints

Let us now consider the following situation. We have proven that a SCIFF specification containing one integrity constraint, say, IC, is equivalent w.r.t. compliance to a specification containing the integrity constraint IC′ (this is the case of Example 4.14). Formally, we have proven that $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \{IC\} \rangle \stackrel{c}{\sim} \mathcal{S}' = \langle \mathcal{KB}, \mathcal{A}, \{IC'\} \rangle$. This means that the two specifications can be substituted for each other without affecting compliance. But what happens now if the original specification does not contain only IC but also other integrity constraints, i.e., $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \{IC\} \cup \mathcal{IC} \rangle$? Can we still guarantee equivalence w.r.t. compliance by substituting IC with IC′ and maintaining the other integrity

constraints untouched? If the original specification ($\mathcal{S}$) is compositional, the following theorem claims that the answer is "yes": it holds that $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \{IC\} \cup \mathcal{IC} \rangle \overset{c}{\sim} \mathcal{S}' = \langle \mathcal{KB}, \mathcal{A}, \{IC'\} \cup \mathcal{IC} \rangle$.

THEOREM 4.1 (Replaceability of integrity constraints). *Given four compositional SCIFF specifications $\mathcal{S}^1 = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC}_1 \rangle$, $\mathcal{S}^{\cup 1} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \cup \mathcal{IC}_1 \rangle$, $\mathcal{S}^2 = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC}_2 \rangle$, $\mathcal{S}^{\cup 2} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \cup \mathcal{IC}_2 \rangle$, the following holds:*

$$\mathcal{S}^1 \overset{c}{\sim} \mathcal{S}^2 \Rightarrow \mathcal{S}^{\cup 1} \overset{c}{\sim} \mathcal{S}^{\cup 2}$$

This means that if one demonstrates that $\mathcal{S}^1 \overset{c}{\sim} \mathcal{S}^2$, then the integrity constraints (sub)set $\mathcal{IC}_1$ of $\mathcal{S}^{\cup 1}$ can be seamlessly replaced with $\mathcal{IC}_2$ without affecting compliance.

Proof. Let us consider $\alpha = \mathcal{S}^1 \overset{c}{\sim} \mathcal{S}^2$ and $\beta = \mathcal{S}^{\cup 1} \overset{c}{\sim} \mathcal{S}^{\cup 2}$. One has to prove that $\alpha \Rightarrow \beta$. From the definition of $\overset{c}{\sim}$ (Definition 4.14), we have

$$\alpha = \forall \mathcal{T} \ \textsc{compliant}\left(\mathcal{S}^1_{\mathcal{T}}\right) \Leftrightarrow \textsc{compliant}\left(\mathcal{S}^2_{\mathcal{T}}\right)$$

Since the specifications are compositional, we have:

$$\begin{aligned} \beta &= \forall \mathcal{T} \ \textsc{compliant}\left(\mathcal{S}^{\cup 1}_{\mathcal{T}}\right) \Leftrightarrow \textsc{compliant}\left(\mathcal{S}^{\cup 2}_{\mathcal{T}}\right) = && \text{(Def. 4.14)} \\ &= \forall \mathcal{T} \ \textsc{compliant}\left(\mathcal{S}_{\mathcal{T}}\right) \wedge \textsc{compliant}\left(\mathcal{S}^1_{\mathcal{T}}\right) \\ &\qquad \Leftrightarrow \textsc{compliant}\left(\mathcal{S}_{\mathcal{T}}\right) \wedge \textsc{compliant}\left(\mathcal{S}^2_{\mathcal{T}}\right) && \text{(Def. 4.15)} \end{aligned}$$

where $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$. Let us now separately discuss the case in which $\mathcal{S}_{\mathcal{T}}$ is compliant, and the case in which $\mathcal{S}_{\mathcal{T}}$ is not compliant, showing that in both cases $\alpha \Rightarrow \beta$ is $\mathtt{true}$.
If $\mathcal{S}_{\mathcal{T}}$ is not compliant (i.e., $\textsc{compliant}(\mathcal{S}_{\mathcal{T}})$ is $\mathtt{false}$), then

$$\begin{aligned} \beta &= \forall \mathcal{T} \ \mathtt{false} \wedge \textsc{compliant}\left(\mathcal{S}^1_{\mathcal{T}}\right) \Leftrightarrow \mathtt{false} \wedge \textsc{compliant}\left(\mathcal{S}^2_{\mathcal{T}}\right) \\ &= \forall \mathcal{T} \ \mathtt{false} \Leftrightarrow \mathtt{false} = \mathtt{true} \end{aligned}$$

Therefore, $\alpha \Rightarrow \beta = \alpha \Rightarrow \mathtt{true} = \mathtt{true}$.
If $\mathcal{S}_{\mathcal{T}}$ is compliant (i.e., $\textsc{compliant}(\mathcal{S}_{\mathcal{T}})$ is $\mathtt{true}$), then

$$\begin{aligned} \beta &= \forall \mathcal{T} \ \mathtt{true} \wedge \textsc{compliant}\left(\mathcal{S}^1_{\mathcal{T}}\right) \Leftrightarrow \mathtt{true} \wedge \textsc{compliant}\left(\mathcal{S}^2_{\mathcal{T}}\right) \\ &= \forall \mathcal{T} \ \textsc{compliant}\left(\mathcal{S}^1_{\mathcal{T}}\right) \Leftrightarrow \textsc{compliant}\left(\mathcal{S}^2_{\mathcal{T}}\right) = \alpha \end{aligned}$$

Therefore, $\alpha \Rightarrow \beta = \alpha \Rightarrow \alpha = \mathtt{true}$. □

Unchained specifications

An important question now is: given an arbitrary SCIFF specification, is it possible to know whether it is compositional or not? The answer is: if the syntax of the specification has a particular form (called *unchained*), then the specification is compositional. We define the class of unchained specifications and prove this claim, showing also that all CLIMB specifications are unchained, hence compositional.

DEFINITION 4.16 (unchained specification). A SCIFF specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ is *unchained* iff there does not exist an integrity constraint in $\mathcal{IC}$ containing an abducible in the body:

$$\forall IC \in \mathcal{IC}, \forall \beta_i \in \mathrm{body}(IC), \nexists \theta \text{ s.t. } [\beta_i]\theta \in \mathcal{A}$$

EXAMPLE 4.15 (unchained specifications). *Let us consider the integrity constraints*

$$\mathbf{H}(\mathrm{external}(e_1), T_1) \rightarrow \mathbf{H}(\mathrm{internal}(e_2), T_2). \quad (4.1)$$
$$\mathbf{H}(\mathrm{internal}(e_2), T_2) \rightarrow \mathbf{E}(\mathrm{external}(e_3), T_3). \quad (4.2)$$
$$\mathbf{H}(\mathrm{external}(e_3), T_3) \rightarrow \mathbf{EN}(\mathrm{external}(e_4), T_4). \quad (4.3)$$

and the SCIFF-lite specifications

$$\mathcal{S}_1 = \langle \emptyset, \{\mathbf{E}/2, \mathbf{EN}/2, \mathbf{H}(\mathrm{internal}(E), T)\}, \{(4.1), (4.2)\} \rangle$$
$$\mathcal{S}_2 = \langle \emptyset, \{\mathbf{E}/2, \mathbf{EN}/2, \mathbf{H}(\mathrm{internal}(E), T)\}, \{(4.1), (4.3)\} \rangle$$
$$\mathcal{S}_3 = \langle \emptyset, \{\mathbf{E}/2, \mathbf{EN}/2, \mathbf{H}/2\}, \{(4.1), (4.3)\} \rangle$$

$\mathcal{S}_2$ *is the only unchained specification, because the body of its integrity constraints contains the occurrence of an "external" event, and only "internal events" can be abduced.*

THEOREM 4.2 (Compositionality of unchained specifications). *If a SCIFF specification is unchained, then it is also compositional.*

Relationship between unchained specifications and compositionality

Proof. Let us consider the specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$. By taking into account also abducible sets, we rephrase compositionality as follows: for each execution trace $\mathcal{T}$

$$\text{COMPLIANT}_{\Delta_1}(\mathcal{S}^1_{\mathcal{T}}) \wedge \text{COMPLIANT}_{\Delta_2}(\mathcal{S}^2_{\mathcal{T}}) \Leftrightarrow \text{COMPLIANT}_{\Delta}(\mathcal{S}_{\mathcal{T}})$$

where $\mathcal{S}^1 = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC}_1 \rangle$, $\mathcal{S}^2 = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC}_2 \rangle$ and $\mathcal{IC} = \mathcal{IC}_1 \cup \mathcal{IC}_2$. One has to prove that suitable Δ_1 and Δ_2 exist iff a suitable Δ exist.
$\Leftarrow$
One has to prove that if an abducible set Δ exists s.t. $\mathcal{S}_{\mathcal{T}}$ is compliant, then also two abducible sets Δ_1 and Δ_2 must exist s.t. $\mathcal{S}^1_{\mathcal{T}}$ and $\mathcal{S}^2_{\mathcal{T}}$ are compliant.

This is trivial, by choosing $\Delta_1 = \Delta_2 = \Delta$. Remark 4.3 states that Δ is an abductive explanation for any specification $\langle \mathcal{KB}, \mathcal{IC}' \rangle$ where $\mathcal{IC}' \subseteq \mathcal{IC}$, hence also for $\mathcal{S}_1$ and $\mathcal{S}_2$.
$\Rightarrow$
One has to prove that if two abducible sets Δ_1 and Δ_2 exist s.t. $\mathcal{S}^1_{\mathcal{T}}$ and $\mathcal{S}^2_{\mathcal{T}}$ are compliant, then also an abducible set Δ must exist s.t. $\mathcal{S}_{\mathcal{T}}$ is compliant.

Let us consider $\Delta = \Delta_1 \cup \Delta_2$. The proof reduces to check that such Δ obeys the three properties required by the definition of compliance (Definition 4.13). Let us focus on each property separately:

A. Δ is an abductive explanation for $\mathcal{S}_{\mathcal{T}}$. Let us suppose that Δ is not an abductive explanation, i.e., that there exists an integrity

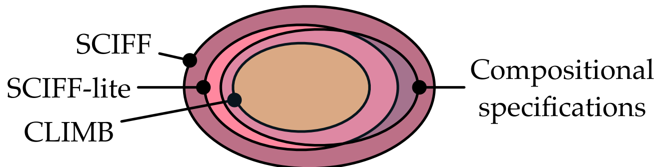


Figure 19: The set of compositional specifications compared to the variants of SCIFF.

constraint IC which is not entailed. Let us suppose $IC \in \mathcal{IC}_1$. Being Δ_1 and Δ_2 abductive explanations for $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, IC is not entailed only if $body(IC)$ is not made true by Δ_1, but it is made true by $\Delta_1 \cup \Delta_2$[16]. If it is the case, then some part of $body(IC)$ must unify with an element in Δ_2: $\exists \epsilon \in \Delta_2$, $\exists \beta \in body(IC)$ s.t. $[\beta]\theta = \epsilon$. However, since $\Delta_2 \subseteq \mathcal{A}$, the existence of such ϵ contradicts the hypotheses that $\mathcal{S}$ is unchained.

B. Δ is **E**-consistent. Let us suppose that Δ is not **E**-consistent; since Δ_1 and Δ_2 are both **E**-consistent, the only way to make Δ **E**-inconsistent is that there exist $e \in \mathfrak{U}$ and $t \in \mathfrak{T}$ s.t. $\mathbf{E}(e,t) \in \Delta_1$ and $\mathbf{EN}(e,t) \in \Delta_2$. But this is impossible, because both Δ_1 and Δ_2 are $\mathcal{T}$-fulfilled: the $\mathcal{T}$-fulfillment of Δ_1 would require that $\mathbf{H}(e,t) \in \mathcal{T}$, whereas the $\mathcal{T}$-fulfillment of Δ_2 would state that $\mathbf{H}(e,t) \notin \mathcal{T}$.

C. Δ is $\mathcal{T}$-fulfilled. Indeed, each element of Δ belongs to Δ_1 or Δ_2, which are both $\mathcal{T}$-fulfilled.

□

Relationship between CLIMB specifications and compositionality

COROLLARY 4.1 (CLIMB compositionality). *CLIMB specifications are compositional.*

Proof. By looking at Remark 4.2 and at Definitions 4.6 and 4.16, it becomes apparent that CLIMB specifications are unchained: the body of each CLIMB integrity constraint contains only happened events, which are not abducibles (only expectations can be abduced). Being CLIMB specifications unchained, Theorem 4.2 guarantees that they are compositional. □

By taking into account compositionality, the hierarchical schema of SCIFF specifications shown in Figure 14 becomes the one depicted in Figure 19.

16 This would require the abductive explanation of $\mathcal{S}_{\mathcal{T}}$ to contain also a disjunct of $head(IC)$, a requirement which is not satisfied by Δ.

5

TRANSLATING CONDEC TO CLIMB

Contents

5.1 Translation of a ConDec Model to CLIMB **84**
5.2 Translation of Events **85**
5.3 Embedding a Qualitative Characterization of Time in a Quantitative Setting **85**
5.3.1 Temporal Contiguity 85
5.3.2 Compact Execution Traces 86
5.4 Translation of Constraints **87**
5.4.1 Translation of Existence Constraints 88
5.4.2 Translation of Choice Constraints 89
5.4.3 Translation of Relation Constraints 91
5.4.4 Translation of Negation Constraints 94
5.4.5 Dealing with Branching ConDec Constraints 95
5.4.6 Equivalence Between ConDec Constraints 95
5.5 Soundness of the Translation **96**
5.5.1 Trace Mapping 96
5.5.2 Compliance Preservation 97
5.5.3 Proof of Soundness 97
5.6 On the Expressiveness of SCIFF **99**
5.6.1 A Separated Normal Form for LTL Formulae 100
5.6.2 Translation of SNF Formulae to SCIFF-lite 101
5.6.3 Translation of Arbitrary LTL Formulae to SCIFF-lite 104

In Chapters 4 and 3 we have discussed two different approaches for the specification of Event-Based Systems: CLIMB and ConDec (together with its LTL formalization). On the one hand, the two approaches have complementary scopes and objectives: while ConDec adopts an intuitive graphical notation and pays particular attention to the usability by non-IT savvy, CLIMB is based on a rigorous, rule-based language, it provides a formal characterization of its specification (centered around the notion of compliance), and it is therefore proned to verification. On the one hand, the two approaches have the same, underlying philosophy: they both rely on the notion of *constraint* as the basic mean to capture desired and forbidden courses of interaction, following an *open* and *declarative* approach. This Chapter exploits these similarities to provide a complete formalization of ConDec terms of CLIMB specifications. In this respect, it lays the foundation for the verification of

ConDec models along their entire lifecycle, which will be matter of the next parts of this dissertation.

We show how a ConDec model can be automatically translated to a CLIMB specification. Then, we discuss the relationship between CLIMB and propositional Linear Temporal Logic (LTL), which was the first language exploited for providing an underlying semantics to ConDec. Such a relationship is investigated along two different dimensions: a specific dimension related to ConDec aiming at proving that the provided translation to CLIMB is *sound* w.r.t. the original LTL mapping, and a general dimension, focused on the comparison between the two approaches for what concerns their *expressiveness*.

In this chapter, we make the assumption that the activities involved in the ConDec model are all *atomic*: they identify atomic units of work and can be represented by a single, punctual event. The next chapter will exploit more deeply the expressiveness of the CLIMB language, proposing several extensions to ConDec, such as the possibility of expressing and formalizing non-atomic activities and time/data-related constraints.

5.1 TRANSLATION OF A CONDEC MODEL TO CLIMB

A ConDec model is translated to a CLIMB specification by mapping its mandatory constraints to CLIMB Integrity Constraints. In this way, an execution trace is supported by the model iff it is compliant with the obtained CLIMB specification. In this respect, optional constraints are not part of the formalization, because they are not employed to effectively constrain the interaction, but only as a mean to express preferable executions, alerting the user if they are not met. Chapter 14 will show how SCIFF-lite can be employed to deal also with optional constraints.

Since the basic ConDec models do not support data, the KB of the obtained specification will be empty. Chapter 6 will introduce an extended version of ConDec, able to deal with data as well; in this setting, the KB can be exploited to formalize data-related decisions and background knowledge.

The translation of ConDec to CLIMB is encapsulated in a translation function called t_{CLIMB}.

DEFINITION 5.1 (ConDec to CLIMB translation). t_{CLIMB} is a function which translates a ConDec model $\mathcal{CM} \triangleq \langle \mathcal{A}, \mathcal{C}_m, \mathcal{C}_o \rangle$ to a CLIMB specification as follows:

$$t_{CLIMB} : \qquad \mathcal{CM} \longmapsto t_{CLIMB}(\mathcal{CM}) = \langle \emptyset, t_{IC}(\mathcal{C}_m) \rangle$$

where t_{IC} is a function capable to translate a set of ConDec mandatory constraints to a set of CLIMB ICs.

By looking at Definition 5.1, we can notice that only activities connected to a mandatory constraint will take part of the produced CLIMB formalization, and that unconstrained activities will not appear at all.

The openness of CLIMB specifications guarantees that these activities can be executed an arbitrary number of times, without producing an impact on compliance; such a behavior is in accordance with the intended ConDec semantics, pointed out in Chapter 3.

Finally, following the intuitive principle that an execution trace is supported by a ConDec model iff all its constraints are satisfied, the application of the t_{IC} to an entire set $\mathcal{C}_m$ of mandatory constraints corresponds to the union of the ICs obtained by applying t_{IC} to each mandatory ConDec constraint:

$$t_{IC}(\mathcal{C}_m) = \bigcup_{C_i \mid C_i \in \mathcal{C}_m} t_{IC}(C_i)$$

In this way, similarly to the case of LTL (discussed in Section 3.7), the *support* provided by a ConDec model w.r.t. a given execution trace is reduced, in the context of CLIMB, to the declarative notion of compliance.

DEFINITION 5.2 (Supported execution trace in the CLIMB setting). Given a CLIMB execution trace $\mathcal{T}$ and a ConDec model $\mathcal{CM}$, $\mathcal{T}$ is *supported* by $\mathcal{CM}$ iff:

$$\textsc{compliant}\left(t_{IC}(\mathcal{CM})_{\mathcal{T}}\right)$$

5.2 TRANSLATION OF EVENTS

Each atomic ConDec activity can be simply translated to a CLIMB term; being atomic, its (non)expected or occurred execution is associated to a single time-point, and is therefore formalizable by means of a single expectation/happened event.

DEFINITION 5.3 (Translation of an atomic ConDec activity). Given a ConDec atomic activity a,

- $\mathbf{H}(exec(a), T)$ states that a has been executed at time T;
- $\mathbf{E}(exec(a), T)$ states that a is expected to be executed at time T;
- $\mathbf{EN}(exec(a), T)$ states that a cannot be executed at time T.

5.3 EMBEDDING A QUALITATIVE CHARACTERIZATION OF TIME IN A QUANTITATIVE SETTING

As discussed in Chapters 4 and 3, ConDec relies on a qualitative point-based characterization of time, while CLIMB adopts a point-based quantitative characterization, where CLP constraints can be used to express both qualitative and quantitative constraints on time points.

5.3.1 *Temporal Contiguity*

From an ontological viewpoint, moving from ConDec to CLIMB requires to define how the first characterization of time can be embedded

into the second one. We argue that, in the ConDec setting, time does not flow independently from the interaction, but is instead marked by the occurring of events. The state of affairs reached during the execution is determined and changed only by the happening of a new event (such as performing an activity): when the execution is quiescent, time remains idle as well. To encode this concept in a quantitative setting, we could imagine that the current time clock remains fixed when the execution is quiescent, and then increases of one time units when a new activity (a set of concurrent activities) is (are) executed. What matters is not the quantitative value of time/date at which an event occurs, but only the ordering among event occurrences. Hence, the shift from a quantitative to a qualitative setting does not affect time ordering, but changes the concept of *temporal contiguity*: while in CLIMB *two occurred events are contiguous iff between them no further event has occurred*, in this new vision we can assume without loss of generality that *two occurred events are contiguous iff their execution times differ of a single time unit*. If we focus on the LTL setting, this empirical discourse has a formal counterpart: the temporal operator $\bigcirc$ precisely captures this notion of contiguity, and its semantics is that at a certain state t, $\bigcirc a$ holds iff a is executed in state $t + 1$ (see Section 3.6.3).

5.3.2 *Compact Execution Traces*

We define the notion of *compact* CLIMB execution trace to capture this new concept of temporal contiguity: a trace is compact iff all its contiguous events occur at times differing from one single time unit. The definition exploits the fact that if time is marked only by the execution of events, then it is not possible to have a time value, inside the trace, at which nothing happens.

DEFINITION 5.4 (Compact CLIMB execution trace). A CLIMB execution trace $\mathcal{T}$ is *compact* iff

$$\forall\, t \in [0, t_{max}],\ \exists\, \mathbf{H}(e, t) \in \mathcal{T}$$

where $t_{max} = \max(t \mid \exists\, \mathbf{H}(e, t) \in \mathcal{T})$.

EXAMPLE 5.1. *Let us consider the following execution traces:*

$$\begin{aligned}
\mathcal{T}_1 = \{\ & \mathbf{H}(exec\,(choose_item)\,, 0), \\
& \mathbf{H}(exec\,(choose_item)\,, 7), \\
& \mathbf{H}(exec\,(pay)\,, 15), \\
& \mathbf{H}(exec\,(send_receipt)\,, 30)\ \} \\
\mathcal{T}_2 = \{\ & \mathbf{H}(exec\,(choose_item)\,, 0), \\
& \mathbf{H}(exec\,(choose_item)\,, 1), \\
& \mathbf{H}(exec\,(pay)\,, 2), \\
& \mathbf{H}(exec\,(send_receipt)\,, 3)\ \}
\end{aligned}$$

$\mathcal{T}_2$ *is compact, while* $\mathcal{T}_1$ *is not.*

The translation presented in the following Sections will take into account such an issue. However, CLIMB execution traces come with quantitative time values associated to the events they are composed by. Let us consider the following example: during the execution of a BP, activity choose_item is performed at time 5, and then the order is paid at time 10. Between time 5 and 10, nothing happens, and therefore the two executions are temporally contiguous. In a qualitative setting, the two executions would be registered as consecutive; for example, the LTL trace representing such an execution would state that the valuation functions of the two events contain two consecutive integers. Instead, the CLIMB trace representing such an execution would contain two happened events at respectively time 5 and 10, thus loosing the qualitative notion of temporal contiguity. A way must be therefore defined to map an execution trace with quantitative time values to an execution trace where ordering is maintained, but contiguous occurred events differ of a single time unit. The `compact` function is introduced to deal with this requirement, and can be applied to an arbitrary CLIMB trace before evaluating its compliance w.r.t. the specification formalizing the ConDec regulatory model.

DEFINITION 5.5 (Compaction function). `compact` is a *compaction function* which translates an execution trace to a corresponding compact execution trace. It is defined as follows:

$$\begin{aligned} \texttt{compact}: \quad & \mathfrak{U}^{\mathbf{H}} \longrightarrow \mathfrak{U}^{\mathbf{H}} \\ & \mathcal{T} \longmapsto \mathcal{T}_{comp} \text{ s.t. } \forall\, \mathbf{H}(e,t),\ \mathbf{H}(e,t') \in \mathcal{T}_{comp} \\ & \qquad \text{having } t' = \|\{t_< \mid \mathbf{H}(e^*, t_<), t_< < t\}\| \end{aligned}$$

where $\mathfrak{U}^{\mathbf{H}}$ is the Herbrand universe built upon CLIMB happened events.

By considering the execution traces of Example 5.1, it holds that

$$\mathcal{T}_2 = \texttt{compact}(\mathcal{T}_1)$$

As pointed out in Chapter 4, one of the most interesting features of CLIMB is that it supports the possibility of expressing quantitative temporal constraints such as delays and deadlines. Chapter 6 will discuss how ConDec can be extended with such additional features, by maintaining a valid and complete mapping to CLIMB. Obviously, this extension is possible only if the quantitative nature of CLIMB is recovered, and therefore part of the Chapter will be dedicated to show how the formalization changes in order to reflect the quantitative notion of temporal contiguity.

5.4 TRANSLATION OF CONSTRAINTS

t_{IC} maps each ConDec constraint to one or more CLIMB ICs. Obviously, many possible different mappings can be provided, proving that they are equivalent w.r.t. compliance (see Definition 4.14); here we try to preserve as much as possible the simplicity and readability of the translation, in order to stress the similarities between the two approaches,

CONSTRAINT C	$t_{IC}(C)$
0 a	$\mathsf{true} \rightarrow \mathbf{EN}(\mathsf{exec}(a), T).$
0..n a	$\bigwedge_{i=1}^{n} \mathbf{H}(\mathsf{exec}(a), T_i) \wedge T_i > T_{i-1} \rightarrow \mathbf{EN}(\mathsf{exec}(a), T_{n+1}) \wedge T_{n+1} > T_n.$
n..* a	$\mathsf{true} \rightarrow \bigwedge_{i=1}^{n} \mathbf{E}(\mathsf{exec}(a), T_i) \wedge T_i > T_{i-1}.$
n a	t_{IC} (n..* [a]) $\cup\, t_{IC}$ (0..n [a])
init a	$\mathsf{true} \rightarrow \mathbf{E}(\mathsf{exec}(a), T) \wedge \mathbf{EN}(\mathsf{exec}(X), T_X) \wedge T_X < T.$

Table 18: Translation of ConDec existence constraints to CLIMB. We assume $T_0 = 0$.

and to show that the natural language description of each ConDec constraint can be represented in a straightforward manner by means of CLIMB rules.

5.4.1 *Translation of Existence Constraints*

AbsenceN constraints

The application of the translation function on existence constraints is shown in Table 18. `AbsenceN` constraints express a prohibition about executing the involved activity too many times, and are therefore formalized by exploiting negative expectations. In the general case (prohibition of executing activity a n times), the formalization states that a is expected not to be executed if n different executions have already occurred[1]; as a consequence, a compliant execution trace must contain at most $n-1$ different executions of a. The intended meaning of "different executions" is "executions at different times"; in fact, each atomic activity can be executed, at a given time, at most once. Since CLIMB adopts an explicit point-based notion of time, the difference between executions of the same activity is modeled as a difference between their corresponding execution times, which in turn is captured by means of CLP constraints.

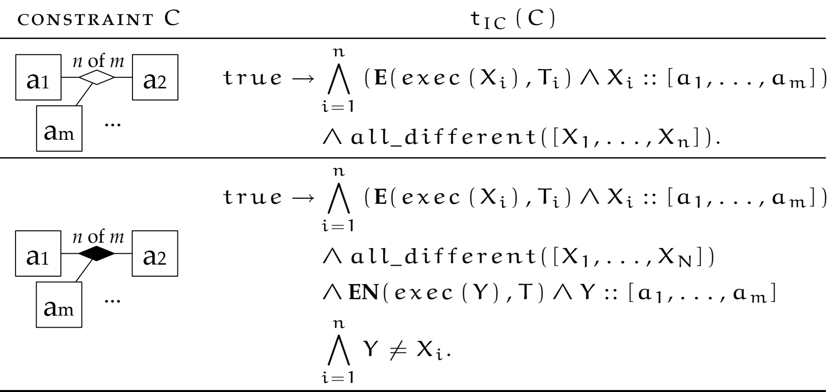
ExistenceN constraints

In a specular way, `existenceN` constraints express that the involved activity is expected to be executed at least a minimum number of times. An `existenceN` constraint stating that the a must be executed at least n times is therefore represented by imposing a conjunction of n different expectations on the execution of a.

ExactlyN constraints

A proper combination of the `absenceN` and `existenceN` constraints can be used to formalize the `exactlyN` one. In particular, expressing that an activity a must be executed exactly n times can be reformu-

1 The borderline case in which $n = 0$ is captured by the `absenceN` constraint, which simply states that the execution of a is prohibited.



CONSTRAINT C	$t_{IC}(C)$
a_1 — *n* of *m* — a_2, a_m ...	$\mathtt{true} \rightarrow \bigwedge_{i=1}^{n} (\mathbf{E}(\mathtt{exec}(X_i), T_i) \wedge X_i :: [a_1, \ldots, a_m])$ $\wedge\ \mathtt{all_different}([X_1, \ldots, X_n])$.
a_1 — *n* of *m* — a_2, a_m ...	$\mathtt{true} \rightarrow \bigwedge_{i=1}^{n} (\mathbf{E}(\mathtt{exec}(X_i), T_i) \wedge X_i :: [a_1, \ldots, a_m])$ $\wedge\ \mathtt{all_different}([X_1, \ldots, X_N])$ $\wedge\ \mathbf{EN}(\mathtt{exec}(Y), T) \wedge Y :: [a_1, \ldots, a_m]$ $\bigwedge_{i=1}^{n} Y \neq X_i$.

Table 19: Translation of ConDec choice constraints to CLIMB.

lated by stating that a is expected to be executed at least n times and cannot be executed more than n times. From the CLIMB viewpoint, an execution trace containing n executions of a:

- satisfies all the n expectations imposed by $t_{IC}\left(\boxed{\mathsf{a}}^{\,n..*}\right)$;
- triggers $t_{IC}\left(\boxed{\mathsf{a}}^{\,0..n+1}\right)$, generating a negative expectation about further executions of a.

Init constraint

Finally, the `init` constraint is modeled by imposing that a is expected to be executed as the first activity. The concept of "first" is formalized by means of a general negative expectation, which forbids the execution of all activities before the time at which a is expected to be executed.

5.4.2 *Translation of Choice Constraints*

The formalization of choice constraints is reported in Table 19. It is a natural extension of the one concerning the `existenceN` constraint. None of them has a triggering condition (i.e., the body of the corresponding ICs does not contain happened events): the involved expectations are always generated and must be fufilled independently from the course of interaction. However, while the `existenceN(n, a)` constraint imposes n different expectations on the same activity a, `choice(n of m, [a1,...,am])` imposes n expectations on n different activities belonging to the set $\{a_1, \ldots, a_m\}$; such n activities are not fixed by the constraint, but must be anyway chosen among $a_1, \ldots, a_m$. This intended meaning is formalized in two steps:

- each expectation is not imposed directly on a concrete activity, but on a variable X_i which belongs to the set $\{a_1, \ldots, a_m\}$; this latter requirement is expressed by means of the CLP *membership constraint* (::). The membership constraint $V :: \mathtt{List}$ states that V is

a discrete variable ranging over the domain of elements in `List`. Intuitively, it may considered as a compact way to impose, in a disjunctive non-deterministic manner, that V must unify with one of the elements in `List`.

- Difference between expectations is not imposed on the involved times, as in case of `existenceN`, but on the expected activities[2]. This is achieved by using the `all_different(List)` global constraint, which supports a compact way to express that all the elements of `List` must be different, and is handled by the underlying CLP solvers in an efficient way[200].

As the following example points out, the combined use of the membership CLP constraint makes it possible to maintain the formalization as concise as possible, avoiding the combinatorial explosion experienced when disjunctions are explicitly introduced in the head of the Integrity Constraints (ICs).

EXAMPLE 5.2 (Formalization of a choice constraint). *Let us consider the following statements, expressing a fragment of a business process in which the customer must be alerted:*

A. *the user can be alerted by ordinary mail, by e-mail, by SMS or by executing a vocal call;*

B. *to ensure that the alert is correctly delivered to the user, she must be alerted using at least two different methods.*

The first statement points out that four different activities are available to alert the user, while the second statement suggests that they must be interconnected by means of a "2 of 4" choice constraint. The resulting ConDec model is depicted in Figure 20.

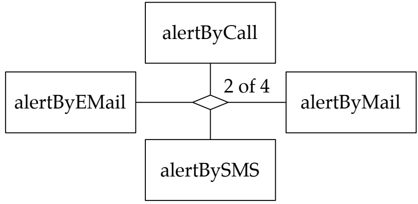


Figure 20: A simple ConDec model containing a choice constraint.

By following the directive of Table 19, the `choice` *constraint of the model can be represented as follows.*

2 In fact, the n parameter ranges over activities, and not on the executions of the same activity; furthermore, it may be possible that two different activities are executed concurrently (i.e., at the same time), and therefore time variables must be left unconstrained.

t_{IC} (`choice(2 of 4,[alertByEMail,alertByMail,alertBySMS,alertByCall])`) =

$$\begin{aligned} true \rightarrow & X_1 :: [alertByEMail, alertByMail, alertBySMS, alertByCall] \\ & \wedge X_2 :: [alertByEMail, alertByMail, alertBySMS, alertByCall] \\ & \wedge \mathbf{E}(X_1, T_1) \wedge \mathbf{E}(X_2, T_2) \wedge all_different([X_1, X_2]). \end{aligned}$$

This compact IC could be rewritten by removing the membership constraints and making an explicit use of disjunctions of expectations, enumerating all the possible (combination of) choices:

$$\begin{aligned} true \rightarrow & \mathbf{E}(alertByEMail, T_{1,1}) \wedge \mathbf{E}(alertByMail, T_{1,2}) \\ & \vee \mathbf{E}(alertByEMail, T_{2,1}) \wedge \mathbf{E}(alertBySMS, T_{2,2}) \\ & \vee \mathbf{E}(alertByEMail, T_{3,1}) \wedge \mathbf{E}(alertByCall, T_{3,2}) \\ & \vee \mathbf{E}(alertByMail, T_{4,1}) \wedge \mathbf{E}(alertBySMS, T_{4,2}) \\ & \vee \mathbf{E}(alertByMail, T_{5,1}) \wedge \mathbf{E}(alertByCall, T_{5,2}) \\ & \vee \mathbf{E}(alertBySMS, T_{6,1}) \wedge \mathbf{E}(alertByCall, T_{6,2}) \end{aligned}$$

Although the two formulations are equivalent w.r.t. compliance, the head of the first IC contains a single disjunct, while the head of the second one contains 6 *disjuncts. For a generic* `n of m` *choice, the number of disjuncts required for the first formulation is always equal to* 1*, while the number of disjuncts required for the second is equal to* $\binom{m}{n}$.

Choice vs exclusive choice constraints

The formalization of `ex_choice` contains an additional part, used to impose that the others $m - n$ activities belonging to $\{a_1, \ldots, a_m\}$ (the not chosen ones) cannot be executed at all. To express such a behaviour, a negative expectation is imposed on an activity variable which belongs to $\{a_1, \ldots, a_m\}$ but is different than all the n expected activities (represented by the activity variables $X_1, \ldots, X_n$). Since Y appears only in a negative expectation, it is universally quantified, and therefore this additional part states that all the activities belonging to $\{a_1, \ldots, a_m\}/\{X_1, \ldots, X_n\}$ are forbidden.

5.4.3 *Translation of Relation Constraints*

The translation of relation constraints strictly adheres to the guidelines described in Table 20. These guidelines establish a precise and univocal link between each single graphical element of the constraint and the corresponding (part of the) formalization.

Following these guidelines, Table 21 characterizes the t_{IC} function for each relation constraint.

Interposition constraint

In this respect, two points deserve a close examination. First of all, it is worth noting that the formalization of alternate constraints relies on a further constraint, called `interposition` (`interpos` for short), which is not a first-class relationship in the ConDec setting. It is a ternary constraint, where `interpos(a, b, c)` states that activity `b` must be executed between every execution of activity `a` and every following execution of activity `c`. The three activities should not necessarily be different; in fact, when interposition is used to specify part of the alternate constraint, the first and the third activity are always the same

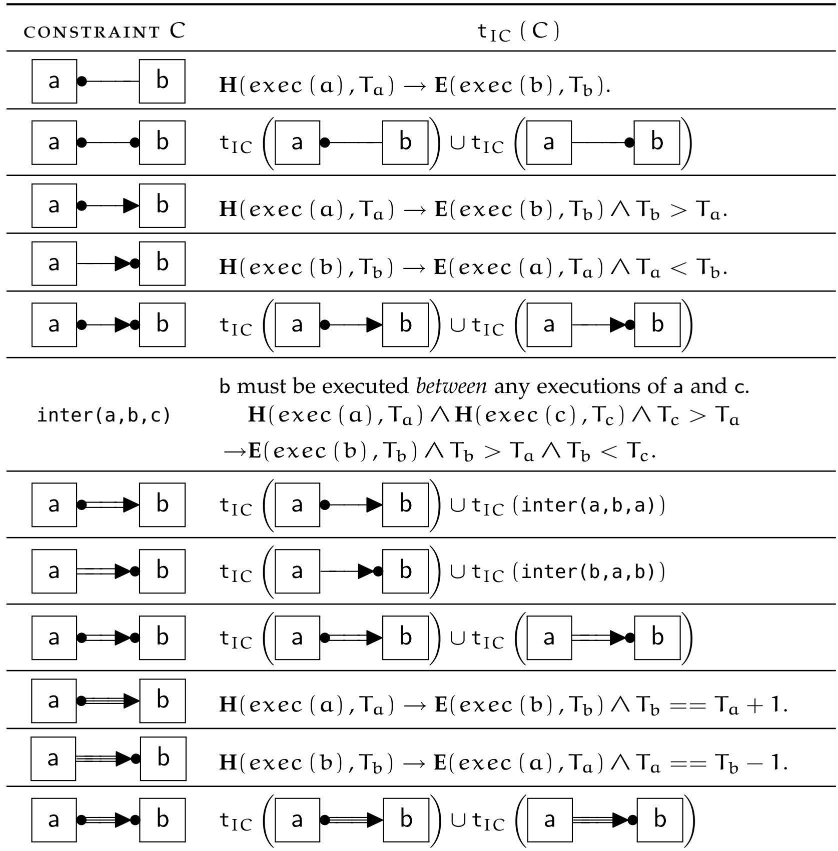
ASPECT	DESCRIPTION	CLIMB REPRESENTATION
BINDING		
[a]●—	Presence of ● means that a is the constraint's *source*, its execution has the effect of triggering the constraint.	It will contain **H**(exec (a), T) in the body.
—[a]	Absence of ● means that a is the constraint's *target*, its execution is therefore expected/ forbidden when the constraint is triggered.	It will contain **E**(exec (a), T) or **EN**(exec (a), T) in the head.
ORDERING		
—	The constraint does not involve any temporal ordering.	It will not contain CLP constraints over time variables.
—▶	The constraint involves a forward temporal ordering ("after")	It will contain a CLP constraint stating that the target time is greater than the source time.
—▶●	The constraint involves a backward temporal ordering ("before")	It will contain a CLP constraint stating that the source and target time are contiguous.
STRENGTH		
—	Basic constraint.	The constraint is mapped to a single IC (two iff it is a succession constraint).
=	Alternate constraint (the target activity must/cannot be executed between two executions of the source).	The formalization is decomposed in two parts: one expressing its basic behavior, one expressing the alternation.
≡	Chain constraint (source and target activity must/cannot be next to each other).	A CLP constraint is used to denote that the source and target times differ of one single time unit.
KIND		
—	Relation constraint.	Use of positive expectations.
╫	Negation constraint.	Use of negative expectations.
`succession`	It is a compact way to express two mutual constraints (a `response` and `precedence` one).	The formalization is decomposed in two parts, which separately capture the `response` and `precedence` part.

Table 20: Guidelines followed for translating ConDec relation and negation constraints to CLIMB.

(i.e., interpositions like `interpos(a, b, a)` are employed). In this way, interposition is used to state that a certain activity must be executed between two different executions of another activity.

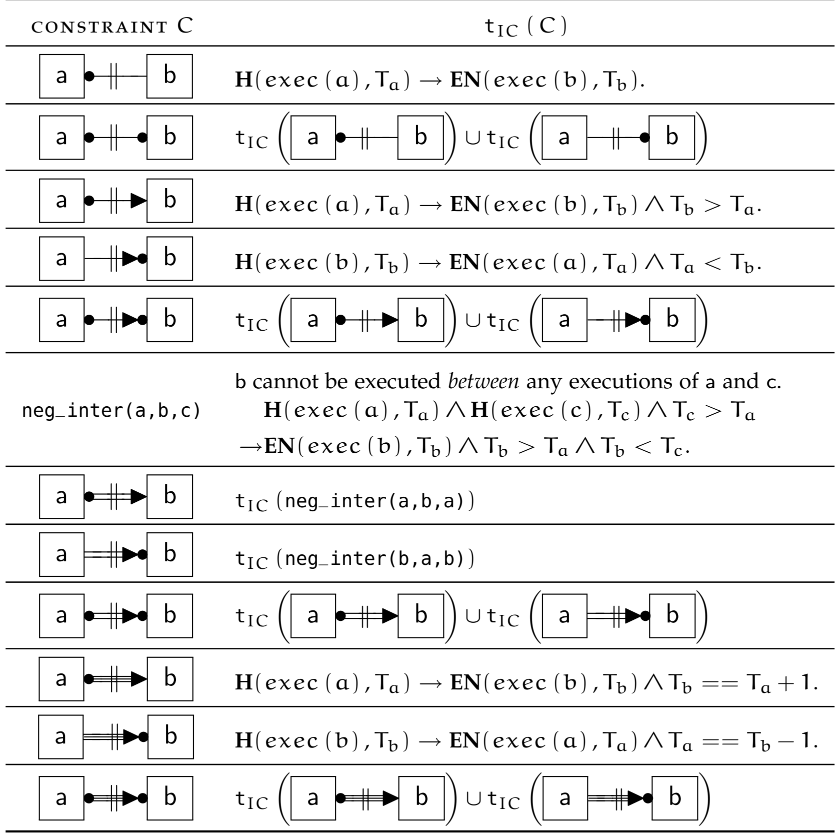
Chain relations and temporal contiguity

A second important issue regards the formalization of chain relations. Chain relations informally states that the involved activities have to be executed *next to each other*, i.e., that their execution times are contiguous. As discussed in Section 5.3, formalizing temporal contiguity



CONSTRAINT C	$t_{IC}(C)$
a b	$\mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b).$
a b	t_{IC} (a b) $\cup$ t_{IC} (a b)
a b	$\mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b) \land T_b > T_a.$
a b	$\mathbf{H}(exec(b), T_b) \rightarrow \mathbf{E}(exec(a), T_a) \land T_a < T_b.$
a b	t_{IC} (a b) $\cup$ t_{IC} (a b)
`inter(a,b,c)`	b must be executed *between* any executions of a and c. $\mathbf{H}(exec(a), T_a) \land \mathbf{H}(exec(c), T_c) \land T_c > T_a$ $\rightarrow \mathbf{E}(exec(b), T_b) \land T_b > T_a \land T_b < T_c.$
a b	t_{IC} (a b) $\cup$ t_{IC} (`inter(a,b,a)`)
a b	t_{IC} (a b) $\cup$ t_{IC} (`inter(b,a,b)`)
a b	t_{IC} (a b) $\cup$ t_{IC} (a b)
a b	$\mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b) \land T_b == T_a + 1.$
a b	$\mathbf{H}(exec(b), T_b) \rightarrow \mathbf{E}(exec(a), T_a) \land T_a == T_b - 1.$
a b	t_{IC} (a b) $\cup$ t_{IC} (a b)

Table 21: Translation of ConDec relation constraints to CLIMB.



CONSTRAINT C	$t_{IC}(C)$
	$\mathbf{H}(exec(a), T_a) \rightarrow \mathbf{EN}(exec(b), T_b)$.
	$t_{IC}(\ldots) \cup t_{IC}(\ldots)$
	$\mathbf{H}(exec(a), T_a) \rightarrow \mathbf{EN}(exec(b), T_b) \wedge T_b > T_a$.
	$\mathbf{H}(exec(b), T_b) \rightarrow \mathbf{EN}(exec(a), T_a) \wedge T_a < T_b$.
	$t_{IC}(\ldots) \cup t_{IC}(\ldots)$
`neg_inter(a,b,c)`	`b` cannot be executed *between* any executions of `a` and `c`. $\mathbf{H}(exec(a), T_a) \wedge \mathbf{H}(exec(c), T_c) \wedge T_c > T_a \rightarrow \mathbf{EN}(exec(b), T_b) \wedge T_b > T_a \wedge T_b < T_c$.
	t_{IC} (`neg_inter(a,b,a)`)
	t_{IC} (`neg_inter(b,a,b)`)
	$t_{IC}(\ldots) \cup t_{IC}(\ldots)$
	$\mathbf{H}(exec(a), T_a) \rightarrow \mathbf{EN}(exec(b), T_b) \wedge T_b == T_a + 1$.
	$\mathbf{H}(exec(b), T_b) \rightarrow \mathbf{EN}(exec(a), T_a) \wedge T_a == T_b - 1$.
	$t_{IC}(\ldots) \cup t_{IC}(\ldots)$

Table 22: Translation of ConDec negation constraints to CLIMB.

is a central issue w.r.t. the qualitative vs quantitative characterization of the temporal dimension. In particular, when the qualitative characterization of ConDec is embedded in the quantitative setting of CLIMB, then the concept of temporal contiguity can be encoded by stating that two execution times must be consecutive to each other (i.e., their difference must be equal to 1). Table 21 provides a formalization of chain relations which reflects such an idea.

Chapter 6 will discuss the introduction of quantitative temporal constraints in ConDec by exploiting the pecularities of CLIMB, showing also how the formalization of chain relationships must be changed accordingly.

5.4.4 *Translation of Negation Constraints*

As pointed out in Table 20, negation constraints are mapped in the same way as relation constraints, but by substituting positive expectations with negative ones. The result is shown in Table 22.

5.4.5 *Dealing with Branching ConDec Constraints*

As far as now, we have presented the formalization of relation and negation constraints by considering always only two interconnected activities. As dicussed in Section 3.3.5, ConDec supports the possibility of *branching* constraints, spanning three or more activities. Generally speaking, the presence of branches is interpreted in a disjunctive manner, and introduces a choice for those who execute the model. It therefore comes as no surprise that the extension of the formalization presented in the basic binary case resembles the one presented for the `choice` constraint: happened events and expectations now involve activity variables instead of concrete activities, imposing the membership of such variables to the sets of source/target activities belonging to the ConDec constraint.

Branched coexistence constraint

For example, the branched `coexistence` constraint is formalized as follows:

$$t_{IC}\left(\text{[branched coexistence: } a_1, a_2, \dots, a_n]\right) \triangleq \begin{array}{l}\mathbf{H}(X, T_X) \wedge X :: [a_1, \dots, a_n] \\ \rightarrow \mathbf{E}(Y, T_Y) \wedge X \neq Y \wedge Y :: [a_1, \dots, a_n].\end{array}$$

Branched response constraint

Similarly, the generalized `response` constraint, with both branched source and target, is formalized in the following way:

$$t_{IC}\left(\text{[branched response: } s_1, \dots, s_n \rightarrow t_1, \dots, t_m]\right) \triangleq \begin{array}{l}\mathbf{H}(X, T_X) \wedge X :: [s_1, \dots, s_n] \\ \rightarrow \mathbf{E}(Y, T_Y) \wedge T_Y > T_X \wedge Y :: [t_1, \dots, t_m].\end{array}$$

Other ConDec relation and negation extended constraints could be formalized by exactly following this procedure.

It is worth noting that while the use of membership CLP constraints in the head implicitly models a disjunction, their effect in the body is to replicate the IC for each possible choice. For the generalized `response` constraint shown above, this means that only one IC is produced, instead of n ICs, differing for what concerns the triggering happened event.

5.4.6 *Equivalence Between ConDec Constraints*

Thanks to the mapping of ConDec constraints to CLIMB, it is possible to prove whether two different constraints are equivalent w.r.t. compliance (see Definition 4.14). In particular, in [186] van der Aalst and Pesic pointed out that negation constraints can be reduced to a core set; through the application of the t_{CLIMB} function and the notion of equivalence w.r.t. compliance, it is possible to provide a formal account for such a reduction.

For example, Example 4.14 at Page 78 proves that:

$$t_{CLIMB}\left(\boxed{a}\bullet\!\!-\!\!\!-\boxed{b}\right) \overset{c}{\sim} t_{CLIMB}\left(\boxed{a}-\!\!\!-\!\!\bullet\boxed{b}\right)$$

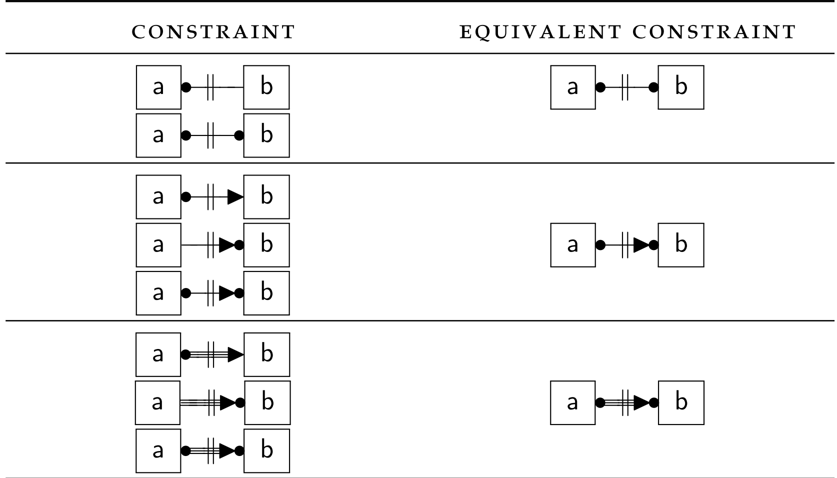


Table 23: Equivalence of ConDec negation constraints[186].

and, as a consequence, it holds that

$$t_{\text{CLIMB}}\left(\boxed{a}\bullet\!\!-\!\!-\!\!-\boxed{b}\right) \stackrel{c}{\sim} t_{\text{CLIMB}}\left(\boxed{a}\bullet\!\!-\!\!-\!\!-\!\!\bullet\boxed{b}\right)$$

Following the approach used in Example 4.14, all the equivalences introduced in [186], which are listed in Table 23, can be easily proven by exploiting the mapping to CLIMB.

5.5 SOUNDNESS OF THE TRANSLATION

Thanks to the translation presented in this chapter, ConDec has two possible different underlying semantics: LTL and CLIMB. However, an important question arises: do the two mappings effectively provide the same semantics for a given ConDec constraint? In other words, is the proposed formalization to CLIMB *sound* w.r.t. the original one, based on LTL?

We discuss this issue introducing an isomorphism between LTL and CLIMB execution traces, and reducing the concept of soundness to the one of compliance preservation.

5.5.1 *Trace Mapping*

In order to be able to compare an LTL formula with a CLIMB specification, the first step is to define how an execution trace in one setting can be mapped to an equivalent execution trace in the other setting. Note that the following Definitions are given on general SCIFF specifications, and are therefore all valid for CLIMB.

DEFINITION 5.6 (Trace mapping). A *trace mapping* tm is an isomorphism which maps LTL execution traces $\mathcal{T}_{\mathcal{L}}$ to SCIFF traces:

$$\begin{aligned} \mathtt{tm}: \quad & (\mathbb{N}, <, \nu_{occ}) \longrightarrow \mathfrak{U}^{\mathbf{H}} \\ & \mathcal{T}_{\mathcal{L}} \longmapsto \mathcal{T} = \{\mathbf{H}(exec(e), t) | t \in \mathcal{T}_{\mathcal{L}}(e)\} \end{aligned}$$

where $\mathfrak{U}^{\mathbf{H}}$ is the Herbrand universe built upon SCIFF happened events.

EXAMPLE 5.3. *Let us consider an LTL execution trace* $\mathcal{T}_{\mathcal{L}} = (\mathbb{N}, <, \nu_{occ})$, *where* $\mathcal{E} = \{choose_item, pay, send_receipt, report_failure\}$ *and:*

$$\nu_{occ}(choose_item) = \{0, 1\} \qquad \nu_{occ}(pay) = \{2\}$$
$$\nu_{occ}(send_receipt) = \{3\} \qquad \nu_{occ}(report_failure) = \emptyset$$

Then

$$\begin{aligned} \mathtt{tm}[\mathcal{T}_{\mathcal{L}}] = \{ & \mathbf{H}(exec(choose_item), 0), \\ & \mathbf{H}(exec(choose_item), 1), \\ & \mathbf{H}(exec(pay), 2), \\ & \mathbf{H}(exec(send_receipt), 3) \} \end{aligned}$$

5.5.2 *Compliance Preservation*

Thanks to the trace mapping function tm, it is possible to evaluate whether the "same" execution trace complies with an LTL and a CLIMB specification: if the outcomes agree, then the two specifications impose the same relevant set of constraints on that trace. Generalizing, if the outcomes agree for all the possible traces, then the two specifications are "behaviourally" sound.

DEFINITION 5.7 (Compliance preservation). Given a SCIFF specification $\mathcal{S}$ and an LTL formula ϕ, $\mathcal{S}$ *preserves compliance* w.r.t. ϕ ($\phi \overset{c}{\leftrightsquigarrow} \mathcal{S}$) iff:

$$\forall \text{ LTL trace } \mathcal{T}_{\mathcal{L}},\ \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \phi \Longleftrightarrow \text{COMPLIANT}\left(\mathcal{S}_{\mathtt{tm}[\mathcal{T}_{\mathcal{L}}]}\right)$$

5.5.3 *Proof of Soundness*

When it comes to the ConDec setting, the proof of soundness between the two underlying formalizations must compare the LTL mapping provided in [186, 146] with the CLIMB translation introduced in Section 5.4. The first step is to consider the formalizations of an arbitrary ConDec model as a whole, which is formalized:

- in LTL, as a conjunction of the formulae representing each single constraint;
- in CLIMB, as a specification containing the union of ICs representing each single constraint.

The following Theorem states that compliance is preserved for the model as a whole, iff it is preserved for each single constraint. Therefore, soundness can be proven by comparing the formalizations of each single constraint in isolation.

THEOREM 5.1 (Compliance preservation w.r.t. whole/parts). *Given an LTL conjunction formula* $\psi = \bigwedge_i \phi_i$ *and a CLIMB specification* $\mathcal{S} = \langle \emptyset, \bigcup_i \mathtt{IC}_i \rangle$*:*

$$\left(\forall\, i,\ \phi_i \overset{c}{\leftrightsquigarrow} \langle \emptyset, \{\mathtt{IC}_i\}\rangle\right) \Longrightarrow \psi \overset{c}{\leftrightsquigarrow} \mathcal{S}$$

Proof. By considering the semantics of LTL w.r.t. conjunction, and remembering that CLIMB specifications are compositional, we have:

$$\begin{aligned}
(\forall i,\ \phi_i \overset{c}{\leftrightsquigarrow} \langle \emptyset, \{\mathtt{IC}_i\}\rangle) \Leftrightarrow & \quad \text{(Def. 5.7)}\\
\left(\forall \mathcal{T}_{\mathcal{L}}\ \forall i,\ \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \phi_i \Leftrightarrow \textsc{compliant}\left(\langle \emptyset, \{\mathtt{IC}_i\}\rangle_{\mathtt{tm}[\mathcal{T}_{\mathcal{L}}]}\right)\right) \Rightarrow & \\
\left(\forall \mathcal{T}_{\mathcal{L}}, \bigwedge_i \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \phi_i \Leftrightarrow \bigwedge_i \textsc{compliant}\left(\langle \emptyset, \{\mathtt{IC}_i\}\rangle_{\mathtt{tm}[\mathcal{T}_{\mathcal{L}}]}\right)\right) \Leftrightarrow & \quad \text{(Sec. 3.6.3)}\\
\left(\forall \mathcal{T}_{\mathcal{L}},\ \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \psi \Leftrightarrow \bigwedge_i \textsc{compliant}\left(\langle \emptyset, \{\mathtt{IC}_i\}\rangle_{\mathtt{tm}[\mathcal{T}_{\mathcal{L}}]}\right)\right) \Leftrightarrow & \quad \text{(Cor. 4.1)}\\
\left(\forall \mathcal{T}_{\mathcal{L}},\ \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \psi \Leftrightarrow \textsc{compliant}\left(\mathcal{S}_{\mathtt{tm}[\mathcal{T}_{\mathcal{L}}]}\right)\right) \Leftrightarrow & \quad \text{(Def. 5.7)}\\
\psi \overset{c}{\leftrightsquigarrow} \mathcal{S} & \quad \square
\end{aligned}$$

We do not report here the proof of soundness for each single ConDec constraint, but limit our discussion to some illustrative examples.

LEMMA 5.1 (Compliance preservation for the `absence` constraint). *It holds that*

$$t_{LTL}\left(\boxed{\mathsf{a}}^{\,0}\right) \overset{c}{\leftrightsquigarrow} t_{CLIMB}\left(\boxed{\mathsf{a}}^{\,0}\right), \textit{ i.e.}$$

$$\square \neg a \overset{c}{\leftrightsquigarrow} \mathcal{S} = \langle \emptyset, \{true \rightarrow \mathbf{EN}(exec\,(a)\,, T).\}\rangle$$

Proof. Let us prove one side of the equivalence ($\overset{c}{\rightsquigarrow}$); the other side can be proven in a very similar way. To disprove $\overset{c}{\rightsquigarrow}$, one must find an LTL execution trace which is compliant with $\square \neg a$, but whose corresponding CLIMB trace is not compliant with $\mathcal{S}$. To violate $\mathcal{S}$, a CLIMB execution trace must contain the execution of a at a certain time, say, t. Hence, $\mathcal{T}_{neg} = \{\mathbf{H}(exec\,(a)\,, t)\}$ violates $\mathcal{S}$. The corresponding LTL trace $\mathcal{T}_{\mathcal{L}} = \mathtt{tm}^{-1}(\mathcal{T}_{neg})$, has the property that $a \in \mathcal{T}_{\mathcal{L}}(t)$, but this contradicts the LTL formalization, whose semantics states that $\nexists\, i$ s.t. $a \in \mathcal{T}_{\mathcal{L}}(i)$. $\square$

LEMMA 5.2 (Compliance preservation for the `response` constraint). *It holds that*

$$t_{LTL}\left(\boxed{\mathsf{a}} \bullet\!\!\longrightarrow \boxed{\mathsf{b}}\right) \overset{c}{\leftrightsquigarrow} t_{CLIMB}\left(\boxed{\mathsf{a}} \bullet\!\!\longrightarrow \boxed{\mathsf{b}}\right), \textit{ i.e.}$$

$$\square(a \Rightarrow \Diamond b) \overset{c}{\leftrightsquigarrow} \mathcal{S} = \langle \emptyset, \{ \begin{array}{c} \mathbf{H}(exec\,(a)\,, T_a) \\ \rightarrow \mathbf{E}(exec\,(b)\,, T_b) \wedge T_b > T_a. \end{array} \}\rangle$$

Proof. Let us prove one side of the equivalence ($\overset{c}{\rightsquigarrow}$); the other side can be proven in a very similar way. To disprove $\overset{c}{\rightsquigarrow}$, one must find an LTL

execution trace which is compliant with $\Box\neg a$, but whose corresponding CLIMB trace is not compliant with $\mathcal{S}$. To violate $\mathcal{S}$, a CLIMB execution trace must contain the execution of a at a certain time, say, t, s.t. after t activity b is never executed. By applying the $\mathtt{tm}^{-1}$ function on this trace, one obtains an LTL trace $\mathcal{T}_{\mathcal{L}}$ which obeys to the following properties:

A. $a \in \mathcal{T}_{\mathcal{L}}(t)$

B. $\forall t' > t,\ b \notin \mathcal{T}_{\mathcal{L}}(t')$

By combining the first property with the LTL formula $\Box(a \Rightarrow \Diamond b)$, we obtain $(\mathcal{T}_{\mathcal{L}},\ t \models_{\mathcal{L}} \Diamond b)$, and therefore it must hold that $\exists t' > t$ s.t. $b \in \mathcal{T}_{\mathcal{L}}(t')$. However, this contradicts the second property. □

LEMMA 5.3 (Compliance preservation for the `chain response` constraint). *It holds that*

$$t_{LTL}\left(\boxed{a}\bullet\!\!\Longrightarrow\!\!\blacktriangleright\boxed{b}\right) \overset{c}{\leftrightsquigarrow} t_{CLIMB}\left(\boxed{a}\bullet\!\!\Longrightarrow\!\!\blacktriangleright\boxed{b}\right),\ \textit{i.e.}$$

$$\Box(a \Rightarrow \bigcirc b) \overset{c}{\leftrightsquigarrow} \mathcal{S} = \langle \emptyset, \{ \begin{array}{l} \mathbf{H}(exec(a), T_a) \\ \rightarrow \mathbf{E}(exec(b), T_b) \\ \wedge T_b == T_a + 1. \end{array} \} \rangle$$

Proof. Let us prove one side of the equivalence ($\overset{c}{\rightsquigarrow}$); the other side can be proven in a very similar way. To disprove $\overset{c}{\rightsquigarrow}$, one must find an LTL execution trace which is compliant with $\Box\neg a$, but whose corresponding CLIMB trace is not compliant with $\mathcal{S}$. To violate $\mathcal{S}$, a CLIMB execution trace must contain the execution of a at a certain time, say, t, s.t. at time $t+1$ activity b is not executed. By applying the $\mathtt{tm}^{-1}$ function on this trace, one obtains an LTL trace $\mathcal{T}_{\mathcal{L}}$ which obeys to the following properties:

A. $a \in \mathcal{T}_{\mathcal{L}}(t)$

B. $b \notin \mathcal{T}_{\mathcal{L}}(t+1)$

By combining the first property with the LTL formula $\Box(a \Rightarrow \bigcirc b)$, we obtain $(\mathcal{T}_{\mathcal{L}},\ t \models_{\mathcal{L}} \bigcirc b)$, and therefore it must hold that $b \in \mathcal{T}_{\mathcal{L}}(t+1)$. However, this contradicts the second property. □

5.6 ON THE EXPRESSIVENESS OF SCIFF

Having shown that all the ConDec constraints can be translate not only to LTL, but also on CLIMB (i.e., on SCIFF), and that compliance is preserved when switching from one formalization to the other, a more general question arises: what is the relationship between LTL and SCIFF? Is it possible to express an arbitrary LTL formula in SCIFF?

We discuss this issue demonstrating that the answer is "yes". Note that the opposite question (is it possible to express an arbitrary SCIFF

specification in LTL?) has a trivial negative answer, because LTL is propositional whereas SCIFF belongs to the first-order setting. E.g., quantitative (metric) temporal constraints can be expressed in SCIFF by imposing CLP constraints on time variables, while LTL does not support this feature.

One**Theoretical vs practical issues!** point deserves a close examination here: the fact that each LTL formula can be suitably translated to SCIFF is investigated at a pure theoretical level, and does not imply that reasoning on LTL formulae can be completely carried out by reasoning on the corresponding SCIFF specifications. As we will see in Chapter 9, the proof procedures able to perform a-priori and run-time/a-posteriori reasoning on SCIFF specifications impose syntactic restrictions which rule out many specifications used to represent LTL formulae.

5.6.1 *A Separated Normal Form for LTL Formulae*

In [76], the authors introduce a Separated Normal Form (SNF) capable of expressing an arbitrary LTL formula by adopting a conjunction of three-basic forms, while preserving satisfiability. The translation of an arbitrary LTL formula to SNF will take part of the proof aimed at demonstrating that SCIFF is able to express LTL. Therefore, we briefly recall here the main Definitions and results given in [76], referring to [76] for an exhaustive description.

Definition 5.8 (SNF Formula [76]). An LTL formula ϕ is in *Separated Normal Form* iff ϕ is a conjunction of the following forms:

$$\mathbf{start} \implies \bigwedge_c l_c \qquad \text{(an initial LTL-clause)}$$

$$\Box\Big(\bigwedge_a k_a \implies \bigcirc \bigvee_d l_d\Big) \qquad \text{(a step LTL-clause)}$$

$$\Box\Big(\bigwedge_b k_b \implies \Diamond l\Big) \qquad \text{(a sometime LTL-clause)}$$

where k_i and l_j are literals (i.e., atomic propositions or negation of atomic propositions) and **start** is a special symbol true only at the initial time (i.e., whose valuation function is the set $\{0\}$).
In this case, we say that ϕ is an SNF formula.

Definition 5.9 (LTL to SNF translation [76]). `snf` is a function which translates an arbitrary LTL formula to a corresponding SNF formula. The transformation rules are given in [76][3].

During the transformation, new proposition symbols are introduced: they do not represent any event, but are used to rename complex subformulae. We differentiate, among the whole set of proposition symbols, the sub-set used to represent activities/events, and the sub-set used for renaming.

3 Note that in [76] the `snf` function is called τ.

DEFINITION 5.10 (Formula proposition symbols). Given an LTL formula ϕ, $\mathcal{P}(\phi)$ is the set of proposition symbols contained in ϕ.

DEFINITION 5.11 (Renaming and event sets). Given an LTL formula ϕ and an SNF formula σ s.t. $\sigma = \mathtt{snf}(\phi)$, it holds that $\mathcal{P}(\sigma) = \mathcal{E}(\sigma) \cup \mathcal{R}(\sigma)$, where:

A. *event set* $\mathcal{E}(\sigma)$ is the set of atomic propositions contained in the original LTL formula ϕ, which denote "real" events ($\mathcal{E}(\sigma) = \mathcal{P}(\phi)$)

B. *renaming set* $\mathcal{R}(\sigma)$ is the set of atomic propositions used for renaming during the transformation.

EXAMPLE 5.4 (SNF representation of the ConDec `response` constraint). *Let us consider the* `response` *ConDec constraint stating that each excution of the* pay *activity must be eventually followed by the* send_receipt *one:*

pay •——▶ send_receipt

Its corresponding LTL formalization is:

$$\phi = \square(pay \Rightarrow \Diamond send_receipt)$$

Hence, we have $\mathcal{P}(\phi) = \{pay, send_receipt\}$*. According to Definition 5.9,* ϕ *is a sometime LTL-clause, and it is therefore already in SNF. Thus, we have:* $\mathtt{snf}(\phi) = \phi$ *and* $\mathcal{R}(\mathtt{snf}(\phi)) = \emptyset$*.*

EXAMPLE 5.5 (SNF representation of the ConDec `precedence` constraint). *Let us consider the* `precedence` *ConDec constraint, stating that the* send_receipt *activity can be executed only after having executed the* pay *activity:*

pay ——▶• send_receipt

Its corresponding LTL formalization is:

$$\phi = \neg send_receipt \mathcal{W} pay$$

Hence, $\mathcal{P}(\phi) = \{pay, send_receipt\}$*. The SNF translation of* ϕ *is:*

$$\begin{aligned}
\sigma = \mathtt{snf}\,&[\neg send_receipt \mathcal{W} pay] = \\
&= \mathbf{start} \Rightarrow \mathbf{x} \wedge \\
&\quad \mathtt{snf}\,[\mathbf{x} \Rightarrow \neg send_receipt \mathcal{W} pay] = \\
&= \mathbf{start} \Rightarrow \mathbf{x} \wedge \\
&\quad \mathbf{x} \Rightarrow (\neg send_receipt \vee pay) \wedge \\
&\quad \mathbf{x} \Rightarrow (\mathbf{y} \vee pay) \wedge \\
&\quad \mathbf{y} \Rightarrow \bigcirc(\neg send_receipt \vee pay) \wedge \\
&\quad \mathbf{y} \Rightarrow \bigcirc(\mathbf{y} \vee pay)
\end{aligned}$$

Therefore, $\mathcal{R}(\sigma) = \{\mathbf{start}, \mathbf{x}, \mathbf{y}\}$*.*

5.6.2 *Translation of SNF Formulae to SCIFF-lite*

We provide a syntactic procedure to translate an arbitrary SNF formula to SCIFF-lite, and prove that such a translation preserves compliance.

The target language is SCIFF-lite and not CLIMB, because the abducible set of the obtained specifications is not limited to expectations, but contains other predicates as well.

DEFINITION 5.12 ($\mathcal{IC}$-mapping). An $\mathcal{IC}$-mapping icm is a function which translates an SNF formula to a set of SCIFF-lite ICs. It is defined as follows:

$$\mathtt{icm}\left[\bigwedge_i \phi_i\right] \triangleq \bigcup_i \mathtt{icm}\left[\phi_i\right]$$

$$\mathtt{icm}\left[\textbf{start} \implies \bigwedge_c l_c\right] \triangleq \mathtt{icm}\left[start, 0\right] \rightarrow \bigwedge_c \mathtt{icm}\left[l_c, 0\right].$$

$$\mathtt{icm}\left[\Box\left(\bigwedge_a k_a \implies \bigcirc \bigvee_d l_d\right)\right] \triangleq \bigwedge_a \mathtt{icm}\left[k_a, T\right] \rightarrow \bigvee_d \left(\mathtt{icm}\left[l_d, T_2\right] \wedge T_2 == T + 1\right).$$

$$\mathtt{icm}\left[\Box\left(\bigwedge_a k_a \implies \Diamond l\right)\right] \triangleq \bigwedge_a \mathtt{icm}\left[k_a, T\right] \rightarrow \mathtt{icm}\left[l, T_2\right] \wedge T_2 \geqslant T.$$

$$\mathtt{icm}\left[true, T\right] \triangleq \textbf{true}(T)$$

$$\mathtt{icm}\left[a, T\right] \triangleq \textbf{occ}(a, T)$$

$$\mathtt{icm}\left[\neg a, T\right] \triangleq \textbf{not}(a, T)$$

DEFINITION 5.13 ($\mathcal{S}$-mapping). Given an SNF formula ϕ and a set $\mathcal{S}$ of proposition symbols s.t. $\mathcal{S} \subseteq \mathcal{P}(\phi)$, the *$\mathcal{S}$-mapping* sm translates ϕ to a SCIFF-lite specification depending on $\mathcal{S}$. sm is defined as follows:

$$\mathtt{sm}: \quad \phi, \mathcal{S} \longmapsto \langle \emptyset, \{\mathbf{E}/2, \mathbf{EN}/2, \textbf{start}/1, \textbf{true}/1, \textbf{occ}/2, \textbf{not}/2\}, \mathcal{IC} \rangle$$

where

$$\begin{aligned}
\mathcal{IC} = \mathtt{icm}(\phi) \cup \{\ & true \rightarrow \textbf{occ}(start, 0). && \text{(S)}\\
& true \rightarrow \textbf{true}(0). && (\mathrm{T}_1)\\
& \textbf{true}(T) \rightarrow \textbf{true}(T_2) \wedge T_2 == T + 1. && (\mathrm{T}_2)\\
& \forall p \in \mathcal{P}(\phi), \textbf{true}(T) \rightarrow \textbf{occ}(p, T) \vee \textbf{not}(p, T). && \text{(2V)}\\
& \textbf{occ}(X, T) \wedge \textbf{not}(X, T) \rightarrow \bot. && \text{(C)}\\
& \mathbf{H}(X, T) \wedge X \in \mathcal{S} \rightarrow \textbf{occ}(X, T). && \text{(O)}\\
& \textbf{occ}(X, T) \wedge X :: \mathcal{S} \rightarrow \mathbf{E}(X, T). && (\mathrm{E}_1)\\
& \textbf{not}(X, T) \wedge X :: \mathcal{S} \rightarrow \mathbf{EN}(X, T).\ \} && (\mathrm{E}_2)
\end{aligned}$$

$\mathcal{S}$-mapping works as follows. First of all, the presence of a certain proposition in a given state is mapped to an abducible stating that the proposition *occurs* in that state. Conversely, the absence of the proposition is mapped to an abducible stating that the proposition does *not* occur in that state. A further abducible is used to model the concept of truth in LTL, which is implicitly subject to the formula $\Box true$.

- IC (S) translates the special **start** symbol, which is introduced by SNF and is true at and only at the initial state (i.e., at time point 0).
- ICs (T_1) and (T_2) formalizes that the **true** abducible holds in all the states, i.e., at all time points.
- ICs (2V) and (C) are used to model the two-valued semantics of LTL: in LTL, in each state either a proposition is true (ex-)or false.
- ICs (O), (E_1) and (E_2) relate the (not) occurrence of each proposition in each state with the SCIFF concepts of happened events and positive (negative) expectations.

The role of the set $\mathcal{S}$, used in the last three ICs, will be discussed later. At the moment, we consider $\mathcal{S}$ to be the same set of all proposition symbols used in the SNF-formula.

The following Theorem proves that the sm function preserves compliance, hence an arbitrary SNF formula is translatable to a "behaviourally equivalent" SCIFF-lite specification.

THEOREM 5.2 (SCIFF-lite can express SNF formulae). *Given an SNF formula σ and the SCIFF-lite specification $\mathcal{S} = \mathtt{sm}[\sigma, \mathcal{P}(\sigma)]$, it holds that $\sigma \overset{c}{\leftrightsquigarrow} \mathcal{S}$.*

Proof. First of all, it is worth noting that LTL and SCIFF share the same semantics for basic logical connectives: $\wedge, \vee, \Rightarrow / \rightarrow$. We will therefore focus only on the simplest SNF-forms, composed by single proposition symbols instead of conjunctions/disjunctions of them. We consider each basic SNF-form separately.

Let σ be a simple initial LTL-clause, i.e., $\sigma = \mathbf{start} \Rightarrow l$. If l is a positive literal, say, $l = a$, each compliant LTL execution trace $\mathcal{T}_{\mathcal{L}}$ must satisfy the property that $a \in \mathcal{T}_{\mathcal{L}}(0)$, because **start** always holds in state 0. The obtained $\mathcal{S}$ contains the corresponding IC $\mathtt{icm}[\mathbf{start} \Rightarrow a] = \mathbf{occ}(\mathtt{start}, 0) \rightarrow \mathbf{occ}(a, 0)$. By taking into account also the two general ICs (S) and (E_1), all abductive explanations of $\mathcal{S}$ must expect a at time point 0, i.e., they must contain $\mathbf{E}(a, 0)$. Therefore, each compliant trace $\mathcal{T}$ must contain $\mathbf{H}(a, 0)$. By considering the trace mapping function tm, this is exactly the same property required for compliant LTL traces, and therefore compliance is preserved by switching from σ to $\mathcal{S}$ or vice-versa. The case in which l is a negative literal, say, $l = \neg a$, can be proven in a similar way: each compliant LTL trace $\mathcal{T}_{\mathcal{L}}$ must satisfy the property that $a \notin \mathcal{T}_{\mathcal{L}}(0)$, each compliant SCIFF trace $\mathcal{T}$ must satisfy the property that $\mathbf{H}(a, 0) \notin \mathcal{T}$, and the two properties are equivalent.

Let σ be a simple step LTL-clause, i.e., $\sigma = k \implies \bigcirc l$. If both k and l are positive literals, the formula correspond to the LTL formalization of the ConDec `chain response` constraint. The obtained IC, together with the general rules (O) and (E_1), is equivalent to the translation of `chain response` to CLIMB, and therefore the proof is given in Lemma 5.3. The case in which k is positive and l is negative can be proven in the same way (in fact, it resembles the proof of compliance preservation between the two representations of the ConDec `negation`

`chain response` constraint). Let us now consider the case in which k is a negative literal, say $k = \neg a$, and l is a positive literal, say $l = b$ (the case in which l is a negative literal can be proven in the same way). Each LTL compliant trace must obey to the following property: $\forall\ t,\ a \in \mathcal{T}_{\mathcal{L}}(t) \vee b \in \mathcal{T}_{\mathcal{L}}(t+1)$. The IC obtained by the application of `icm` is $\mathbf{not}(a, T) \rightarrow \mathbf{occ}(b, T_2) \wedge T_2 == T+1$. For each time t, if a happens at time t then rule (O) states that $\mathbf{occ}(a, t)$ is abduced, rule (C) prevents $\mathbf{not}(a, t)$ to be abduced and thus the IC does not trigger. If, conversely, a does not happen at time t, then rule (2V) leads to abduce $\mathbf{not}(a, t)$; the IC triggers, abducing $\mathbf{occ}(b, t+1)$, which in turn triggers (E_1), imposing that b is expected to happen at time $t+1$. Therefore, each SCIFF compliant execution trace $\mathcal{T}$ must satisfies that $\forall\ t, \mathbf{H}(a, t) \in \mathcal{T} \vee \mathbf{H}(b, t+1) \in \mathcal{T}$, which is in equivalent, under `tm`, with the property on LTL traces.

The case of a simple sometime LTL-clause trivially follows from the discussion made for the step LTL-clause.

Having proven that `sm` preserves compliance for each SNF basic form, we must prove that the translation preserves compliance when applied to a conjunction of these forms. As shown in Definition 5.12, the translation of $\bigwedge_i \phi_i$ is the union of the translation applied to each ϕ_i. Since the translation of each ϕ_i preserves compliance, Theorem 5.1 guarantees that compliance is preserved translating the whole conjunction as well. $\square$

5.6.3 *Translation of Arbitrary LTL Formulae to SCIFF-lite*

By exploiting the one of the main result presented in [76], and the fact that `sm` is able to represent all the SNF formulae in SCIFF-lite, we now demonstrate that an arbitrary LTL formula can be represented in SCIFF-lite as well. The central technical problem that must be still tackled is the fact that the SNF translation introduces new symbols (used for renaming complex sub-formulae) which do not represent events. At the SNF level, the distinction between concrete events and renaming symbols get lost, and therefore the SCIFF-lite specification produced by applying in cascade the SNF and the `sm` translation does not preserve compliance w.r.t. the original LTL formula: positive expectations are imposed also on renaming symbols, which however do not appear in the original LTL formula, leading to a mismatch between the two notions of compliance.

EXAMPLE 5.6 (Compliance mismatch). *Let us consider the ConDec* [1..∗ a] *constraint. Its LTL representation is $\phi = \Diamond a$. The SNF translation of ϕ is $\sigma = \mathtt{snf}(\Diamond a) = \square(\mathbf{start} \Rightarrow \mathbf{y}) \wedge \square(\mathbf{y} \Rightarrow \Diamond a)$. The LTL execution trace containing only the execution of a in state 0 is compliant with ϕ, but it is instead not compliant with σ, which requires also the presence of $\mathbf{y}$ in state 0.*

To overcome this issue, the intuitive idea is to propose a translation that does not impose expectations on renaming symbols, i.e., restricts

the set $\mathcal{S}$ involved in the definition of the `sm` function only to events. The (compliant) execution traces considered by the SNF translation and the corresponding SCIFF-lite representation extend the execution traces compliant with the original LTL formula with symbols taken from the renaming set. The first step is therefore to define, in both settings, a suitable trace projection, which filters an execution trace by maintaing only certain symbols (in particular, the ones which correspond to events), ruling out the other ones.

DEFINITION 5.14 (SCIFF trace projection). Given a SCIFF execution trace $\mathcal{T}$ and a set $\mathcal{S}$ of function symbols, the *trace projection* of $\mathcal{T}$ on $\mathcal{S}$ ($\mathcal{T}|_{\mathcal{S}}$) is the projection of $\mathcal{T}$ containing only events taken from $\mathcal{S}$:

$$\mathcal{T}|_{\mathcal{S}} \triangleq \{\mathbf{H}(e,t) \mid \mathbf{H}(e,t) \in \mathcal{T} \wedge e \in \mathcal{S}\}$$

DEFINITION 5.15 (LTL trace projection). Given an LTL execution trace $\mathcal{T}_{\mathcal{L}} = (\mathbb{N}, <, \nu_{occ})$ and a set $\mathcal{S}$ of proposition symbols, the *trace projection* of $\mathcal{T}_{\mathcal{L}}$ on $\mathcal{S}$ ($\mathcal{T}_{\mathcal{L}}|_{\mathcal{S}}$) is the projection of $\mathcal{T}_{\mathcal{L}}$ containing only events taken from $\mathcal{S}$:

$$\mathcal{T}_{\mathcal{L}}|_{\mathcal{S}} = (\mathbb{N}, <, \nu_{occ}') \text{ s.t. } \nu_{occ}'(e) \triangleq \begin{cases} \nu_{occ}(e) & \text{if } e \in \mathcal{S}; \\ \emptyset & \text{otherwise.} \end{cases}$$

LEMMA 5.4 (Commutativity between trace projection and trace mapping). *For each LTL execution trace $\mathcal{T}_{\mathcal{L}}$ and for each set of proposition symbols $\mathcal{S}$, it holds that*

$$\mathtt{tm}[\mathcal{T}_{\mathcal{L}}|_{\mathcal{S}}] = \mathtt{tm}[\mathcal{T}_{\mathcal{L}}]|_{\mathcal{S}}$$

Proof. Trivial from the definitions of trace mapping (Definition 5.6) and of trace projection (Definitions 5.14 and 5.15). Let us first consider an element e belonging to $\mathcal{S}$. On the left side, the valuation function of e is maintained by the LTL projection and then subject to the `tm` mapping; on the right side, the valuation function of e is subject to the `tm` mapping, and the obtained result is maintained by the SCIFF projection. Let us now consider an element e' outside $\mathcal{S}$. On the left side, the valuation function of e' is "nullified" by the LTL projection, and therefore no happened event concerning e' is produced by `tm`; on the right side, a set of happened events concerning e' is produced by `tm`, but they are then ruled out by the SCIFF projection. □

We now briefly recall one of the main result presented in [76], which proves that SNF preserves satisfiability, which in our setting means that it preserves compliance. Lemma 5.5 reviews the satisfiability result by explicitly taking into account execution traces. In particular, it states that execution traces compliant respectively with an LTL formula and its corresponding SNF are exactly the same if we limit the comparison only on concrete events.

THEOREM 5.3 (SNF preserves satisfiability [76]). *An LTL formula ϕ is satisfiable iff* $\mathtt{snf}(\phi)$ *is satisfiable.*

LEMMA 5.5 (Compliance preservation via extended traces[76]). *For each LTL formula* ϕ*, it holds that*

$$\forall \mathcal{T}_{\mathcal{L}},\ \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \mathtt{snf}[\phi] \Longrightarrow \mathcal{T}_{\mathcal{L}}|_{\mathcal{E}(\mathtt{snf}[\phi])} \models_{\mathcal{L}} \phi$$
$$\forall \mathcal{T}_{\mathcal{L}},\ \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \phi \Longrightarrow \exists \mathcal{T}'_{\mathcal{L}}\ \mathcal{T}_{\mathcal{L}} = \mathcal{T}'_{\mathcal{L}}|_{\mathcal{E}(\mathtt{snf}[\phi])} \wedge \mathcal{T}'_{\mathcal{L}} \models_{\mathcal{L}} \mathtt{snf}[\phi]$$

We are now ready to prove that each LTL formula is translatable to a SCIFF-lite specification, preserving compliance.

THEOREM 5.4 (SCIFF can express LTL). *Given an arbitrary LTL formula* ϕ *and the SCIFF specification* $\mathcal{S}^{\mathcal{E}} = \mathtt{sm}[\mathtt{snf}[\phi], \mathcal{P}(\phi)]$*, it holds that* $\mathcal{S}^{\mathcal{E}} \overset{c}{\leftrightsquigarrow} \phi$.

Proof. Let us denote $\sigma = \mathtt{snf}[\phi]$. From Definition 5.7, and by remembering that the event set of σ contains all the proposition symbols of ϕ ($\mathcal{P}(\phi) = \mathcal{E}(\sigma)$), one has to prove that

$$\forall \mathcal{T}_{\mathcal{L}},\ \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \phi \Longleftrightarrow \textsc{compliant}\left(\mathtt{sm}[\sigma, \mathcal{E}(\sigma)]_{\mathtt{tm}[\mathcal{T}_{\mathcal{L}}]}\right)$$

$\Longrightarrow$
Let us consider the following schema:

$$\begin{array}{ccc}
\forall \mathcal{T}_{\mathcal{L}},\ \mathcal{T}_{\mathcal{L}} \models_{\mathcal{L}} \phi & \xRightarrow{(*)} & \textsc{compliant}\left(\mathtt{sm}[\sigma, \mathcal{E}(\sigma)]_{\mathtt{tm}[\mathcal{T}_{\mathcal{L}}]}\right) \\
\Big\Downarrow \text{Lemma 5.5} & & \Big\Uparrow (\dagger) \\
\exists \mathcal{T}'_{\mathcal{L}},\ \mathcal{T}'_{\mathcal{L}} \models_{\mathcal{L}} \sigma \\ \wedge \mathcal{T}_{\mathcal{L}} = \mathcal{T}'_{\mathcal{L}}|_{\mathcal{E}(\sigma)} & \xRightarrow{\text{Theorem 5.2}} & \textsc{compliant}\left(\mathtt{sm}[\sigma, \mathcal{P}(\sigma)]_{\mathtt{tm}[\mathcal{T}'_{\mathcal{L}}]}\right)
\end{array}$$

The schema shows that proving $(*)$ reduces to prove that

$$\textsc{compliant}\left(\mathtt{sm}[\sigma, \mathcal{P}(\sigma)]_{\mathtt{tm}[\mathcal{T}'_{\mathcal{L}}]}\right) \Longrightarrow \textsc{compliant}\left(\mathtt{sm}[\sigma, \mathcal{E}(\sigma)]_{\mathtt{tm}[\mathcal{T}'_{\mathcal{L}}|_{\mathcal{E}(\sigma)}]}\right) \quad (\dagger)$$

By taking into account abducible sets, Definition 5.11 and Lemma 5.4, $(\dagger)$ becomes:

$$\textsc{compliant}_{\Delta}\left(\mathcal{S}^{\mathcal{ER}}_{\mathcal{T}}\right) \Longrightarrow \textsc{compliant}_{\Delta'}\left(\mathcal{S}^{\mathcal{E}}_{\mathcal{T}|_{\mathcal{E}(\sigma)}}\right) \quad (\ddagger)$$

where $\mathcal{S}^{\mathcal{ER}} = \mathtt{sm}[\sigma, \mathcal{E}(\sigma) \cup \mathcal{R}(\sigma)]$, $\mathcal{S}^{\mathcal{E}} = \mathtt{sm}[\sigma, \mathcal{E}(\sigma)]$ and $\mathcal{T} = \mathtt{tm}\left[\mathcal{T}'_{\mathcal{L}}\right]$.
To prove $(\ddagger)$, we demonstrate that

$$\Delta' = \Delta / \{\mathbf{E}(e,t) | e \in \mathcal{R}(\sigma)\} / \{\mathbf{EN}(e,t) | e \in \mathcal{R}(\sigma)\}$$

obeys the three properties required by the definition of compliance:

A. Δ' is an abductive explanation for $\mathcal{S}^{\mathcal{E}}_{\mathcal{T}|_{\mathcal{E}(\sigma)}}$. The only difference between $\mathcal{S}^{\mathcal{E}}$ and $\mathcal{S}^{\mathcal{ER}}$ is that, for the first specification, rules (O), (E_1) and (E_2) of Definition 5.13 do not trigger for events outside $\mathcal{E}(\sigma)$ (in particular, they do not trigger for events inside $\mathcal{R}(\sigma)$). From

Remark 4.3, Δ is therefore a suitable abductive explanation for $\mathcal{S}^{\mathcal{E}}$ too. Furthermore, being (E_1) and (E_2) the only constraints involving positive and negative expectations concerning elements in $\mathcal{R}(\sigma)$, it is not required for an abductive explanation to contain them anymore.

B. Δ' is **E**-consistent, because $\Delta' \subseteq \Delta$ and Δ is **E**-consistent.

C. Δ' is $\mathcal{T}|_{\mathcal{E}(\sigma)}$-fulfilled. Since $\mathcal{T}|_{\mathcal{E}(\sigma)}$ is a projection of $\mathcal{T}$, $\Delta' \subseteq \Delta$ and Δ is $\mathcal{T}$-fulfilled, no negative expectation in Δ' can be violated by $\mathcal{T}|_{\mathcal{E}(\sigma)}$. Positive expectations concerning elements in $\mathcal{E}(\sigma)$ are maintained in Δ', and so are the corresponding happened events after the trace projection. Positive expectations concerning elements in $\mathcal{R}(\sigma)$ are removed from Δ when obtaining Δ', and therefore the application of the trace projection, which rules out happened events concerning elements in $\mathcal{R}(\sigma)$, does not compromise trace fulfillment.

$\Longleftarrow$
Let us consider the following schema:

$$
\begin{array}{ccc}
\mathtt{tm}^{-1}[\mathcal{T}] \models_{\mathcal{L}} \phi & \xLeftarrow{(**)} & \forall\, \mathcal{T},\ \textsc{compliant}\left(\mathtt{sm}[\sigma, \mathcal{E}(\sigma)]_{\mathcal{T}}\right) \\
\Big\Uparrow \text{Lemmas5.4 and 5.5} & & \Big\Downarrow (\S) \\
\mathtt{tm}^{-1}[\mathcal{T}'] \models_{\mathcal{L}} \sigma & \xLeftarrow{\text{Theorem 5.2}} & \exists\, \mathcal{T}',\ \textsc{compliant}\left(\mathtt{sm}[\sigma, \mathcal{P}(\sigma)]_{\mathcal{T}'}\right) \\
 & & \wedge \mathcal{T} = \mathcal{T}'|_{\mathcal{E}(\sigma)}
\end{array}
$$

The schema shows that proving $(**)$ reduces to prove that

$$
\begin{aligned}
&\forall\, \mathcal{T},\ \textsc{compliant}_{\Delta}\left(\mathcal{S}^{\mathcal{E}}_{\mathcal{T}}\right) \\
&\qquad \Longrightarrow \exists\, \mathcal{T}',\ \mathcal{T} = \mathcal{T}'|_{\mathcal{E}(\sigma)} \wedge \textsc{compliant}_{\Delta'}\left(\mathcal{S}^{\mathcal{E}\mathcal{R}}_{\mathcal{T}'}\right)
\end{aligned} \tag{\S}
$$

where $\mathcal{S}^{\mathcal{E}\mathcal{R}} = \mathtt{sm}[\sigma, \mathcal{E}(\sigma) \cup \mathcal{R}(\sigma)]$ and $\mathcal{S}^{\mathcal{E}} = \mathtt{sm}[\sigma, \mathcal{E}(\sigma)]$.

First of all, it is worth noting that $\mathcal{S}^{\mathcal{E}\mathcal{R}}$ extends $\mathcal{S}^{\mathcal{E}}$ by imposing that rules (O), (E_1) and (E_2) can be also triggered by **occ**/**not** abducibles involving symbols in $\mathcal{R}(\sigma)$, generating a larger set of expectations. Since $\mathcal{T}' \supseteq \mathcal{T}$, an abductive explanation Δ' can be therefore found for $\mathcal{S}^{\mathcal{E}\mathcal{R}}$ by extending Δ with the new generated expectations: $\Delta' = \Delta \cup \Delta^{\mathbf{E}}_{\mathcal{R}} \cup \Delta^{\mathbf{EN}}_{\mathcal{R}}$, where $\Delta^{\mathbf{E}}_{\mathcal{R}}$ and $\Delta^{\mathbf{EN}}_{\mathcal{R}}$ respectively represent the inserted positive and negative expectations.

Δ' is **E**-consistent. Indeed, since $\Delta^{\mathbf{E}}_{\mathcal{R}}$ and $\Delta^{\mathbf{EN}}_{\mathcal{R}}$ contain only expectations generated by rules (E_1) and (E_2), by construction we have:

$$
\begin{aligned}
\forall\, \mathbf{E}(a,t),\ \mathbf{E}(a,t) \in \Delta^{\mathbf{E}}_{\mathcal{R}} &\Rightarrow \mathbf{occ}(a,t) \in \Delta' \\
\forall\, \mathbf{EN}(a,t),\ \mathbf{EN}(a,t) \in \Delta^{\mathbf{EN}}_{\mathcal{R}} &\Rightarrow \mathbf{not}(a,t) \in \Delta'
\end{aligned} \tag{\S\S}
$$

Let us suppose by absurdum that there exist a, t (with $a \in \mathcal{R}(\sigma)$) s.t. $\mathbf{E}(a,t) \in \Delta^{\mathbf{E}}_{\mathcal{R}}$ and $\mathbf{EN}(a,t) \in \Delta^{\mathbf{EN}}_{\mathcal{R}}$. In this case, (§§) would state that

$\mathbf{occ}(a, t) \in \Delta'$ and $\mathbf{not}(a, t) \in \Delta'$. This would violate rule (C), making impossible that Δ' is an abductive explanation.

An execution trace $\mathcal{T}^*$ compliant with $\mathcal{S}^{\mathcal{ER}}$ can be therefore built as follows:

$$\mathcal{T}^* = \mathcal{T} \cup \mathcal{T}^{\mathcal{R}}, \text{ where } \mathbf{H}(a, t) \in \mathcal{T}^{\mathcal{R}} \Leftrightarrow \mathbf{E}(a, t) \in \Delta^{\mathbf{E}}_{\mathcal{R}}$$

Under this choice:

A. Δ' is left untouched by $\mathcal{T}^*$. Indeed, the only impact of $\mathcal{T}^{\mathcal{R}}$ on the ICs of $\mathcal{S}^{\mathcal{ER}}$ is to trigger rule (O), generating corresponding **occ** abducibles. However, from (§§) we know that all this abducibles are already contained in Δ'.

B. Δ' is $\mathcal{T}^*$-fulfilled by construction.

C. $\mathcal{T}^*|_{\mathcal{E}(\sigma)} = \mathcal{T}$, because all the happened events contained in $\mathcal{T}^{\mathcal{R}}$ involve symbols belonging to $\mathcal{R}(\sigma)$, and are therefore ruled out by applying the projection.

□

6

EXTENDING CONDEC

Contents

6.1 Temporal-constrained Relationships **109**
6.1.1 Temporal Contiguity in a Quantitative Setting 110
6.1.2 Quantitative Formalization of Chain Constraints 110
6.1.3 Metric ConDec Constraints 111
6.2 Data-Related Aspects **114**
6.2.1 The MXML Meta-Model 114
6.2.2 The Life Cycle of ConDec Activities 115
6.2.3 An illustrative example 116
6.3 Introducing Data in ConDec^{++} **116**
6.3.1 Representing Non-Atomic Activities in ConDec^{++} 117
6.3.2 Formalizing the Activity Life Cycle 117
6.3.3 Modeling the Submit-Review Example 119
6.3.4 Cross-Flow Constraints 120

In this Chapter, we investigate how the first-order nature of the CLIMB language can be exploited to extend the expressiveness of ConDec along different dimensions, namely:

- quantitative (metric) temporal constraints;
- activity data and data-related conditions;
- support of a non-atomic model of activities.

We call the language extended with such features ConDec^{++}. ConDec^{++} reconciles ConDec with MXML, a well known meta-model proposed by van Dongen and van der Aalst for representing and storing execution traces[199].

6.1 TEMPORAL-CONSTRAINED RELATIONSHIPS

As pointed out in Chapters 3 and 4, ConDec adopts a qualitative characterization of time, while CLIMB opts for a quantitative characterization, modeling execution times explicitly and relying on CLP for expressing metric constraints among them.

As we have deeply discussed in Section 5.3–Page 85, the formalization of ConDec in terms of CLIMB specifications requires to uniform the

characterization of time: CLIMB must accommodate qualitative time, formalizing the concept of temporal contiguity as "difference of a single time unit" and imaging that execution traces are compact, i.e., that their associated time points only reflect the ordering of occurrences, and not the actual times at which they happened.

Anyway, nothing stop us to follow the opposite direction, i.e., to embed a point-based qualitative characterization of time into a quantitative one. The advantage of this choice is that all the expressiveness of CLIMB w.r.t. the temporal dimension can be exploited at the level of ConDec. We investigate this possibility, showing how the formalization of some ConDec constraints must be properly revised, and how the ConDec relation and negation constraints can be extended at the graphical level to accommodate quantitative temporal constraints.

6.1.1 *Temporal Contiguity in a Quantitative Setting*

Let us remind the key aspects related to the embedding of the qualitative notion of temporal contiguity in a quantitative setting, which have been discussed in Section 5.3.

When quantitative time is used in a purely qualitative setting, what matters is not the actual time values at which event occur, but only that the ordering among event occurrences is preserved. Time evolves only in view of event occurrences, and therefore it can be assumed, without loss of generality, that two occurrences differing of a single time unit are contiguous (and vice-versa).

Temporal contiguity in a quantitative setting

In a quantitative setting, instead, time evolves independently from the flow of events. Two contiguous occurrences may then be separated by an arbitrary time gap: what matters, to define their contiguity, is that, between them, no further event has occurred. The absence of a whatever event inside a certain time interval can be expressed in CLIMB as a general negative expectation, matching with any possible event occurrence and ranging inside the targeted time interval.

DEFINITION 6.1 (Temporal contiguity and distance in CLIMB). Given an execution trace $\mathcal{T}$, two happened events $\mathbf{H}(exec(e_1), t_1)$ and $\mathbf{H}(exec(e_2), t_2)$ belonging to $\mathcal{T}$, s.t. $t_2 > t_1$, are *contiguous* iff

$$\mathbf{EN}(exec(X), T_x) \wedge T_x > t_1 \wedge T_x < t_2 \text{ is } \mathcal{T}\text{-fulfilled}$$

Conversely, the two happened events are *distant* iff

$$\mathbf{E}(exec(X), T_x) \wedge T_x > t_1 \wedge T_x < t_2 \text{ is } \mathcal{T}\text{-fulfilled}$$

6.1.2 *Quantitative Formalization of Chain Constraints*

As pointed out in Section 5.3, temporal contiguity affects the formalization of chain constraints. In particular, to reflect the notion of contiguity given in Definition 6.1, [a] •⇛ [b] must be rephrased as "whenever activity a is executed, then a following occurrence of b is expected s.t. no further activity is executed inbetween". Starting from

this description, a new CLIMB formalization of the chain response constraint can be provided in a straightforward way. We introduce a new translation function $t^{\odot}_{CLIMB}$ to represent a revision of the basic translation function t_{CLIMB} in a quantitative setting; as t_{CLIMB} relies on t_{IC} for the translation of ConDec constraints, $t^{\odot}_{CLIMB}$ relies on $t^{\odot}_{IC}$. It is worth noting that, being CLIMB specifications compositional, changing the formalization of a certain constraint does not impact on the meaning of the other ones.

The revised formalization of the chain response constraint is:

$$t^{\odot}_{IC}\left(\boxed{a}\bullet\!\!\equiv\!\!\blacktriangleright\boxed{b}\right) \triangleq \mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b) \wedge T_b > T_a \\ \wedge\, \mathbf{EN}(exec(X), T_x) \\ \wedge\, T_x > T_a \wedge T_x < T_b.$$

Similarly, the chain precedence constraint is reformulated as follows:

$$t^{\odot}_{IC}\left(\boxed{a}\equiv\!\!\blacktriangleright\!\!\bullet\boxed{b}\right) \triangleq \mathbf{H}(exec(b), T_b) \rightarrow \mathbf{E}(exec(a), T_a) \wedge T_a < T_b \\ \wedge\, \mathbf{EN}(exec(X), T_x) \\ \wedge\, T_x > T_a \wedge T_x < T_b.$$

The last constraint that requires a new formulation is the negation chain response one[1]. Here, the notion of distance must be employed, to state that *all* the occurrences of the source activity and the following occurrences of the target activity cannot be contiguous, i.e., that at least one further occurrence must interpose inbetween:

$$t^{\odot}_{IC}\left(\boxed{a}\bullet\!\!\equiv\!\!\|\!\!\equiv\!\!\blacktriangleright\boxed{b}\right) \triangleq \mathbf{H}(exec(a), T_a) \\ \wedge \mathbf{H}(exec(b), T_b) \\ \wedge T_b > T_a \rightarrow \mathbf{E}(exec(X), T_x) \\ \wedge\, T_x > T_a \wedge T_x < T_b.$$

6.1.3 *Metric ConDec Constraints*

Having fully recovered the quantitative nature of CLIMB, it is now possible to extend ConDec relation constraints, as well as the "normal" negation constraints (i.e., negation response and negation precedence) with metric information, e.g., to express *delays* and *deadlines*. Such an information is used to reduce the validity of constraint's target time (or, in the negative case, to delimit the forbidding of the target), by

1 Indeed, negation chain precedence is superfluous, being equivalent to the chain response one, as showin in Table 23 and pointed out in [186].

defining a quantitative lower and/or an upper bound on it. For example, the response constraint could be now modified by stating that "when the source activity occurs, then the target activity is expected to occur afterwards, but *within a maximum timespan*".

Representation of metric constraints

To graphically show these temporal extensions, a possible choice is to annotate the different ConDec constraints with a time interval marked off by two non negative instants (T_{min} and T_{max}) which could be considered both in an exclusive or inclusive manner. As usually, parentheses (...) are used to indicate exclusion and square brackets [...] to indicate inclusion. The interval is treated as relative w.r.t. the time at which the source happens, and is translated backward or forward w.r.t. it depending on the nature of the constraint (i.e., whether it is a response or precedence one).

CLIMB formalization of metric constraints

Such a feature can be seamlessly modeled in CLIMB. For example, the formalization of the metric response is[2]:

$$t_{IC}^{\circ}\left(\boxed{a} \overset{(n,m)}{\bullet\!\!\longrightarrow} \boxed{b}\right) \triangleq \mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b) \wedge T_b > T_a + n \wedge T_b < T_a + m.$$

The constraint expresses that if activity a is executed at time T_a, then a following activity b is expected to be executed with a minimum delay of n time units, and a maximum delay of m time units. These two temporal conditions can be seamlessly expressed by means of CLP constraints: activity b is expected to happen at a time T_b which must be greater than $T_a + n$ and lower than $T_a + m$.

Symmetrically, precedence is extended as follows:

$$t_{IC}^{\circ}\left(\boxed{a} \overset{(n,m)}{\longrightarrow\!\!\bullet} \boxed{b}\right) \triangleq \mathbf{H}(exec(b), T_b) \rightarrow \mathbf{E}(exec(a), T_a) \wedge T_a > T_b - m \wedge T_a < T_b - n.$$

Metric precedence expresses that activity b is executable only within the time window ranging from n to m time units after an occurrence of activity a. In CLIMB, we can express this by stating that if b is executed at time T_b, then a previous execution of activity a is expected to happen at a time T_a, s.t. T_a makes T_b belonging to the desired time window. The membership of T_b to such a time window is expressed by means of the two CLP constraints $T_b > T_a + n$ and $T_b < T_a + m$, which are equivalent to the ones shown in the formalization.

Figure 21 shows how the extended ConDec^{++} is able to combine the new notation with "normal" relation constraints (i.e., responded existence, response and precedence) to cover different metric relations between the involved activities.

Deadlines and latency constraints

A typical use of the metric extension is to model *deadlines*. For ex-

2 We show the case in which bounds are exluded. Inclusion of bounds is modeled by substituting the $<$ and $>$ CLP constraint with $\leqslant$ and $\geqslant$ respectively.

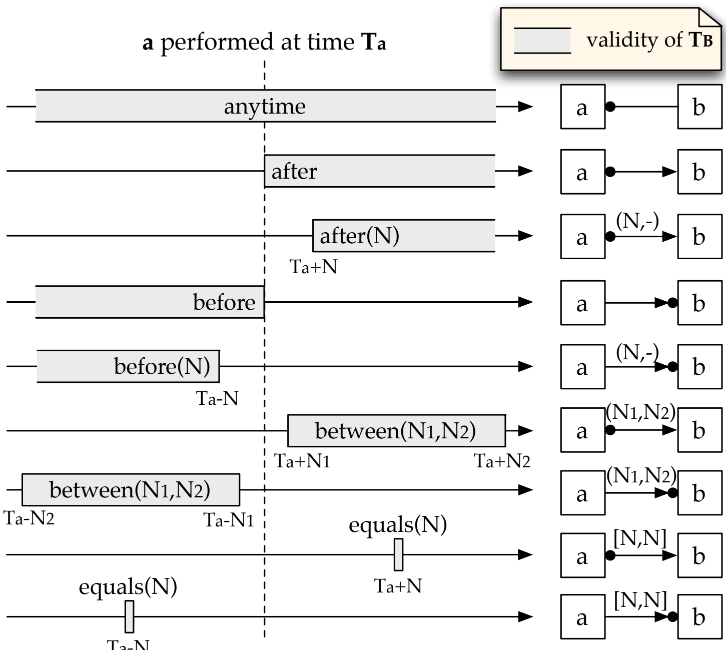
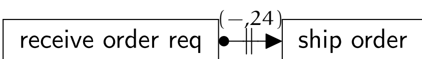


Figure 21: Metric constraints in ConDec^{++}.

ample, a customer could express that she wants to interact only with sellers able to deliver an ordered good *by at most two days* after the payment for the order. By supposing that the time granularity is a hour, ConDec^{++} can express such an extended constraint as:

pay (−,48) delivery

In combination with negation constraints, instead, the metric extension can be used to model *latency* constraints, i.e., constraints stating that at least a minimum amount of time is needed to accomplish a certain operation after having received a request for that operation. For example, a warehouse could state that it takes at least one day to ship a requested order; ConDec^{++} models such a latency constraint as:



Use of quantitative temporal constraints

Quantitative temporal constraints should not be interpreted as a way to *force* the execution of a certain activity within a desired time interval; indeed, interacting entities are autonomous, and cannot be controlled. Instead, temporal constraints should be considered as a mean to specify further requirements on the interacting parties, contributing to the definition of the QoS that must be guaranteed during the interaction: compliant executions must not only respect the modeled constraints, but also satisfy all the temporal requirements. For example, a customer's QoS requirement stating that she wants to obtain an ordered good by at most two days after the order could be used

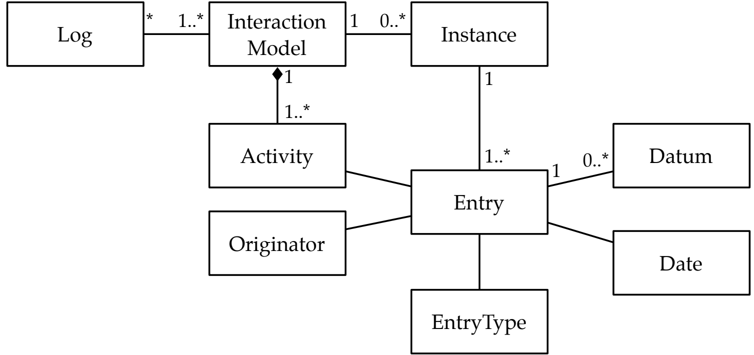


Figure 22: An MXML-like meta model for representing the traces produced by the execution of interaction instances (the original MXML diagram can be found in [199]).

to identify what requirements should be met by a seller in order to consider it as a potential partner. This information could then be used either to statically select good candidates among the possible sellers (e.g., e-sellers whose behavioural interface promises to respect the desired deadline), or to check at run-time if the real behaviour effectively satisfies it. Identifying a violation may be useful in this setting to alert the customer that the seller is breaking its published contract.

6.2 DATA-RELATED ASPECTS

As shown in Chapter 4, CLIMB is not only able to constrain execution times, but it also support any kind of variable inside events. In this way, modeling data related to activities and conditions on that data becomes possible. We briefly introduce the MXML meta-model, and then show how all its features can be seamlessly expressed in CLIMB. This opens the way to the insertion of such advanced features also in the graphical ConDec^{++} notation; we have not yet investigated how the graphical notation could be extended in a user-friendly way, but we will anyway provide some examples of how the resulting framework could look.

6.2.1 *The MXML Meta-Model*

The MXML meta-model has been proposed by van Dongen and van der Aalst as a "universal format" to represent and store execution traces[199]. The meta-model has been originally developed to address the execution traces produced by executing BP models, but it can seamlessly accommodate execution traces produced in many other settings. Figure 22 recall the UML schema of the meta-model, by adapting the name of the different concepts to the case of interaction models.

The MXML meta-model states that a *log* is associated to a set of executed *interaction models*. Interaction models give raise to many different executions, called *instances*. Each instance is described by an execution

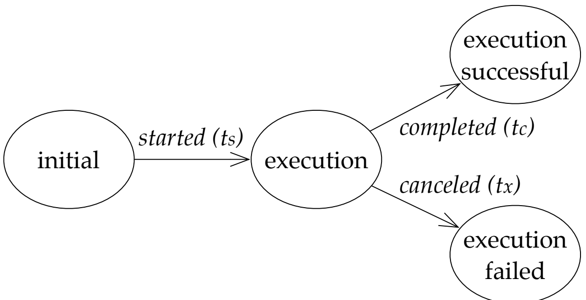


Figure 23: The life cycle of ConDec activities (from [157]).

trace, composed by a set of *entries*. An entry, in turn, models the occurrence of an event during the execution, and is associated to:

- a corresponding *activity* of the interaction model;
- an *originator*, i.e., the interacting entity responsible for the generation of the event;
- an *event type*, reflecting the kind of event which has been generated on the involved activity;
- a *date*, representing the time-stamp at which the event has been generated;
- a (possible empty) set of further specific *data* related to the event.

6.2.2 *The Life Cycle of ConDec Activities*

In MXML, an activity is associated to a corresponding life cycle, which defines the set of operations (i.e., events) that can be executed on the activity, and how the corresponding activity-state is affected. This operations fills the *event type* part of an occurrence inside the MXML meta model, and introduces the possibility of modeling non-atomic activities: the same activity execution spans among different atomic occurrences of the events involved in its life cycle; for example, the execution of an activity could be *started* by someone, then *assigned* to a new originator, and finally *completed* by her.

In [157], a simplified life cycle for the ConDec activities is introduced. We refer to this simplified life cycle (but the complete MXML life cycle could be seamlessly handled as well). The states and transitions of this life cycle are reported in Figure 23. At quiescence, the activity is in the *initial* state, representing that it is not currently in execution. When the t_s event related to the activity occurs, then the activity is *started*, becoming in *execution*. To complete the execution of the activity, two possibilities are supported:

- event t_c occurs, and the activity is successfully completed;
- the activity cannot be completed, and therefore its execution is canceled by means of the t_x event.

6.2.3 *An illustrative example*

Let us consider a simple but illustrative example, to show how the introduction of data in the model, as well as the adoption of a non-atomic model of activities, would enable the possibility of expressing complex constraints.

EXAMPLE 6.1 (Fragment of a Submit&Review process). *Let us consider a fragment of a review process. Such a fragment involves two* roles*: an* author *and a* reviewer*.*

A person playing the role of author has the possibility of submitting a paper. If the paper submission is successfully completed, the author receives a paper identifier, and a review must be provided by a person playing the role of reviewer within 45 days from the submission.

However, the four-eyes principle *must be respected between the execution of the two activities: the same person cannot start a review of her own paper, even if she is entitled to play both the roles of author and reviewer.*

We now discuss how ConDec^{++} can extend ConDec with such new features, maintaing a valid and complete mapping to CLIMB. We then show how this example can be modeled in the new framework.

6.3 INTRODUCING DATA IN CONDEC^{++}

When formalizing the basic ConDec constraints, CLIMB adopts a simple $\mathit{exec}(A)$ term to represent the execution of (atomic) activities. Such a term could be extended with further variables, to reconcile ConDec with the elements of the MXML meta-model.

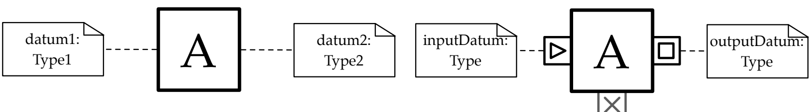
DEFINITION 6.2 (Extended event). An extended CLIMB event is the term

$$\mathit{exec}(E_{type}, E_{ID}, I_{ID}, A, O, DataList)$$

where:

- E_{type} is the event type, taken from the ConDec activity life cycle, i.e., $E_{type} \in \{t_s, t_c, t_x\}$;
- E_{ID} is the univocal identifier representing an occurrence of the event;
- I_{ID} is the instance identifier, used to refer the occurrence of the event to its specific execution context;
- A is the activity the event refers to;
- O is the originator responsible for the occurrence of the event;
- $DataList$ is a list of further data associated to the event.

All these variables can be subject to CLP constraints or Prolog predicates, whose definition must be inserted in the knowledge base of the CLIMB specification.



(a) Atomic activity with data.

(b) Non-atomic activity with start, complete and cancel ports and input/output data.

Figure 24: Atomic and non-atomic activities in ConDec^{++}.

6.3.1 *Representing Non-Atomic Activities in ConDec^{++}*

The ConDec activity life cycle shown in Figure 23 cannot be formalized in LTL, because there is no possibility to *correlate* two separate events, a feature that is needed to put in relationship a t_s event with the corresponding t_c or t_x one. With CLIMB, such a correlation could be easily captured by exploiting the E_{ID} datum introduced in the extended representation of events (Definition 6.2).

Since the activity life cycle is supported by CLIMB, we give the possibility to adopt, in ConDec^{++}, two kind of activities:

- atomic activity, corresponding to the one of ConDec;
- non-atomic activity, characterized by three ports which graphically capture the t_s, t_c and t_x events, i.e., represent the start, completion and cancelation of the activity.

The proposed graphical notation is shown in Figure 24. Atomic activities, as well as the ports of non-atomic ones, can be associated to data using a notation resembling the one of BPMN; for non-atomic activities, data connected to the t_s port represent the input data for that activity, whereas data connected to the t_c port represent output data. We suppose that the completion and cancelation port of an activity have a complete visibility of the data associated to the starting port.

Furthermore, while ConDec^{++} constraints are directly associated to atomic activities, they are associated to the ports of non-atomic ones. In this way, it is possible to state that a certain activity must have at least one successful completion (by attaching a `1..*` cardinality constraint on the completion port), or that a certain activity can be started only if another activity has been successfully completed before (by interconnecting the start and the completion port of the two activities by means of a precedence constraint).

6.3.2 *Formalizing the Activity Life Cycle*

Let us now discuss how the activity life cycle can be modeled as a set of CLIMB ICs. Such constraints can be considered as fixed axioms that must be repeated for each non-atomic activity contained in the model. We therefore introduce the axioms supposing their application on an arbitrary activity a. The axioms center around the idea of *sharing* the

same event identifier among a starting event and a corresponding completion or cancelation; such a correlation supports the identification of a *non-atomic execution* of the activity.

In particular, the first axiom expresses the uniqueness of execution w.r.t. a given event identifier, whereas the next three axioms deal with the legal ordering between t_s and t_c/t_x, as depicted in Figure 23.

AXIOM 6.1 (Uniqueness of the event identifier). *Each event performed on the activity must occur at most once with a given event identifier:*

$$\mathbf{H}(exec(E, E_{ID}, I_{ID}, a, _, _), T_e) \rightarrow \mathbf{EN}(exec(E, E_{ID}, I_{ID}, a, _, _), T_{e2}) \wedge T_{e2} > T_e.$$

where E *is a variable used to intensionally represent either* t_s, t_c *and* t_x. *Since the negative expectation is imposed with the same* E_{ID} *contained in the body, many occurrences of the same event can happen if they have different identifiers.*

AXIOM 6.2 (Completion of a started activity). *When* a *is started by a certain originator, and an identifier* E_{ID} *is associated to the corresponding execution of the activity, then the same originator is committed to complete or cancel the execution of* a *identified by* E_{ID}*:*

$$\begin{aligned}\mathbf{H}(exec(t_s, E_{ID}, I_{ID}, a, O, In), T_s) \\ \rightarrow &\mathbf{E}(exec(t_c, E_{ID}, I_{ID}, a, O, In), T_c) \\ &\wedge T_c > T_s \\ &\vee \mathbf{E}(exec(t_x, E_{ID}, I_{ID}, a, O, [In, Out]), T_x) \\ &\wedge T_x > T_s.\end{aligned}$$

The IC *also achieves the data visibility informally pointed out above: the cancelation event is associated to the input data, while the completion event is associated to the input data as well as to the output data produced by the activity execution.*

AXIOM 6.3 (Cancelation enabling). *An execution of the activity, associated to an event identifier* E_{ID}*, can be canceled by an originator iff the activity has been previously started with identifier* E_{ID} *by the same originator:*

$$\mathbf{H}(exec(t_x, E_{ID}, I_{ID}, a, O, _), T_x) \rightarrow \mathbf{E}(exec(t_s, E_{ID}, I_{ID}, a, O, _), T_s) \wedge T_s < T_x.$$

AXIOM 6.4 (Completion enabling). *An execution of the activity, associated to an event identifier* E_{ID}*, can be successfully completed by an originator iff the activity has been previously started with identifier* E_{ID} *by the same originator:*

$$\mathbf{H}(exec(t_c, E_{ID}, I_{ID}, a, O, _), T_c) \rightarrow \mathbf{E}(exec(t_s, E_{ID}, I_{ID}, a, O, _), T_s) \wedge T_s < T_c.$$

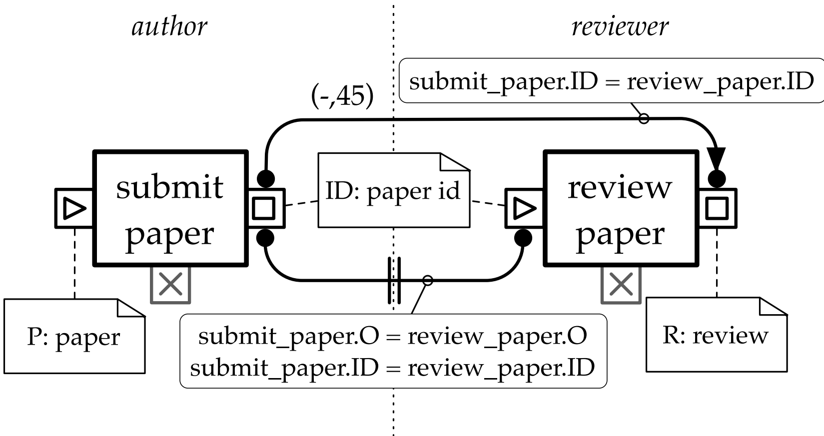


Figure 25: Example of a ConDec++ model.

6.3.3 *Modeling the Submit-Review Example*

Figure 25 shows a possible ConDec++ model representing the Submit&Review process fragment described in Section 6.2.3. When a constraint must involve conditions on data, they are inserted in a rounded box, attached to the constraint.

The following choices have been made to represent the model; we also report how the corresponding CLIMB formalization looks like.

- A non-atomic activity called submit paper is inserted to represent the submission of a paper; the activity takes as input a paper and, if successfully completed, associates an identifier to the paper. The CLIMB formalization will contain the four axioms modeling the activity life cycle, grounded on the submit paper activity.
- A non-atomic activity called review paper is inserted to represent the review of a paper; the activity takes as input the identifier of a paper, and produces a corresponding result. The CLIMB formalization will contain the four axioms modeling the activity life cycle, grounded on the review paper activity.
- An extended succession constraint is used to connect the completion of a submission to the completion of a review, modeling the requirement that a submitted paper must be successfully reviewed. The constraint is extended in two respects. First of all, it must incorporate the desired deadline of 45 days; supposing that the time granularity is fixed to a single day, then such an extension can be simply modeled attaching to the succession constraint the notation $(-, 45)$, following the proposal discussed in Section 6.1.3. Second, the constraint must express that the review must effectively refer to the submitted paper; this can be done by imposing that the paper identifier of the two activities

is the same. By explicitly taking into account the involved roles, the corresponding CLIMB formalization is:

$$\begin{aligned}&\mathbf{H}(exec\,(t_c, E^s_{ID}, I_{ID}, submit_paper, O^s, [ID]), T^s)\\&\wedge role(O^s, author)\\\rightarrow&\mathbf{E}(exec\,(t_c, E^r_{ID}, I_{ID}, review_paper, O^r, [ID, R^r]), T^r)\\&\wedge T^r > T^s \wedge T^r < T^s + 45 \wedge role(O^r, reviewer).\end{aligned} \quad (\bullet\!\!\longrightarrow\!\!\blacktriangleright)$$

$$\begin{aligned}&\mathbf{H}(exec\,(t_c, E^r_{ID}, I_{ID}, review_paper, O^r, [ID, R]), T^r)\\&\wedge role(O^r, reviewer)\\\rightarrow&\mathbf{E}(exec\,(t_c, E^s_{ID}, I_{ID}, submit_paper, O^s, [ID]), T_s)\\&\wedge T^s < T^r \wedge T^s > T^r - 45 \wedge role(O^s, author).\end{aligned} \quad (\longrightarrow\!\!\blacktriangleright\!\!\bullet)$$

Roles are expressed with predicates which affects the possibility of triggering the ICs, restricting the matching originators to the ones which play the requested role. Before an instance of the system begins its execution, a knowledge base must be provided to capture the knowledge which allows to infer who plays a given role[3]. An example of role specification in CLIMB can be found in Example 4.5 – Page 64. Finally, note that equivalence between data (such as the paper identifier) can be expressed by using the same variable in the happened event and expectation; e.g., when someone complete the submission of a paper obtaining 123 as paper identifier, then the expectation about the review is directly generated with a ground identifier equals to 123.

- An extended not coexistence constraint states that the submit paper activity is incompatible with the review paper activity, if executed by the same originator on the same paper. In CLIMB, it can be specified as[4]:

$$\begin{aligned}&\mathbf{H}(exec\,(t_c, E^s_{ID}, I_{ID}, submit_paper, O, [ID]), T_s)\\&\wedge role(O, author)\\\rightarrow&\mathbf{EN}(exec\,(t_s, E^r_{ID}, I_{ID}, review_paper, O, [ID, R]), T_r).\end{aligned}$$

Obviously, the CLIMB knowledge base could be extended by dealing not only with participant-role relationships, but expressing all the background knowledge of the system, such as for example pricing conditions in a BP, or the taxonomy of drugs in a clinical setting.

6.3.4 *Cross-Flow Constraints*

As far as now, we have always made the implicit assumption that all the constraints of a ConDec model act within a specific instance of

3 Remember that, by default, the knowledge base produced by the translation function from ConDec to CLIMB is the empty knowledge base.

4 We show only one side of the not coexistence constraint. Indeed, remember that [a]•‖•[b] is equivalent to [a]•‖—[b].

the system; many multiple instances of the system can evolve in parallel, istantiating the model's constraints independently from each other. This is well reflected in the example formalized in Section 6.3.3: all the ICs of the specification use the same instance identifier in happened events and expectations: an expected occurrence must be generated by the interacting entities within the same instance, and negative expectations have a scope limited to the instance.

However, nothing prevent us to relax such a limitation, modeling *cross-flow* constraints which span across multiple instance. Let us consider for example the case of an e-commerce choreography, where customers and sellers interact to buy/sell goods. If the seller detects that a certain customer C is behaving in a fraudolent way, then it will never deliver anything to C in the future, even in new instances of execution. Another cross-flow example, referred to the Submit&Review ConDec++ fragment discussed in Sections 6.2.3 and 6.3.3, is the insertion of a global uniqueness constraint, stating that if an author receives a certain paper identifier attesting that the paper has been correctly submitted, then such a paper identifier must be unique w.r.t. all the possible instances of the conference submission system. Only if the identifier is globally unique, it is possible to track all the relevant information concerning its review. If two instances share the same paper identifier, then it would be impossible to separate the review process of the two papers.

Cross-flow constraints can be modeled by simply introducing two instance identifiers variables, without imposing that they must unify. For example, to model the "paper identifiers global uniqueness" constraint, we could employ the following CLIMB IC:

$$
\begin{aligned}
&\mathbf{H}(exec\left(t_c, E^{s1}_{ID}, I^{s1}_{ID}, submit_paper, O^{s1}, [ID]\right), T^{s1}) \\
\rightarrow &\mathbf{EN}(exec\left(t_s, E^{s2}_{ID}, I^{s2}_{ID}, submit_paper, O^{s2}, [ID]\right), T^{s2}) \\
&\wedge T^{s2} > T^{s2}.
\end{aligned}
$$

The only shared variable among the happened event and the negative expectation is the paper identifier. Therefore, all the other variables contained in the negative expectation are universally quantified: they forbid the presence of a whatsoever submission in each possible instance of the system, if it shares the same identifier of the occurrence which triggers the IC. Note that, as in the formalization of the `0..1` ConDec constraint, the difference between the two executions is modeled as a difference between the involved execution times[5].

5 Forgetting such a time constraint would produce an IC stating that no paper submission is accepted by the system: the same happened event triggering the IC would be forbidden by the generated negative expectation!

7

RELATED WORK AND SUMMARY

Contents

7.1 Related Work **123**
7.1.1 Business Process Management 123
7.2 Clinical Guidelines **125**
7.2.1 Service-Oriented and Systems 126
7.2.2 Multi-Agent Systems 127
7.3 Summary of the Part **128**

In this Chapter, related work regarding the specification languages of interaction models is presented. Then, the major contributions of this part of the dissertation are briefly summarized.

7.1 RELATED WORK

We present related work by taking into account the different areas discussed in Chapter 2:

- Business Process Management;
- Clinical Guidelines;
- Service-Oriented Systems;
- Multi-Agent Systems.

We discuss in particular those approaches that face the specification of such systems by adopting a flexible, declarative and open approach.

7.1.1 *Business Process Management*

The need for flexibility in BPM systems has been recognized by many different authors as a key feature towards the effective adoption of Business Processes and their ability to adapt to changing business strategies and contexts.

In [185], van der Aalst proposes a framework in which process models are enhanced with the possibility of including *generic* parts, for the sake of flexibility. In particular, models are hierarchically decomposed in sub-processes, which can be of two types: concrete and generic. While concrete sub-processes are statically linked to an underlying complete specification, generic sub-processes are linked in a dynamic

fashion. That is, when the execution reaches a generic sub-process, the model that must be effectively executed is selected on-the-fly.

In [173], Sadiq et al. introduce an hybrid framework, in which a procedural and a declarative modeling style benefit from each other. In particular, BP models mix pre-defined activities, inter-connected by means of procedural control-flow relations (i.e., sequence, fork, split and merge constructs), with partially specified parts, called *pockets of flexibility*. A pocket of flexibility is a partially specified sub-process, consisting of activities and a set of *order* and *inclusion* constraints. These constraints share many similarities with the ConDec response, precedence, existence and choice constraints. During the enactment phase, when a pocket of flexibility is encountered, the user is guided in the dynamic construction of a completely specified sub-process model, combining the enclosed activities so as to satisfy all the order and inclusion constraints. The authors then present Chamaleon, a prototypical management system supporting the enactment of pockets of flexibility, as well as a set of verification techniques able to detect conflicting and redundant constraints.

In [87], Goedertier proposes the EM-BrA^{2}CE framework, which supports declarative BP modeling by relying on a vocabulary based on the OMG Semantics for Business Vocabulary and Business Rules (SBVR) specification. The vocabulary of EM-BrA^{2}CE supports the specification of highly expressive statements, concerning business concepts and rules of the organization, the involved interacting entities (called agents) and the events characterizing the execution of the systems. Among the advantages of the EM-BrA^{2}CE specification language, we cite the use of Natural Language Expressions to characterize business rules in a human-readable and machine-understandable way, and *local closure*, i.e., the possibility of specifying which parts of the modeled system must be treated with an open or closed world assumption, making it possible to capture semi-open models. Among the limits of the approach, we cite the lack of an underlying temporal logic: while the translation from ConDec to LTL and CLIMB enables the formal specification of qualitative and quantitative time constraints, EM-BrA^{2}CE cannot deal with temporal constructs.

In [158, 157], ConDec models are specified and enacted by a prototypical management system called DECLARE, thanks to the translation to LTL. Beside the basic features of ConDec, which have been presented in Chapter 3, DECLARE provides support for:

- modeling instance-related data, whose value is affected by the execution of activities;
- dealing with a non-atomic model of activities;
- attaching data-related conditions to relation constraints, inhibiting their triggering when the associated condition is false.

Activities-related data, which have been introduced in Chapter 6, are not treated. Furthermore, all these added features are not part of the formalization, i.e., they are treated by DECLARE separately from the

LTL conjunction formula which formalizes the model. In this dissertation, instead, all the added features are part of the CLIMB formalization, and can be therefore formally verified with the reasoning techniques that will be presented in the next parts.

7.2 CLINICAL GUIDELINES

Several proposals for representing the procedural knowledge of Clinical Guidelines (CGs) can be found in the literature: in [155, 182] the interested reader can find surveys and comparisons of the most popular ones.

In [150], Mulyar et al. provide a pattern-based evaluation of four among the most popular CGs modeling languages, focusing in particular on the control-flow perspective which, in the clinical setting, determines the order of actions in medical treatments. In [149], the authors point out that this comparison reveals limited capabilities of the modeling language w.r.t. flexibility aspects, which are identified as a critical issue in the medical setting. In fact, the authors argue that there is a high degree of unpredictability when treating a patient: exceptional situations and emergencies may arise at any point in time, making it difficult or even impossible to foresee what activity should be performed next. In other words, even if they do not explicitly make a distinction between CGs and the Basic Medical Knowledge, as we have done in Section 2.5, they recognize that the application of an ideal CG in a concrete situation is a complex task, which requires novel modeling techniques. Instead of investigating the integration of procedural and declarative knowledge, they propose to overcome this lack by modeling control-flow in a declarative and flexible manner. In particular, they propose the adoption of CigDec as a suitable modeling language. CigDec is a variant of the ConDec notation, which has been described in Chapter 4.

To the best of our knowledge, only the research activity carried out within the Protocure and Protocure II EU projects[1] has attempted to explicitly couple CGs and the Basic Medical Knowledge. CGs are modeled using the Asbru language[141], while the Basic Medical Knowledge is captured by means of LTL formulae.

The integration between a declarative and a procedural style of modeling have instead been recently investigated in the BPM context. In [157], a layered integration between ConDec and the YAWL procedural language [187] is described. The integration is "layered" because a ConDec non-atomic activity could be internally modeled as a YAWL process, or, conversely, a YAWL activity could be expanded as a ConDec sub-diagram. The integration of the two kind of knowledge at the same level, as in the case of Basic Medical Knowledge and CGs, is however not investigated.

Another approach investigating this line of research is provided in [126]. The authors propose to complement the specifications of the adaptive BPM System Adept (which supports flexibility by change and

1 `http://www.protocure.org/`

by deviation, and has been largely applied to the medical setting) with domain knowledge, through the adoption of semantic constraints. A generic criterion for semantic correctness of processes is given, avoiding semantic conflicts like violation of dependency and mutual exclusion between activites. Semantic constraints resemble the one of ConDec and CLIMB, but CLIMB is more expressive in that it can take into account also contexts, time, locations and data.

7.2.1 *Service-Oriented and Systems*

As pointed out in Chapter 2, state-of-the-art approaches model Service-Oriented systems by adopting a procedural flavor, i.e., by completely fixing control and message-flow of the interacting services. This is reflected also on the underlying formal languages to which these specification languages are mapped: Petri Nets [139, 193], state machines [23] and message sequence charts [79] to cite some.

In [207], Zaha et al. argue that the procedural nature of such modelling languages is an obstacle in developing choreographies of autonomous web services because possible orderings of message exchange between services must be explicitly included in the model. They propose a declarative language called *Let's Dance* for modeling interactions of web services in a flexible way. *Let's Dance* provides a graphical notation for capturing message exchange patterns, and adopts

- Computation Tree Logic (CTL) for formally modeling the message exchange between services;
- π-calculus [142] for capturing the corresponding execution semantics [62].

Let's Dance and ConDec share the same philosophy: the possible orderings among activities/messages are not explicitly enumerated, but rather are implicitly derived from constraints, as all the orderings which comply with them.

A vast literature relies on the use of variants of temporal logics for capturing the properties that a service composition must meet. For example, in [96] a restricted version of LTL is translated to the XQuery language for monitoring web services. LTL is also adopted in [69] for verification of correctness properties of service compositions., while [81] exploits the SPIN model checker [99] to verify LTL properties of service conversations.

The issue of flexibility and declarativeness in modeling (semantic) web services has also been deeply investigated in the semantic web community. The two principal approaches in this area are the OWL-based Web Service Ontology[2] (OWL-S) and the Web Service Modeling Ontology[3] (WSMO). Both approaches model the contract published by web services in terms of their Input, Output, Preconditions and Effect (IOPEs). All these elements are semantically annotated, so as

2 http://www.w3.org/Submission/OWL-S
3 http://www.wsmo.org

to express their intended meaning in a machine-understandable way. In this respect, service retrieval/composition is done by taking into account the semantics of concepts involved in the published contracts, overcoming the limitations of a pure syntactic matching. Therefore, while ConDec is focused on flexibility w.r.t. the specification of control flow, these approaches guarantee flexibility concerning terminological issues.

7.2.2 *Multi-Agent Systems*

To the best of our knowledge, the SOCS Project, upon which SCIFF found its main motivations, is the first attempt to use Abuctive Logic Programming (ALP) to reason about agent interaction at a social, open level. Many other logics have been proposed to represent richer social and institutional entities, such as normative systems and electronic institutions.

Artikis et al. [13] present a theoretical framework for providing executable specifications of particular kinds of multi-agent systems, called open computational societies, and present a formal framework for specifying, animating and ultimately reasoning about and verifying the properties of systems where the behaviour of the members and their interactions cannot be predicted in advance. Three key components of computational systems are specified, namely the social constraints, social roles and social states. The specifications of these concepts is based on and motivated by the formal study of legal and social systems (a goal of the ALFEBIITE project), and therefore operators of Deontic Logic are used for expressing legal social behaviour of agents [204, 196].

SCIFF (and therefore also CLIMB) shares some concepts with normative systems, being **E** related with the $\mathcal{O}$ (obligation) operator of deontic logic [174], and **EN** with the $\mathcal{F}$ (forbidden) operator. Similarities and differences between the two frameworks have been studied in [5].

The social approach to the semantic characterization of agent interaction is adopted by many researchers to allow for flexible, architecture-independent and verifiable protocol specification. Prominent schools, including Castelfranchi's [43], Singh et al.'s [179, 206], and Colombetti et al.'s [77, 60] indicate commitments as first class entities in social agents, to represent the state of affairs in the course of social agent interaction. The resulting framework is more flexible than traditional approaches to protocol specification, as it does not necessarily define action sequences, nor it prescribes initial/final states or necessary transitions.

For example, an interaction model could state that the seller is committed to send a certain good if the customer has paid for it. Such a requirement could be represented as a conditional commitment

$$\mathsf{CC}(\mathsf{seller}, \mathsf{customer}, \mathsf{orderPaid}, \mathsf{orderSent})$$

The conditional commitment states that the customer is committed to the seller that when $\mathsf{orderPaid}$ becomes true, it will bring about

orderSent. It is worth noting that the commitment centers around the notion of "property that must be brought about", and that the activities which have the effect of bringing about the property are left unspecified. Interacting entities are then free to choose the most suitable activities on their own, provided that such activities are able to bring about the desired properties. The ConDec approach is different: effects are not first-class object of the language, and therefore the activities must be chosen at design time.

Similarly to the activity life cycle in the BPM setting, commitments have a life cycle reflecting their evolution in relation to the occurrence of activities and their corresponding effects. In [206], a variant of the Event Calculus (EC) [177] is applied to commitment-based protocol specification. The semantics of messages (i.e., their effect on commitments) is described by a set of *operations* whose semantics, in turn, is described by *predicates* on *events* and *fluents*; the commitment life cycle is specified by a set of EC axioms relating operations and fluents with the possible states of the commitment. In Chapter 14, we will show how the SCIFF-lite language can accommodate a reactive form of EC, to address monitoring issues. This attests that the theory of commitments can be grounded on SCIFF, and that SCIFF can be therefore considered as a unifying framework for dealing with both ConDec constraints and commitments.

7.3 SUMMARY OF THE PART

We have provided an overview of different settings, in which the specification of interaction is still a challenging issue: Business Process Management (BPM), Clinical Guidelines, Service Oriented and Multi-Agent Systems. We have shown that current mainstream approaches fail to model interaction at the right level of abstraction, leading to rigid, over-specified and over-constrained model. We have claimed that such a lack comes from their procedural and closed nature, and that they must be complemented with an open and declarative perspective.

Two different languages for the declarative specification of open interaction models have been then targeted:

- ConDec, a graphical language which focuses on the (minimal) set of constraints to ensure that the collaboration is correctly carried out, instead of imposing a rigid sequence of steps;

- CLIMB, a formal language inspired by SCIFF which captures the desired and undesired behaviours by means of rules relating what has already occurred in the course of interaction to what is (not) expected to occur, and that provides a clear declarative semantics of compliance with such rules.

We have then shown that ConDec can be successfully translated to CLIMB, enabling the possibility of specifying interaction models at a graphical intuitive level, automatically obtaining a corresponding underlying formalization. We have assessed the suitability of CLIMB in

this context, showing that the proposed formalization is sound w.r.t. the original formalization of ConDec, which has been given in terms of propositional Linear Temporal Logic (LTL). We have then broaden the comparison between the two formalisms, showing that, from a theoretical point of view, SCIFF is strictly more expressive than LTL.

We have then exploited the expressiveness of CLIMB to extend ConDec with new interesting features, such as:

- quantitative time constraints, to express delays, deadlines and latencies;
- activity-related data, supporting the insertion of data-related conditions and of non-atomic activities;
- cross-flow constraints, ConDec constraints spanning across multiple instances of the system.

The proposed translation allows us to rely on all the reasoning capabilities provided by SCIFF, for tackling the verification of (extended) ConDec models. This is matter of the next parts of this dissertation.

Part II

STATIC VERIFICATION

8

STATIC VERIFICATION OF DECLARATIVE OPEN INTERACTION MODELS

In formal logic,
a contradiction is the signal of defeat,
but in the evolution of real knowledge
it marks the first step in progress
toward a victory.

— Alfred North Whitehead

Contents

8.1 Desiderata for Verification Technologies **133**
8.2 Verification of a Single Model vs a Composition of Models **134**
8.3 Static Verification of Properties **135**
8.3.1 Existential vs Universal Properties 136
8.3.2 General vs Particular Properties 136
8.3.3 On the Safety-Liveness Classification 138
8.3.4 A ConDec Example 140
8.4 A-priori Compliance Verification **141**
8.5 Compatibility and Legal Compositions **142**
8.5.1 Compatibility Between Local Models 143
8.5.2 From Openness to Semi-Openness 145
8.5.3 Augmenting ConDec Models with Roles and Participants 146
8.6 Conformance With a Choreography **149**

This Chapter introduces the issue of static verification of declarative open interaction models, specified using the ConDec notation. After having discussed the desiderata that a static verification technology must accomplish, two main settings in which static verification is adopted are introduced: verification of a single model vs verification of a composition of models. These two settings are then deeply investigated, discussing the different verification tasks that can be applied on them. The discussion is grounded on different ConDec examples.

8.1 DESIDERATA FOR VERIFICATION TECHNOLOGIES

We argue that, in order to effectively help a developer in the process of ensuring the correctness and safety of an interaction model and in the

related debugging tasks, a static verification framework should satisfy four main desiderata:

SOUNDNESS AND COMPLETENESS If the verifier states that the system will never reach an undesired state of affairs, we must be *sure* that this is truly the case. In other words, our verifier should not compute wrong answers, i.e., it must be *sound*. On the other hand, it must be able to provide a positive answer if such a positive answer exists, i.e., it must be *complete*. If soundness/completeness are guaranteed only under certain assumptions, the verifier must be accompanied by techniques able to identify if these assumptions are respected by the model, taking specific countermeasures and alerting the user if that is not the case.

GENERATION OF (COUNTER-)EXAMPLES When the verifier returns a negative answer, which shows that the model must be revised, the verifier must help isolating the relevant part of the model which violates the requirement. In this respect, it would be desirable that a verification framework does not only provide correct answers, but is also able to generate (counter-)examples in terms of an execution trace which motivates the provided answer

PUSH-BUTTON STYLE After having produced the input to the verifier, the verification process cannot be accomplished manually by the user, nor should be carried out in a semi-automatic way, requiring a constant user interaction; it should operate, as much as possible, in an automatic way.

SCALABILITY Since static verification is carried out at design time, the verifier's performance is not time-critical. However, in a typical usage scenario, static verification is repeatedly exploited by the user during the design phase, to constantly check the model under development and trigger new design cycles when it must be revised (see Section 8.2). In this respect, the design phase involves a constant interaction between the verifier and the modeler, and therefore scalability and good performances of the verification framework matter.

8.2 VERIFICATION OF A SINGLE MODEL VS A COMPOSITION OF MODELS

Static verification aims at checking the model under development during the design phase, *before* the execution. More specifically, static verification can be employed for dealing with two different problems

Relationship between static verification and the model's life-cycle

VERIFICATION OF A SINGLE MODEL During the design phase, a model must be repeatedly verified to ensure its correctness and consistency, to check if it really meets all the requirements of the modeler and the business objectives for which it has been conceived, and to assess its compliance with internal policies, best practices, external regulations and norms. In this setting, static verification

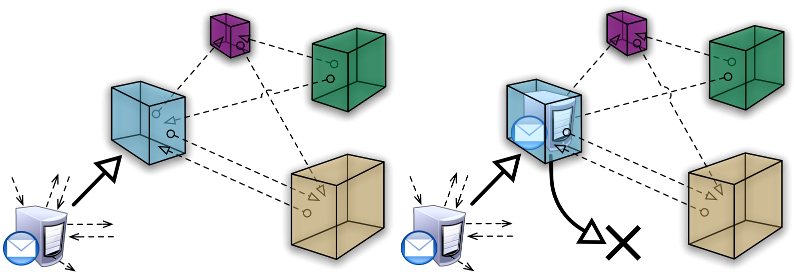


Figure 26: Conformance and replaceability of services in a choreography.

is constantly exploited to follow the evolution of the model, enabling the premature identification of errors and undesired situations which, if encountered at run-time, could be costly to repair or could even compromise the entire system. When a problem is detected, a new design cycle can be triggered to properly revise the model.

VERIFICATION OF A COMPOSITION OF MODELS When the system under study involves different separated components, and many possible compositions can be synthesized, static verification supports the modeler in the process of checking whether different concrete components can correctly interact with each other and, more generally, in the identification of "legal" compositions. A typical example of this setting is Service Composition, in which a global choreography is designed to capture in an abstract way the mutual obligations and requirements of each interacting party, and then a set of concrete services must be found s.t. they can interact correctly by accomplishing the choreography rules of engagement. The same methodology is applicable to deal with service conformance and replaceability, i.e., to respectively check whether a given concrete service can play a role inside the choreography or if a service playing a certain role in the choreography can be substituted with another concrete service (see Figure 26).

Static verification and service composition

8.3 STATIC VERIFICATION OF PROPERTIES

The different static verification tasks carried out on a single model can be generally characterized as verification of *properties*, sketched in Figure 27. Properties formally capture requirements that the user poses on the system under development. Requirements may differ from one another in nature and purpose, and can be classified along different axes.

Examples of properties

For example, a requirement may express that an external regulation must be ensured in every execution of the system, or it may capture hidden dependencies among activities, which should be entailed by

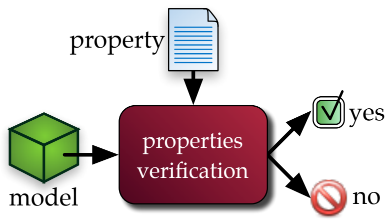


Figure 27: ConDec and verification of properties.

the model even if not explicitly expressed inside it. Here we focus on two fundamental dimensions: *existential vs universal* and *general vs particular* properties.

In the following, when the desired property will be expressible in ConDec, we will use the graphical notation to represent it. For example, the property stating that "a receipt must be eventually sent" could be expressed by means of the [1..* send receipt] ConDec constraint.

8.3.1 *Existential vs Universal Properties*

When the modeler is interested in verifying whether the ConDec model under development always meets a best practice of the company, the intended meaning is that the property must be respected in any possible execution of the system. Conversely, the reachability of a certain situation is guaranteed if there exists at least an execution which reaches that situation. Generalizing, properties quantify over execution traces in two complementary ways: existential and universal.$\mathcal{CM}$

Existential entailment of a property

Definition 8.1 ($\exists$-entailment). A property Ψ is $\exists$-entailed by a ConDec model $\mathcal{CM}$ ($\mathcal{CM} \models_{\exists} \Psi$) if at least one execution trace supported by $\mathcal{CM}$ entails the property as well. If that is the case, then one of the supported execution traces can be interpreted as an *example* which proves the entailment.

Universal entailment of a property

Definition 8.2 ($\forall$-entailment). A property Ψ is $\forall$-entailed by a ConDec model $\mathcal{CM}$ ($\mathcal{CM} \models_{\forall} \Psi$) if all the possible execution traces supported by the model entail the property. If that is *not* the case, the generation of an execution trace which follows the constraints of the model but breaks the property can be interpreted as a *counter-example* which disproves the entailment.

8.3.2 *General vs Particular Properties*

Properties can be *general*, if they describe basic/fundamental requirements which must be satisfied by an arbitrary interaction model, independently from its specific nature, or *particular*, if they target a specific model and domain. Examples of particular, domain-dependent prop-

Examples of particular properties

erties are that a customer will obtain the ordered goods when executing an e-commerce protocol, or that a physician executing a clinical guideline will not administer a certain drug twice in the same day. Examples of general properties in a procedural setting, such as Petri Nets, are liveness and deadlock freedom, whereas examples of general properties in the ConDec setting are *conflict freedom* and *absence of dead activities* [159, 146, 157], which, roughly speaking, identify global and local inconsistencies inside a model.

Examples of general properties

Definition 8.3 (Conflict-free model[157]). A ConDec model $\mathcal{CM}$ is *conflict-free* iff it supports at least one possible execution trace, i.e., iff

Conflict-free models

$$\mathcal{CM} \models_{\exists} \mathtt{true}$$

A conflicting model is an over-constrained model: it is impossible to satisfy all its mandatory constraints at the same time. It cannot be executed at all, and it is therefore globally-inconsistent. As a consequence, it must be revised by rearranging (a sub-set of) its constraints.

In order to identify which sub-set of constraints is the source of conflict, one should check conflict-freeness on each possible combination of mandatory constraints. Therefore, if the ConDec model under study contains a conflict and is composed by n mandatory constraints, 2^{n-1} combinations should be checked to exactly identify the conflicting part.

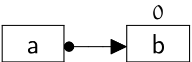
Definition 8.4 (Dead activity[157]). Given a ConDec model $\mathcal{CM}$ and an activity a, a is *dead* w.r.t. $\mathcal{CM}$ iff it can never be executed in any execution trace supported by $\mathcal{CM}$ i.e., iff

Dead activities

$$\mathcal{CM} \models_{\forall} \overset{0}{\boxed{\mathsf{a}}}$$

If a dead activity is executed, independently from the other executed activities at least one constraint of the model is/will be violated. Since dead activities are non executable, their presence reveals possible local inconsistencies in the ConDec model, that must be properly treated during the design phase.

Example 8.1 (Example of dead activities). *Let us consider the simple ConDec model*



Activity b *is dead "by construction": it is associated to an* absence *constraint, which directly expresses that it cannot be never executed, i.e., that it is a dead activity.*

Activity a *is dead as well. Let us suppose, by absurdum, that there exists an execution trace supported by the model which contains the execution of* a. *Due to the presence of the* a •—▶ b *constraint, such an execution trace must also contain a consequent execution of activity* b. *However, the execution of* b

is unlegal, being b *a dead activity. Therefore, there does not exist a supported execution trace containing* a*; this attests that* a *is "indirectly" a dead activity.*

8.3.3 *On the Safety-Liveness Classification*

Formal verification

In the field of *formal verification*, which, roughly speaking, address the problem of proving or disproving the correctness of a model underlying a system w.r.t. a certain property, properties are usually classified along the *safety* vs *liveness* dimension [110].

Intuitive and formal definition of safety and liveness

Intuitively, a safety property asserts that *something bad will not happen* during any execution of the system, whereas a liveness property states that *something good must happen*.

In formal verification, the execution traces of the system are usually of *infinite length*. Under this assumption, the intuitive concepts of safety and liveness are formally denoted as follows:

- Ψ is a safety property iff every execution trace which violates Ψ contains a *finite prefix* to which the violation can be referred to. In other words, there is no way to extend such a prefix with an infinite suffix, s.t. the property becomes satisfied: violation can be detected in a finite initial part of the trace, and its occurrence is irremediable.
- Ψ is a liveness property iff, given an arbitrary partial execution trace of the system, it is possible to find an infinite suffix so that the resulting infinite execution trace satisfies the property.

As pointed out in [151], there is a gap between the intuitive and the formal characterization of safety and liveness. Therefore, the intuitive meaning stated above is misleading, making often difficult to evaluate whether a given property is a safety or a liveness property. For example, while the LTL formalization of the `response` ConDec constraint is a liveness property, its metric variation, stating that every execution of a certain activity must be followed by the execution of another activity within a maximum time span, is a safety property.

This drawback, together with the discrepancy between the time structure upon which safety and liveness has been defined and the one of ConDec[1], makes the safety/liveness distinction inadequate for classifying ConDec properties.

We motivate this assertion by considering the following typical LTL safety and liveness properties used in formal verification, then discussing if they are reasonable in the context of ConDec or not.

Examples of safety and liveness properties

A. $\Diamond a$, which corresponds to the LTL formalization of the [a] (1..∗) ConDec constraint , is a liveness property;

B. $\Box(\neg a)$, which corresponds to the LTL formalization of the ConDec [a] (0) constraint and is commonly called *invariant property*, is a safety property;

1 The formal characterization of safety and liveness property is given on infinite-lenght traces, while in ConDec traces are always finite.

C. $\Box a$ is a safety property;

D. $\Box(a \Rightarrow \Diamond b)$, which corresponds to the LTL formalization of the a •——▶ b ConDec constraint, is a liveness property;

E. a variation of the `response` property, stating for example that b must occur no later than 10 time units after a (which is commonly called *promptness* property[116]), is instead a safety property;

F. $\Box\Diamond a$ (fairness property), is a liveness property.

Termination property in ConDec

In the ConDec setting, it becomes apparent that only certain safety and liveness properties have a meaning. Since there is always the underlying assumption that the process must terminate, we should imagine that each property is implicitly put in conjunction with a *termination* property [186].

DEFINITION 8.5 (ConDec termination property). Given a ConDec model $\mathcal{CM} = \langle \mathcal{A}, \mathcal{C}_m, \mathcal{C}_o \rangle$, the LTL *termination property* $\mathtt{term}(\mathcal{CM})$ states that a termination event $e \notin \mathcal{A}$, incompatible with all other activities, must eventually occur, and then is executed infinitely often in the future[2]:

$$\mathtt{term}(\mathcal{CM}) \triangleq \Diamond e \wedge \Box(e \Rightarrow \bigcirc e) \wedge \forall\ a \in \mathcal{A},\ \Box(e \Rightarrow \neg a)$$

The termination property enables the possibility of embedding each finite ConDec trace $\mathcal{T}_{\mathcal{L}}$ in an infinite trace, composed by $\mathcal{T}_{\mathcal{L}}$ followed by an infinite suffix containing the execution of e in each state.

Let us now for example consider the two safety properties $\Box(\neg a)$ and $\Box a$. Despite their structural similarity, while the first is legitimate in the ConDec setting (it is in fact a ConDec constraint), the second one is not, because it states that activity a must be continuously and infinitely executed, hence it is incompatible with the termination property: $\Box a \wedge \Diamond e \wedge \Box(e \Rightarrow \bigcirc e) \wedge \Box(e \Rightarrow \neg a)$ is unsatisfiable.

Similarly, while the liveness property $\Diamond a$ is acceptable and can be interpreted as "activity a must be performed at least once before the termination of the execution", fairness is never guaranteed by a ConDec model, because it is impossible to execute a certain activity infinitely often.

In conclusion, the implicit assumption that the execution of a ConDec model must eventually terminate (i.e., that execution traces are always finite) makes some typical safety/liveness properties senseless. Therefore, we will not distinct, in the following, between safety and liveness properties, but we will instead rely on the existential vs universal dimension.

2 In some cases, the incompatibility between e and the other activities is not expressed, because there is the implicit assumption that in each state at most one event may happen, and therefore the presence of e automatically excludes the possibility of executing the other activities.

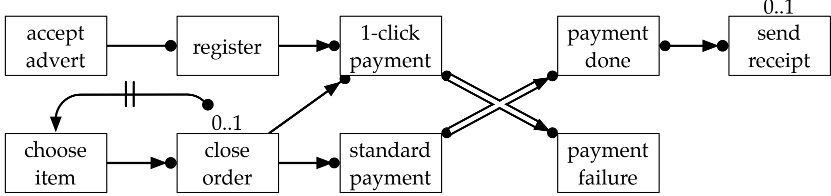


Figure 28: ConDec model of an order&payment protocol.

8.3.4 *A ConDec Example*

Figure 59 shows the ConDec specification of an order&payment protocol. There are three different parties involved–a merchant, a customer, and a banking service to handle the payment– which are left implicit.

An execution of the model deals with a single order, hence the `0..1` constraint on the close order activity. An order can be closed if at least one item has been chosen, and after the closure no more items can be added to the order. Once the order has been closed, two possible payment methods are available: 1-click payment and standard payment. 1-click payment is a special fast payment method which becomes available only if the customer has previously registered; registration, in turn, requires that the customer explicitly accepts to receive advertising (either before or after the registration). The payment phase involves an alternate succession between the payment and the answer from the banking service: after having executed a payment, further payments are forbidden until a response is received, and only a single response is allowed for each payment. Finally, only if the banking service sends a payment confirmation, a single receipt is delivered by the seller, attesting that the order has been correctly completed.

Sample supported executions are: the empty trace (no activity executed), a trace containing one instance of accept advert followed by a registration, and a model containing 5 instances of choose item followed by a close order (in this case, the order is not paid and not delivered by the seller).

Let us now consider some examples of properties verification on this model. We present four different queries which reflect the dimensions presented so far (general vs particular properties, existential vs universal verification).

Query 8.1. *Is* send receipt *a dead activity?*

This is a general property, verified in a universal way. A positive answer to Query 8.1 means that send receipt cannot be executed in any possible valid model, indicating that probably there is a mistake in the design (obtaining a receipt is the business goal of the customer). In our example, a verifier should return a negative answer, together with a sample valid execution, such as: choose item → close order → standard payment → payment done → send receipt, which amounts to a proof that send receipt is not a dead activity.

Query 8.2. *Is it possible to complete a transaction under the hypotheses that the customer does not accept to receive ads, and the merchant does not offer standard payment?*

This is a particular domain-dependent property that must be verified in an existential way. A verifier should return a negative answer. In fact, to complete a transaction the customer must pay, and, the only accepted payment modality is 1-click payment; however, this payment modality is supported only if the customer has previously registered, which in turn requires to accept ads and thus contradicts customer's requirement.

Query 8.3. *It is true of all transactions that with* 1-click payment *a receipt is always sent* after *the customer has accepted the ads?*

This is a domain-dependent property, which must be verified in an universal way. A verifier should discover that the property does not hold: since there is no temporally-ordered constraint associated with accept advert, accept advert is not forced to be executed *before* send receipt. The existence of an execution trace in which the customer chooses 1-click payment but accepts ads only after having received the receipt amounts to a counterexample against the universal query. More specifically, a verifier should produce a counter-example like: choose item → close order → register → 1-click payment → payment done → send receipt → accept advert. That could lead a system designer to decide to improve the model, e.g., by introducing an explicit precedence constraint from send receipt to accept advert.

Query 8.4. *Is there a transaction with no* accept advert*, terminating with a* send receipt *within 12 time units as of* close order*, given that* close order, 1-click payment*, and* standard payment *cause a latency of 6, 3, and 8 time units?*

This is an existential query with explicit times, used to express minimum and maximum latency constraints on the activities execution. It turns out that the specification is unfeasible, because refusing ads rules out the 1-click payment path, and the standard payment path takes more than 12 time units. A verifier should therefore return failure.

8.4 a-priori compliance verification

A special case of property verification is the one in which the property represents a set of regulations that must be always guaranteed by the model (see Figure 29). A typical case is the one in which the model under development represents the Business Process of an organization, and the set of regulations formalizes the business contract stipulated between the organization and the customer; verifying whether the BP actually meets the business contract is of key importance, especially if

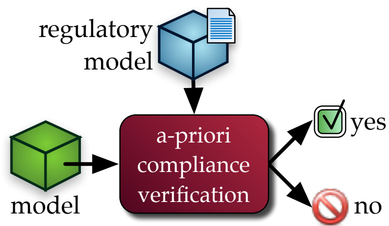


Figure 29: A-priori compliance verification.

we think that these two aspects are often maintained separated by the organization [90].

The importance of this issue has been further grown in the last years, after the outbreak of high-profile financial scandals (Enron and World-Com in the USA to cite some). New legislation such as the Sarbanes-Oxley Act has been produced as a response to such scandals, increasing the interest on corporate governance and compliance and on the corresponding tools and methodologies.

A-priori compliance verification in ConDec

In the ConDec setting, a-priori compliance evaluation deals with the problem of ensuring that each execution supported by a ConDec model meets all the rules and norms of a regulatory model. Therefore, it can be reduced to a universal verification of properties.

Definition 8.6 (A-priori compliance). A ConDec model $\mathcal{CM}$ complies with a set of regulations $\Psi_1, \ldots, \Psi_n$ iff all execution traces supported by $\mathcal{CM}$ entail all the regulations:

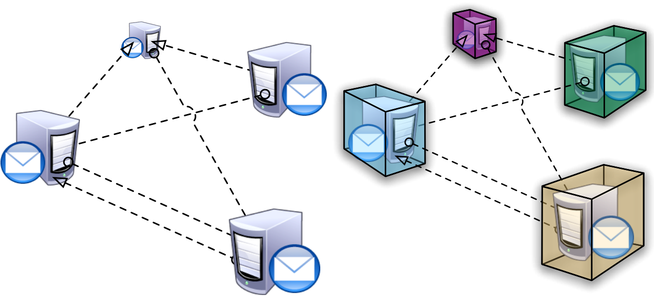
$$\forall\, i \in [1, n],\ \mathcal{CM} \models_{\forall} \Psi_i$$

We speak of "a-priori" compliance because the analysis is carried out before the enactment, i.e., by taking into account a model and not its executions. A positive answer attests that if all the parties will behave in accordance with the model's constraints, then the regulatory model will be surely respected. Anyway, auditing and verifying the actual executions is fundamental, because in many different settings there is no guarantee that, at run-time, participants will follow the model, and behavioural deviations could lead to fraud and non-compliance. This topic will be discussed in Chapter 15.

8.5 COMPATIBILITY AND LEGAL COMPOSITIONS

When it comes to verification of compositions of models, two different sub-cases may arise, as sketched in Figure 30 for the service-oriented setting:

BOTTOM-UP COMPOSITION In this case, the models of all parties (called *local models* thereafter) are directly composed, in order to make them interact and mutually benefit from each other.



(a) Service composition: local models are directly put together.

(b) Service choreography: in addition to the constraints of each local model, the shared global rules of engagement are explicitly captured in an abstract global model.

Figure 30: Two complementary methods to compose local interaction models for realizing a global choreographic model.

A typical example of this case is the one in which the possibility of combining a ticket-reservation service with a hotel-booking service is exploited in order to obtain an integrated service able to organize travels.

composition with an explicit choreographic model In this case, the rules of engagement for achieving a global collaboration are explicitly negotiated by the different parties and captured in a dedicated choreographic model (see Section 2.3.2). The choreographic model plays the role of a public, global shared contract which must be respected by the interacting local models. Such a contract is "abstract", in the sense that its rules of engagement do not predicate over concrete services, but on abstract service roles. After having completed the specification of the choreography, the problem is then to find a legal composition of local models which is, at the same time, consistent and compliant with the choreography.

Typical examples of this scenario are B2B settings, where different organizations are interested in cooperating with each other. Here, the choreography makes explicit the mutual agreement between the parties, which are then free to choose which concrete services will be employed as long as consistency and compliance with the choreography are respected.

8.5.1 *Compatibility Between Local Models*

Checking if two local models can suitably interact is called, in the ConDec setting, *compatibility* verification [157]. Roughly speaking, if two local models are compatible, then there exists at least one course of

interaction which is supported by both models. We precisely capture such an intuitive meaning by defining the concept of *composite model*, obtained by combining the activities and constraints of the local models to be composed, and by relating compatibility of the local models to conflict-freeness of the composite model.

DEFINITION 8.7 (Composite model[157]). Given n ConDec models $\mathcal{CM}^i = \langle \mathcal{A}^i, \mathcal{C}^i_m, \mathcal{C}^i_o \rangle$ and $\mathcal{CM}^2 = \langle \mathcal{A}^2, \mathcal{C}^2_m, \mathcal{C}^2_o \rangle$ $(i = \dots, n)$, the *composite model* obtained by combining $\mathcal{CM}^1, \dots, \mathcal{CM}^n$ is defined as [3][4]:

$$\text{COMP}\left(\mathcal{CM}^1, \dots, \mathcal{CM}^n\right) \triangleq \langle \bigcup_{i=1}^{n} \mathcal{A}^i, \bigcup_{i=1}^{n} \mathcal{C}^i_m, \bigcup_{i=1}^{n} \mathcal{C}^i_o \rangle$$

DEFINITION 8.8 (Compatibility). Two ConDec models $\mathcal{CM}^1$ and $\mathcal{CM}^2$ are *compatible* if their composition is conflict-free, i.e., iff:

$$\text{COMP}\left(\mathcal{CM}^1, \mathcal{CM}^2\right) \models_\exists \mathtt{true}$$

Obviously, the notion of compatibility can be generalized to the case of n local models. The detection of incompatibility means that a subset of the n local models leads to a conflict. A precise identification of the conflicting local models would require to build all the 2^{n-1} possible compositions, checking conflict-freeness on each of them.

It is worth noting that carrying out a compatibility check is a first important step in the formal evaluation of composition's feasibility, but it is not significant if it is not accompanied by further analysis. The following examples point out this issue.

EXAMPLE 8.2 (Trivial compatibility). *Let us consider a simple situation, in which the two local models*

$\mathcal{CM}^1 =$ a •——▶ b $\qquad$ $\mathcal{CM}^2 =$ a •—||—• b

must be composed. The two models are compatible, because they both support the empty execution trace; therefore, by carrying out solely a compatibility check would seem that a composition can be actually built. However, as soon as an activity is executed, $\mathcal{CM}^1$ and $\mathcal{CM}^2$ are contradictory: both activity a *and activity* b *are dead in the composite model.*

In the general case, if none of the local models contains constraints which impose the execution of a certain activity (i.e., `existenceN`, `exactlyN` *and* `choice` *constraints), compatibility always returns a positive answer, because the empty execution trace is supported.*

Evaluating compatibility is the first step towards the guarantee that a composition can be built, but it must be followed by further verifications, aimed at answering to questions like: "Are all activities of the local models still executable in the composition?", "Does the composition effectively achieve the business goals for which the composition

3 We suppose that the different local models share the same *ontology* of activities.
4 Note that in [157] the composition operator is defined as $\oplus$.

itself has been built?", "Does the composition respect the regulations and norms of each involved party?". These questions can be answered by means of properties verification on the global model, obtained by composing activities and constraints of the local models (as stated in Definition 8.7). For example, an answer to the first question could be provided by checking if the composite model contains dead activities.

8.5.2 *From Openness to Semi-Openness*

Since a ConDec model is open, it supports the execution of activities not explicitly contained in the model itself. While this feature is desirable when the ConDec model is used in isolation, when the model is composed with another ConDec model, it could cause undesired compositions to be evaluated as correct. The following example clarifies the point.

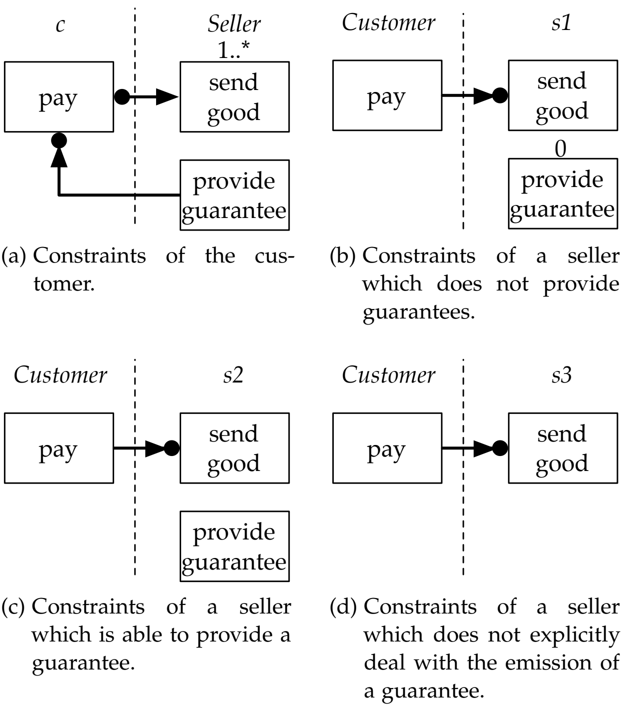
EXAMPLE 8.3 (Composition and openness issues). *When two local models must be composed, each one of them poses requirements on the other one. Let us for example consider the following simple scenario. A customer wants to find a seller to interact with. The customer comes with a ConDec model representing its own desired constraints and requirements. In particular, they express that:*

- *the customer wants to receive a good from a seller;*
- *if the customer pays for a good, then she expects that the seller will deliver it;*
- *before paying, the customer wants the seller to provide a guarantee that the payment method is secure.*

Figure 31 shows the ConDec graphical models ($\mathcal{CM}_C$) of the customer and of three candidate sellers, providing an explicit distinction of the involved roles. The three sellers differ for what concerns the possibility of emitting a guarantee upon request:

1. *the seller depicted in Figure 31(b) ($\mathcal{CM}_S^1$) explicitly states that it does not provide any guarantee upon request;*
2. *the seller depicted in Figure 31(c) ($\mathcal{CM}_S^2$) explicitly supports the possibility of providing a guarantee;*
3. *the seller depicted in Figure 31(d) ($\mathcal{CM}_S^3$) does not mention* provide guarantee *among its activities.*

Following Definition 8.8 checking compatibility between $\mathcal{CM}_C$ and the three candidate sellers would state that $\mathcal{CM}_C$ is not compatible with $\mathcal{CM}_S^1$, but it is compatible with $\mathcal{CM}_S^2$ and $\mathcal{CM}_S^3$. In particular, the two compositions $\mathcal{CM}_C \cup \mathcal{CM}_S^2$ and $\mathcal{CM}_C \cup \mathcal{CM}_S^3$ produce exactly the same global model. However, while the answer given for the first two compositions is in accordance with the intuitive notion of compatibility, the third one is not. In fact, when $\mathcal{CM}_C$ is composed with $\mathcal{CM}_S^2$, the behaviour of the seller is modified in that also



(a) Constraints of the customer.

(b) Constraints of a seller which does not provide guarantees.

(c) Constraints of a seller which is able to provide a guarantee.

(d) Constraints of a seller which does not explicitly deal with the emission of a guarantee.

Figure 31: Local models of a customer and of three candidate sellers.

the constraints of the customer must be respected. Contrariwise, when the composition between $\mathcal{CM}_C$ *and* $\mathcal{CM}_S^3$ *is established, the local model of the customer has the effect of changing the local model of the seller, augmenting it with a new* provide guarantee *activity. During the execution, the customer would expect to receive a guarantee before paying, but this capability has not been mentioned by the seller in its local model, and therefore there could be the case that it is not supported.*

The example clearly shows that the openness assumption must be properly revised when dealing with the composition problem. To ensure that a composition can be established, the obtained global model must obey to the following *semi-openness* requirement: for each involved party, *the activities under the responsibility of that party must also explicitly appear in its local model*.

8.5.3 *Augmenting ConDec Models with Roles and Participants*

In order to ensure the *semi-openness* assumption, each activity must be associated to its corresponding originator or role. The following definition extends the basic definition of a ConDec model with such a relationship.

Definition 8.9 (Augmented ConDec model). An augmented ConDec model is a 4-tuple $\langle \mathcal{AO}, \mathcal{AR}, \mathcal{C}_m, \mathcal{C}_o \rangle$, where:

- $\mathcal{AO}$ is a set of (A, O) pairs where A is an activity and O is its originator;
- $\mathcal{AR}$ is a set of (A, R) pairs where A is an activity and R represents the role of its originator;
- $\mathcal{C}_m$ is a set of mandatory constraints over $\mathcal{AO}$ and $\mathcal{AR}$;
- $\mathcal{C}_o$ is a set of optional constraints over $\mathcal{AO}$ and $\mathcal{AR}$.

If $\mathcal{AR} = \emptyset$, the model is *completely grounded*. Contrariwise, if $\mathcal{AO} = \emptyset$ the model is *abstract*.

In this respect, a ConDec local model is defined as an augmented model containing also an indication about the identifier of the local model, and where an activity is associated either to such an identifier, or to an abstract role.

Definition 8.10 (Local augmented model). A ConDec local augmented model is a 5-tuple $\langle ID, \mathcal{AO}, \mathcal{AR}, \mathcal{C}_m, \mathcal{C}_o \rangle$, where:

- ID is the identifier of the participant executing the local model;
- the other elements retain the meaning of Definition 8.9;
- $\mathcal{AO}$ is a set containing only elements of the type (A, ID).

A role identifies a *class* of originators; in the composition process, abstract roles employed in each local model are mutually grounded to concrete local models which participate to the composition.

Definition 8.11 (Grounding of a model). Given an augmented model $\mathcal{CM}aug = \langle \mathcal{AO}, \mathcal{AR}, \mathcal{C}_m, \mathcal{C}_o \rangle$ and a function `plays` mapping roles to concrete identifiers (i.e., stating that a certain identifier "plays" a given role), the grounding of $\mathcal{CM}aug$ on `plays` is obtained by substituting each role R_i with the corresponding concrete participant identifier $\mathtt{plays}(R_i)$:

- $\mathcal{AO} \downarrow_{\mathtt{plays}} \triangleq \mathcal{AO} \cup \{(A, \mathtt{plays}(R_i)) \mid R_i \in \mathtt{dom}(\mathtt{plays}) \wedge (A, R_i) \in \mathcal{AR}\}$;
- $\mathcal{AR} \downarrow_{\mathtt{plays}} \triangleq \mathcal{AR} / \{(A, R_i) \mid R_i \in \mathtt{dom}(\mathtt{plays})\}$;
- $\mathcal{C}_m \downarrow_{\mathtt{plays}}$ and $\mathcal{C}_o \downarrow_{\mathtt{plays}}$ are updated accordingly.

If $\mathcal{AR} \downarrow_{\mathtt{plays}} = \emptyset$, each role has been substituted by a concrete identifier and the model becomes ground. A legal composite ConDec can be now characterized as an augmented model obtained by composing a set of local models, each one grounded by taking into account the other ones, s.t. the composition is ground.

DEFINITION 8.12 (Augmented composite model). Given a set of augmented local models $\mathcal{L}^i = \langle ID_i, \mathcal{AO}^i, \mathcal{AR}^i, \mathcal{C}_m^i, \mathcal{C}_o^i \rangle$ $(i = 1, \dots, n)$ and a function plays mapping roles to identifiers, the composition of the local models w.r.t. plays is defined as

$$\text{COMP}(\mathcal{L}^1, \dots, \mathcal{L}^n)_{\texttt{plays}} = \bigcup_{i=1}^{n} \mathcal{L}^i \downarrow_{\texttt{plays}}$$

where the union of two augmented models is a shortcut representing the union of each corresponding element[5]. A composition is *legal* iff $\text{COMP}(\mathcal{L}^1, \dots, \mathcal{L}^n)_{\texttt{plays}}$ is ground (see Definition 8.9).

It is now possible to revise the notion of compatibility reflecting also the semi-openness assumption.

DEFINITION 8.13 (Strong compatibility). n local models $\mathcal{L}^i = \langle ID_i, \mathcal{AO}^i, \mathcal{AR}^i, \mathcal{C}_m^i, \mathcal{C}_o^i \rangle$ $(i = 1, \dots, n)$ are *strong compatible* under plays iff their augmented composition $\text{COMP}(\mathcal{L}^1, \dots, \mathcal{L}^n)_{\texttt{plays}} = \langle \mathcal{AO}^\cup, \mathcal{AR}^\cup, \mathcal{C}_m^\cup, \mathcal{C}_o^\cup \rangle$ satisfies the following properties:

- $\text{COMP}(\mathcal{L}^1, \dots, \mathcal{L}^n)_{\texttt{plays}}$ is legal;
- $\text{COMP}(\mathcal{L}^1, \dots, \mathcal{L}^n)_{\texttt{plays}}$ is conflict-free;
- for each (a, ID_i) which belongs to $\mathcal{AO}^\cup$ but does not belong to $\mathcal{AO}^i$, it must hold that:

$$\text{COMP}(\mathcal{L}^1, \dots, \mathcal{L}^n)_{\texttt{plays}} \models_\forall \texttt{absence}((a, ID_i))$$

The third point states that if a certain activity a has been associated to a participant ID_i, but ID_i has not explicitly mentioned a in its specification, then the composition must always ensure that a cannot be executed.

EXAMPLE 8.4 (Example 8.3 revisited). *Let us re-examine the compatibility between the local models of the customer and the second seller shown in Figure 31, supposing that their identifiers are respectively* alice *and* hutter, *and* customer *and* seller *represent their roles. In the composition,* alice *plays the role of* customer, *and* hutter *plays the role of* seller. *Hence,* plays(alice) = customer *and* plays(hutter) = seller.

By adopting the definition of augmented models, the ConDec diagram of alice *is:*

$$\begin{aligned}\mathcal{L}_{alice} = \langle & \{(pay, alice)\},\\ & \{(send\ good, seller), (provide\ guarantee, seller)\},\\ & \{\texttt{existenceN}(1, (send\ good, seller)), \dots\},\\ & \emptyset \rangle\end{aligned}$$

5 I.e., $\langle \mathcal{AO}^1, \mathcal{AR}^1, \mathcal{C}_m^1, \mathcal{C}_o^1 \rangle \cup \langle \mathcal{AO}^2, \mathcal{AR}^2, \mathcal{C}_m^2, \mathcal{C}_o^2 \rangle \triangleq \langle \mathcal{AO}^1 \cup \mathcal{AO}^2, \mathcal{AR}^1 \cup \mathcal{AR}^2, \mathcal{C}_m^1 \cup \mathcal{C}_m^2, \mathcal{C}_o^1 \cup \mathcal{C}_o^2 \rangle$.

(a) Conflicting composition. (b) Local non-conformance. (c) Global non-conformance.

Figure 32: Different possible errors in the realization of a choreography.

The grounding of alice *w.r.t. the* plays *function is* $\mathcal{L}_{alice} \downarrow_{\texttt{plays}}=$

$$\langle\{(\mathsf{pay}, \mathsf{alice}), (\mathsf{send\ good}, \mathsf{hutter}), (\mathsf{provide\ guarantee}, \mathsf{hutter})\},$$
$$\emptyset,$$
$$\{\texttt{existenceN}(1, (\mathsf{send\ good}, \mathsf{hutter})), \ldots\},$$
$$\emptyset\rangle$$

The grounding of hutter *is obtained in a similar way.*

When the two local models are composed, the grounding of alice *causes* (provide guarantee, hutter) *to belong to the set* $\mathcal{AO}^{\cup}$ *of the composition. Since the execution trace* provide guarantee $\rightarrow$ pay $\rightarrow$ send good *is compliant with the composition but* (provide guarantee, hutter) $\notin \mathcal{AO}^{\mathsf{hutter}}$*, the two local models are not strong compatible.*

8.6 CONFORMANCE WITH A CHOREOGRAPHY

In presence of a choreographic model, the global model obtained by composing the chosen local models must not only require their (strong) compatibility, but it must also ensure that such local models correctly play the abstract roles mentioned in the choreography.

A choreography is represented as an augmented abstract ConDec model (i.e., an augmented model associating all the activities to roles and not to concrete participants – see Definition 8.9).

As shown in Figure 32, when realizing a choreography with a set of concrete local models, different possible errors may arise:

CONFLICTING COMPOSITION Independently from the choreography, the chosen local models are not compatible.

LOCAL NON-CONFORMANCE A concrete local models is not able to correctly play, within the choreography, the role it has been assigned to.

GLOBAL NON-CONFORMANCE Even if each single local model is able to correctly play the role it has been assigned to, when the whole composition is built, it could be the case that the global obtained model does not conforms the choreography anymore.

On the other hand, it could be the case that, if considered in isolation, a participant would not be able to play the role it has been assigned to, but it would anyway be able to take part of a conforming composition.

Roughly speaking, such a situation may arise because when the constraints of each local model are joint with the ones of the others, the constraints of the participant could be correctly "completed".

As a consequence, in order to perform a correct verification, it is necessary to first check that the composition is conflict-free, and then verify the whole composition against the choreography.

To verify that a composition conforms to a desired choreography, two approaches can be followed. The *weak* approach states that the composition must be consistent with the choreography constraints in at least one supported execution, while the *strong* approach requires to guarantee that any execution supported by the composition respects the choreography.

Definition 8.14 (Weak conformance). A composition of local models $\text{comp}(\mathcal{L}_1, \ldots, \mathcal{L}_n)_{\texttt{plays}}$ is *weak conformant* with a choreography $\mathcal{C}hor$ iff:

- $\mathcal{L}_1, \ldots, \mathcal{L}_n$ are strong compatible w.r.t. plays (see Definition 8.13);
- $\text{comp}(\mathcal{L}_1, \ldots, \mathcal{L}_n)_{\texttt{plays}} \models_\exists \mathcal{C}hor\downarrow_{\texttt{plays}}$.

Definition 8.15 (Strong conformance). A composition of local models $\text{comp}(\mathcal{L}_1, \ldots, \mathcal{L}_n)_{\texttt{plays}}$ is *strong conformant* with a choreography $\mathcal{C}hor$ iff:

- $\mathcal{L}_1, \ldots, \mathcal{L}_n$ are strong compatible w.r.t. plays (see Definition 8.13);
- $\text{comp}(\mathcal{L}_1, \ldots, \mathcal{L}_n)_{\texttt{plays}} \models_\forall \mathcal{C}hor\downarrow_{\texttt{plays}}$.

It is worth noting the similarity between strong conformance and the notion of a-priori compliance introduced in Definition 8.6. In fact, when verifying strong conformance the composition of local models could be considered as the ConDec model under study, while the choreography could represent (and, indeed, it actually represents) a regulatory model.

The following example discusses the notion of strong and weak compliance on a simple illustrative case, showing that to ensure conformance, it is sufficient that the obtained composition "covers" a part of the choreography.

Example 8.5 (Weak and strong conformance). *Let us consider a simple (fragment of a) choreography involving two roles – a customer and a seller. The choreography states that:*

A. *two possible payment methods are available to the customer (payment by credit card and payment by cash);*

B. *the customer can pay only after having closed the order;*

C. *if the customer pays, then the seller is entitled to send the ordered good and, conversely, a good is sent to the customer only if a payment has been previously done.*

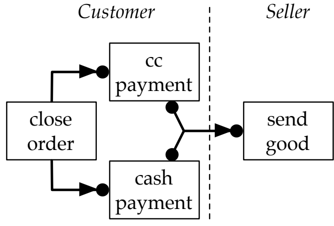
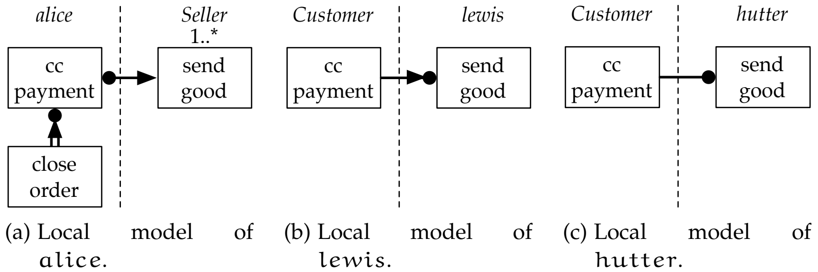


Figure 33: A simple payment choreography.



(a) Local model of alice. (b) Local model of lewis. (c) Local model of hutter.

Figure 34: Three candidate local models (one customer and two sellers) for the payment choreography shown in Figure 33.

Figure 33 shows the ConDec model of the resulting choreography, while Figure 34 depicts three possible local models which can be composed to realize such a choreography. In particular, alice *can play the role of* Customer, *while* hutter *and* lewis *can play the role of* Seller.

Let us first consider the composition obtained by combining the model of alice *with the one of* lewis. *The composition is strong conformant with the choreography:*

- *The choreography allows an open choice on the payment modality, and both local models only deal with payment by credit card.*
- *The combination of the constraints which relate the payment with the delivery of the good in the two local models leads to obtain the following constraint* cc payment •—►• send good, *which is a "specialization" of the choreography one (no choice is present).*
- alice *states that before paying, she wants to close the order, and that between two payments at least one* close order *must be executed; such a constraint is a specialization of the simple* precedence *constraint contained in the choreography.*

The composition obtained by combining the model of alice *with the one of* hutter *is instead not strong conformant. In fact,* hutter *does not impose any temporal ordering between the payment and the delivery of the good. Therefore, it could be possible that the good is sent twice: one time before the payment of alice, and another time afterwards. In other words, the following*

execution trace is supported by the composition: close order $\longrightarrow$ send good $\longrightarrow$ cc payment $\longrightarrow$ send good. *The first execution of the* send good *activity is not preceded by a payment, thus violating a pescription of the choreography.*

However, the composition is weak conformant, because it supports different possible executions which comply with the choreography.

9

PROOF PROCEDURES

Contents

9.1 The SCIFF Proof Procedure **154**
9.1.1 Data Structures and Proof Tree 155
9.1.2 Transitions 157
9.2 Formal Properties of the SCIFF Proof Procedure **163**
9.2.1 Soundness 163
9.2.2 Completeness 163
9.2.3 Termination 164
9.2.4 ConDec Models and Termination of the SCIFF Proof Procedure 165
9.3 The g-SCIFF Proof Procedure **165**
9.3.1 Generation of Intensional Traces 166
9.3.2 Data Structures Revisited 166
9.3.3 Transitions Revisited 167
9.3.4 Comparison of the Proof Procedures 168
9.4 Formal Properties of the g-SCIFF Proof Procedure **169**
9.4.1 Soundness 169
9.4.2 Completeness W.r.t. Generation of Traces 169
9.4.3 Termination 171
9.4.4 ConDec Models and Termination of the SCIFF Proof Procedure 171
9.5 Implementation **171**

This Chapter describes two abductive proof procedures able to reason upon SCIFF (and therefore also CLIMB) specifications[1]:

The `sciff` proof procedure in a nutshell

- the `sciff` proof procedure verifies whether an execution trace *complies* with a SCIFF specification[2], providing a concrete operational framework able to:
  - A. generate positive and negative expectations starting from the trace and the specification;
  - B. check if the generated expectations are fulfilled by the actual occurring events.

1 In the following, we will always take into account CLIMB specifications, but the two proof procedures are able to reason upon specifications relying on the full-SCIFF language.

2 We will use notation `sciff` to denote the SCIFF proof procedure, differentiating it with the SCIFF language and framework.

sciff is able reason upon a complete execution trace, or upon a growing trace, by acquiring and processing dynamically occurring events.

The g-sciff proof procedure in a nutshell

- the g-sciff proof procedure is an extension of sciff dedicated to the verification of properties. It adopts a generative approach, starting from a specification and a property and trying to produce an execution trace which is compliant with the specification and, at the same time, (dis)confirms the property. It is therefore especially suitable for static verification.

Thanks to the mapping from ConDec to CLIMB presented in Chapter 5, these proof procedures will be applied, in the next chapters, to verify ConDec models.

We describe how the two proof procedures work and recall their formal properties. Some details are omitted, because the CLIMB and SCIFF-lite languages are sub-sets of the SCIFF one. For a complete description, the interested reader is referred to [7].

9.1 THE SCIFF PROOF PROCEDURE

sciff is an abductive proof procedure able to verify compliance of execution traces with CLIMB specification. In this respect, it represents the operational counterpart of the declarative semantics described in Section 4.3, and of the notion of compliance in particular (see Definition 4.13 at Page 76). Starting from a (partial) execution trace $\mathcal{T}_i$ and from a CLIMB specification $\mathcal{S}$, sciff is able to dynamically fetch new events and compute abductive explanations for the evolving course of interaction, checking if such abductive explanations are **E**-consistent and fulfilled by the occurred events. Hence, sciff concretely realizes the schema depicted in Figure 17 – Page 76.

Relationships between sciff and the IFF proof procedure

Being the language and declarative semantics of the CLIMB framework closely related with the IFF abductive framework [82], sciff has taken inspiration from the IFF proof procedure. The IFF proof procedure is one of the most well-known proof procedures which combine reasoning with defined predicates together with reasoning with abducible predicates. While IFF is a general abductive proof procedure, sciff is a general abductive proof procedure able to solve the specific problem of compliance verification. In particular, sciff is a substantial extension of the IFF, and adds, in a nutshell, the following features [7]:

- sciff supports the dynamic acquisition of events, i.e., the insertion of new information during the computation;
- sciff supports universally quantified variables in abducibles and quantifier restrictions, in order to properly reason with negative expectations and **E**-consistency;
- sciff supports quantifier restrictions[40] and CLP constraints on variables;

- `sciff` supports the concepts of fulfillment and violation (see Definition 4.12), executing a "hypotheses confirmation" step in which the generated expectations are checked against the actual execution trace.

9.1.1 *Data Structures and Proof Tree*

As in the case of IFF, `sciff` is based on a rewriting system which transforms one node into a successor node or a set of successor nodes by applying *transitions*.

Definition 9.1 (`sciff` node and computed abductive explanation). A `sciff` node is defined by the tuple $\langle R, CS, ps\mathcal{IC}, \mathcal{T}, \Delta_A, \Delta_P, \Delta_F, \Delta_V \rangle$, where:

- R is the resolvent, i.e., a conjunction (of disjunctions) of literals – in CLIMB, literals are positive/negative expectations and predicates defined in the knowledge base of the specification;
- CS is the constraint store, containing CLP constraints and quantifier restrictions;
- $ps\mathcal{IC}$ is a set of partially solved ICs (PSICs), i.e., rules whose body has been partially subject to match with occurred events;
- $\mathcal{T}$ is the currently fetched (partial) execution trace;
- Δ_A is the set of abduced predicates but expectations, which are treated separately[3];
- Δ_P, Δ_F and Δ_V respectively represent the set of *pending*, *fulfilled* and *violated* expectations.

The *abductive explanation computed* in the node is the set Δ of all abduced atoms: $\Delta = \Delta_A \cup \Delta_P \cup \Delta_F \cup \Delta_V$. When at least one element of the node contains $\perp$, then the node is a *failure node*, and the whole tuple can be represented with $\perp$. A failure node does not have successors.

Definition 9.2 (Initial node). Given a CLIMB specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$ and an execution trace $\mathcal{T}$, the *initial node* for the instance $\mathcal{S}_\mathcal{T}$ is $I(\mathcal{S}_\mathcal{T}) = \langle \mathtt{true}, \emptyset, \mathcal{IC}, \mathcal{T}_i, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, i.e., in $I(\mathcal{S}_\mathcal{T})$ the resolvent is $\mathtt{true}$[4], its partially solved ICs are the ICs of the specification, and the computed abductive explanation is the empty set.

Definition 9.3 (Derivation). Given a CLIMB specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$ and an initial execution trace $\mathcal{T}_i$, a *derivation* is a sequence of nodes $I(\mathcal{S}_{\mathcal{T}_i}) \rightarrow N_1 \rightarrow \ldots \rightarrow N_n$, where:

3 in CLIMB, being expectations the only abducible predicates, such a set is always empty.

4 In the general case of SCIFF, the initial resolvent is a goal, expressed in terms of a disjunction (of conjunctions) of expectations. As discussed in Remark 4.1 – Page 70, such a goal can be easily expressed as an IC, and therefore starting with a `true` resolvent does not lead to loose generality.

- nodes $N_1, \dots, N_n$ are obtained by applying the transitions listed in Section 9.1.2;
- N_n is a node in which no transition is applicable, i.e., N_n is a *quiescent* node.

If the initial execution trace is not explicitly mentioned, it is assumed to be the empty trace.

In the general case, for a given node different transitions can be applied by `sciff`. Therefore, starting from an initial node, the rewriting system builds a *proof (OR) tree*, where each leaf node is a quiescent node. In turn, a quiescent node is either the special failure node $\perp$ or a *succcess node*. If at least one leaf success node exists, then `sciff` has a *successful derivation*.

DEFINITION 9.4 (Success node). Given a CLIMB specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$ and a (final) execution trace $\mathcal{T}_f$, a *success node* for $\mathcal{S}_{\mathcal{T}_f}$ has the form $S(\mathcal{S}_{\mathcal{T}_f}) = \langle \mathtt{true}, \mathrm{CS}, \mathrm{ps}\mathcal{IC}, \mathcal{T}_f, \Delta_A, \emptyset, \Delta_F, \emptyset \rangle$, where CS is consisent. $S(\mathcal{S}_{\mathcal{T}_f})$ has a $\mathtt{true}$ resolvent and does not contain pending nor violated expectations.

DEFINITION 9.5 (`sciff` successful derivation). Given a CLIMB specification $\mathcal{S}$, an initial execution trace $\mathcal{T}_i$ and a final execution trace $\mathcal{T}_f$ s.t. $\mathcal{T}_i \subseteq \mathcal{T}_f$, there exists a *successful derivation* for $\mathcal{S}_{\mathcal{T}_f}$ starting from $\mathcal{T}_i$[5] iff the proof tree with root node the initial node $I(\mathcal{S}_{\mathcal{T}_i})$ has at least one leaf success node of the form $S(\mathcal{S}_{\mathcal{T}_f})$. In this case, we write $\mathcal{S}_{\mathcal{T}_i} \vdash_{\Delta}^{\mathcal{T}_f} \mathtt{true}$, where Δ is the computed abductive explanation in $S(\mathcal{S}_{\mathcal{T}_f})$ (as defined in Definition 9.10).

Extraction of (ground) abductive explanations from an intensional computed explanation

It is worth noting that abducibles in Δ may contain variables, i.e., `sciff` computes *intensional abductive explanations* (see Section 4.3). Roughly speaking, variables contained in positive expectations are existentially quantified, whereas variables contained in negative expectations are universally quantified (unless they also appear in a positive expectation). Therefore, the relationship between computed explanations and the (ground) abductive explanations used in the declarative semantics is that:

- An abducible containing universally quantified variables intensionally represents the (possibly infinite) abducibles that are obtained by grounding such variables in all possible ways, while respecting the constraints contained in the constraint store. For example, supposing that the time structure is $(\mathbb{N}, <)$, $\mathbf{EN}(e, T) \wedge T > 9$ intensionally represents the infinite set

$$\{\mathbf{EN}(e, 10), \mathbf{EN}(e, 11), \mathbf{EN}(e, 12), \dots\}$$

5 Without loss of generality, it can be assumed that $\mathcal{T}_i = \emptyset$. As we will see, starting from an empty trace or a trace $\mathcal{T}_i$ has the only difference that a sequence of *happening* transitions must be applied in order to "import" all the events of $\mathcal{T}_i$.

- An abducible containing existentially quantified variables intensionally represents one among the possible abducibles obtained by choosing a grounding which respects the constraints contained in the constraint store[6]. For example, supposing that the time structure is $(\mathbb{N}, <)$, $\mathbf{E}(e, T) \wedge T > 2 \wedge T \leqslant 5$ intensionally represents $\mathbf{E}(e, 3) \vee \mathbf{E}(e, 4) \vee \mathbf{E}(e, 5)$.

9.1.2 *Transitions*

We briefly recall the (most relevant) transitions applied by `sciff`; for a complete technical description of all transitions see [7]. Transitions concretely realize inference rules which are applied on a node generating a set of possible successor nodes, until quiescence is reached. `sciff` adapts all the inference rules developed for IFF [82] and extends them with transitions dedicated to deal with dynamically growing execution traces, and to check **E**-consistency and fulfillment. Furthermore, being CLIMB equipped with CLP constraints, also the transitions of the underlying CLP solver [101] are incorporated.

IFF-inspired transitions

A description of the transitions inherited (and adapted) from the IFF proof procedure follows. Each transition is described supposing its application on node N_k.

PROPAGATION Given a PSIC $\mathbf{H}(E_1, T_1) \wedge Body \rightarrow Head \in ps\mathcal{IC}_k$ and an happened event $\mathbf{H}(E_2, T_2) \in \mathcal{T}_k$ that unifies with $\mathbf{H}(E_1, T_1)$, part of the PSIC can match with this happened event. `sciff` propagates such a possibility by generating a new child node N_{k+1} in which the rule $\mathbf{H}(E_1, T_1) \wedge Body \rightarrow Head$ is copied, obtaining a rule $\mathbf{H}(E_1', T_1') \wedge Body' \rightarrow Head'$[7]; the copy is then modified by removing $\mathbf{H}(E_1', T_1')$ from its body and by inserting the *equality constraint* between the two elements[8]. Therefore,

$$ps\mathcal{IC}_{k+1} = ps\mathcal{IC}_k \cup \{ E_1' = E_2 \wedge T_1' = T_2 \wedge Body' \rightarrow Head' \}$$

The original rule must be maintained in $ps\mathcal{IC}_{k+1}$ because happened events belonging to the body of a PSIC have a universal quantification with scope the entire rule, and a further occurring event able to match with $\mathbf{H}(E_1, T_1)$ would trigger a new, separate propagation on the original rule.

CASE ANALYSIS If the body of a PSIC $\in ps\mathcal{IC}_k$ contains an equality constraint $A = B$, case analysis deals with such an equality constraint generating two child nodes:

---

6 If the constraint solver is (theory) complete [103] (i.e., for each set of constraints c, the solver always returns *true* or *false*, and never *unknown*), then there will always exist, for each success node, a substitution σ' grounding such existential variables. Otherwise, if the solver is incomplete, σ' may not exist. The non-existence of σ' is discovered during the grounding phase, and the result is that the node is marked as a failure node.

7 A copy is obtained by taking into account each constitutive element of the PSIC, renaming its variables.

8 Equality will be then handled by the *case-analysis* transition.

A. in the first node, sciff hypothesizes that the equality $A = B$ holds[9];

B. in the second node, sciff hypothesizes that the equality constraint does not holds, i.e., it assumes $A \neq B$.

Case analysis is applied in a similar way when the body of the PSIC contains a CLP constraint.

LOGICAL EQUIVALENCE When the body of a PSIC $\in ps\mathcal{IC}$ becomes $true$, i.e., PSIC = $true \rightarrow Head$, then a new child node is generated, where PSIC is removed from $ps\mathcal{IC}$ and $Head$ is added to the resolvent:

$$\begin{aligned} ps\mathcal{IC}_{k+1} &= ps\mathcal{IC}_k / \{true \rightarrow Head\} \\ R_{k+1} &= R_k \wedge Head \end{aligned}$$

The resolvent is then rearranged as a disjunction of conjunctions.

UNFOLDING When a predicate p defined in the knowledge base of the specification is encountered by sciff, sciff unfolds it by considering its definitions. sciff could encounter p in two different situations:

A. sciff selects p from the resolvent R_k. In this case, the possible definitions of p are considered in a disjunctive manner. sciff therefore generates a set of candidate successor nodes; in each node, the predicate is substituted with one of its definitions.

B. the body of a PSIC $\in ps\mathcal{IC}$ contains a predicate defined in the knowledge base. Here, all the possible definitions must be simultaneously taken into account, and therefore sciff generates a unique child node which contains as many copies of the IC as the number of definitions of the predicate; each copy substitutes the predicate with one of its definitions.

ABDUCTION When sciff extracts an abducible predicate from R_k, it abduces such a predicate; a new node is generated, where the predicate is moved to the $\Delta_{A\,k+1}$ set if it is a generic abducible, or the predicate is moved to the $\Delta_{P\,k+1}$ set if it is an expectation (in fact, the expectation becomes pending).

SPLITTING If the resolvent of the current node contains a disjunction, i.e., $R_k = A \vee B$, two candidate child nodes N^1_{k+1} and N^2_{k+2} are generated, each one containing only one of the two disjuncts: $R^1_{k+1} = A$ and $R^2_{k+1} = B$.

sciff peculiar transitions

The following transitions are peculiar to sciff, and are added to handle the dynamic acquisition of happened events, CLP constraints, **E**-consistency and fulfillment/violation of expectations:

9 The unification is then handled by the underlying constraint solver, taking into account quantification of the variables and their attached CLP constraints/restrictions.

CLP TRANSITIONS `sciff` inherits all the transitions of CLP [101]. A set of further inference rules is also included, in order to handle universally quantified variables and their interaction with existentially quantified variables, taking into account the involved CLP constraints and quantifier restrictions. Just to cite an example, a disunification $A \neq B$ is replaced with $\perp$ if A is a universally quantified unconstrained variable.

HAPPENING This transition takes a new event $\mathbf{H}(E, T)$ from an external queue and generates a new node in which such a happened event is inserted in the execution trace: $\mathcal{T}_{k+1} = \mathcal{T}_k \cup \{\mathbf{H}(E, T)\}$.

CLOSURE Closure is applicabile when no other transition is applicable, and is used to hypothesize that no further event will occur. Therefore, closure imposes a Closed World Assumption (CWA, [166]) on the execution trace, making possible to evaluate all the expectations which are still pending. In the general case, closure is applied non-deterministically: it generates two child nodes, where the first child imposes the closure hypotheses, and the second child states that the closure hypothesis does not hold. When the execution trace has effectively reached its termination, closure is applied deterministically, and only the first child node is generated. In order to inform `sciff` that the trace is completed, a flag must be set by the user.

TIMES UPDATE This is a special transition which is enabled only by an explicit request from the user, who must be sure that *events occur always in ascending order*, i.e., that each time the *happening* transition is executed starting from a node, the newly inserted happened event is associated to a time which is greater or equal than all the happened events already contained in the execution trace of that node. If this assumption holds, then the time variables associated to pending positive expectations can be updated, stating that they must be greater or equal than the time of the inserted occurred event. In particular, the *times update* transition selects the maximum time contained in the current execution trace. By supposing that such a time is t_{MAX}, the *times update* transition operates as follows: $\forall\ \mathbf{E}(E, T) \in \Delta_{P_k}, CS_{k+1} \ni T \geqslant t_{MAX}$.

E-VIOLATION Violation of a positive expectation can be proven only if there will not be any further event matching the expectation. In particular, let us suppose $\mathbf{E}(E, T) \in \Delta_{P_k}$; $\mathbf{E}(E, T)$ is violated if:

A. The *closure* transition has been previously applied to the derivation N_k belongs to. In fact, *closure* states that no further event will occur, and therefore a still pending expectation cannot be fulfilled anymore.

B. The *times update* transition has been applied, and T is associated to a deadline which has expired.

The *E-violation* transition simply moves the expectation from the set of pending expectations to the set of violated expectations: $\Delta_{P\,k+1} = \Delta_{P\,k} / \{\mathbf{E}(E, T)\}$ and $\Delta_{V\,k+1} = \Delta_{V\,k} \cup \{\mathbf{E}(E, T)\}$

E-FULFILLMENT Let us suppose that $\mathbf{E}(E_1, T_1) \in \Delta_{P\,k}$ and $\mathbf{H}(E_2, T_2) \in \mathcal{T}_k$. The E-fulfillment transition tries to fulfill the expectation with the occurred event. In particular, it generates two child nodes, N^1_{k+1} and N^2_{k+2}.

A. In N^1_{k+1}, `sciff` hypothesizes that the expectation is fulfilled by the happened event:

$$\begin{aligned}\Delta_{P\,k+1} &= \Delta_{P\,k} / \{\mathbf{E}(E_1, T_1)\}\\ \Delta_{F\,k+1} &= \Delta_{F\,k} \cup \{\mathbf{E}(E_1, T_1)\}\\ CS_{k+1} &= CS_k \cup \{E_1 = E_2 \wedge T_1 = T_2\}\end{aligned}$$

B. in N^2_{k+1}, a sort of case-analysis is applied, by stating that the happened event does not match with the expectation, and therefore fulfillment does not hold:

$$\begin{aligned}\Delta_{P\,k+1} &= \Delta_{P\,k}\\ \Delta_{F\,k+1} &= \Delta_{F\,k}\\ CS_{k+1} &= CS_k \cup \{E_1 \neq E_2 \vee T_1 \neq T_2\}\end{aligned}$$

EN-VIOLATION Symmetrically with *E-fulfillment*, when $\mathbf{EN}(E_1, T_1) \in \Delta_{P\,k}$ and $\mathbf{H}(E_2, T_2) \in \mathcal{T}_k$, *EN-violation* generates two child nodes N^1_{k+1} and N^2_{k+1}.

A. In N^1_{k+1}, `sciff` hypothesizes that the happened event matches with the negative expectation, leading to a violation:

$$\begin{aligned}\Delta_{P\,k+1}^{1} &= \Delta_{P\,k} / \{\mathbf{EN}(E_1, T_1)\}\\ \Delta_{V\,k+1}^{1} &= \Delta_{V\,k} \cup \{\mathbf{EN}(E_1, T_1)\}\\ CS^1_{k+1} &= CS_k \cup \{E_1 = E_2 \wedge T_1 = T_2\}\end{aligned}$$

B. in N^2_{k+1}, a sort of case-analysis is applied, by stating that the happened event does not match with the expectation, and therefore that the expectation remains pending:

$$\begin{aligned}\Delta_{P\,k+1}^{2} &= \Delta_{P\,k}\\ \Delta_{V\,k+1}^{2} &= \Delta_{V\,k}\\ CS^2_{k+1} &= CS_k \cup \{E_1 \neq E_2 \vee T_1 \neq T_2\}\end{aligned}$$

The constraint store will then deal with the inserted unification/disunification constraints, taking into account how E_1 and T_1 are quantified. For example, it could be the case that, despite the fact that T_1 belongs to a negative expectation, it is existentially quantified (e.g., because it also appears in a positive expectation). If it is the case, case-analysis is necessary.

Contrarywise, if both E_1 and T_1 are universally quantified, the constraint solver will substitute the two inequalities imposed in CS^2_{k+1} with $\bot$, and therefore the entire node will be declared as a failure node; in this case, violation is deterministic.

EN-FULFILLMENT Symmetrically to E-violation, fulfillment of a negative expectation $\mathbf{EN}(E,T) \in \Delta_{P_k}$ can be ensured

A. After the *closure* transition, if no event in the execution trace matches with the negative expectation[10]; in this case, no further event will occur, and therefore the negative expectation cannot be violated anymore.

B. (If the negative expectation is associated to a deadline) after a *times update* transition, which causes the deadline to expire, provided that none of the previously occurred event matches with the negative expectation.

If one of these two situations holds, *EN-fulfillment* simply moves the negative expectation from the set of pending expectations to the set of fulfilled expectations: $\Delta_{P_{k+1}} = \Delta_{P_k}/\{\mathbf{EN}(E,T)\}$ and $\Delta_{F_{k+1}} = \Delta_{F_k} \cup \{\mathbf{EN}(E,T)\}$.

CONSISTENCY To deal with $\mathbf{E}$-consistency, `sciff` adds the following IC to each specification:

$$\mathbf{E}(X,T) \wedge \mathbf{EN}(X,T) \rightarrow \bot.$$

If two contradictory expectations exist, then `sciff` derives $\bot$, and the current derivation fails. It is worth noting that when a positive and a negative expectation matches with such an IC, case analysis is applied. In this way, `sciff` restricts the domains of the variables contained in the positive expectation so that $\mathbf{E}$-consistency is guaranteed.

Let us consider, for example, that N_k contains two pending expectations $\mathbf{E}(a,T) \wedge T > 4$ and $\mathbf{EN}(a,T_2) \wedge T_2 > 8$. By applying a combination of the *consistency*, *propagation* and *case analysis* transitions, two paths are explored by `sciff`. In the first path, matching between these two expectations and the expectations used in the consistency IC is imposed, leading to a failure node. In the other path, matching is avoided, and the result is that variable T is affected by imposing, in the constraint store, that $T \leqslant 8$: to have an $\mathbf{E}$-consistent explanation, the positive expectation must be fulfilled by an "a" event happening after 4 and before or at 8.

EXAMPLE 9.1 (A `sciff` derivation). *Let us consider the CLIMB specification* $\mathcal{S} = \langle \emptyset, \{IC = \mathbf{H}(a,T) \rightarrow \mathbf{E}(b,T_2) \wedge T_2 > T\}$ *and the execution trace* $\mathcal{T}_i = \{\mathbf{H}(a,5), \mathbf{H}(b,10)\}$ *– we suppose that* $\mathcal{T}_i$ *represents a complete course of interaction, hence the dynamic acquisition of new event occurrences is not exploited. Let us briefly recall how the transitions of* `sciff` *are applied in order to prove that such a trace is compliant with* $\mathcal{S}$ *(for the sake of simplicity, some transitions are omitted):*

10 $\forall E_2 \forall T_2,\ \mathbf{H}(E_2,T_2) \in \mathcal{T}_k \Rightarrow (E_1,T_1)$ does not unify with (E_2,T_2).

1. sciff *starts from* $\mathcal{T}_i$*, i.e., the root node is* $I(\mathcal{S}, \mathcal{T}_i)$.

2. sciff *applies* propagation*, because there is a happened in the trace which can match with the body of* IC*. A new node is generated, in which a copy of* IC *is inserted by imposing the matching between* T *and* 5*:* $\mathsf{ps}\mathcal{IC}_2 = \{IC\} \cup \{T' = 5 \rightarrow \mathbf{E}(b, T'_2) \wedge T'_2 > T\}$

3. *the equality constraint* $T' = 5$ *is imposed, generating a new node* N_3 *in which the constraint store is properly extended, and the body of the second PSIC is consumed:*

$$\begin{aligned} \mathsf{ps}\mathcal{IC}_2 &= \{IC\} \cup \{\mathsf{true} \rightarrow \mathbf{E}(b, T'_2) \wedge T'_2 > 5\} \\ CS_2 &= \{T' = 5\} \end{aligned}$$

   Note that being T *universally quantified with scope the entire clause, the node in which case analysis is applied imposing* $T \neq 5$ *is a failure node.*

4. *Transition* logical equivalence *is applied, moving the head of the PSIC with body* true *to the resolvent; the* CLP *constraint* $T'_2 > 5$ *is then asserted:*

$$\begin{aligned} \mathsf{ps}\mathcal{IC}_3 &= \{IC\} \\ CS_3 &= \{T' = 5 \wedge T'_2 > 5\} \\ R_3 &= \{\mathbf{E}(b, T'_2)\} \end{aligned}$$

5. Abduction *moves the expectation from the resolvent to the set of pending expectations:*

$$\begin{aligned} \mathsf{ps}\mathcal{IC}_4 &= \{IC\} \\ CS_4 &= \{T' = 5 \wedge T'_2 > 5\} \\ R_4 &= \{\mathsf{true}\} \\ \Delta_{P4} &= \{\mathbf{E}(b, T'_2)\} \end{aligned}$$

6. *The second happened event in the trace is able to fulfill the pending expectation.* sciff *therefore takes into account two possibilities, by applying* E-fulfillment*:*

   a) *Fulfillment is effectively executed: constraint* $T'_2 = 10$ *is inserted in the constraint store (note that such an equality constraint is consistent with* $T'_2 > 5$*), and the expectation is moved from the pending to the fulfillment set. Since no further event is present in the external queue, the execution can be declared as* closed*; transition* closure *is then applied deterministically, and a success node is reached, because no expectation is violated and the resolvent is* true*.*

   b) *Fulfillment is avoided: constraint* $T'_2 \neq 10$ *is imposed, waiting for a new happened event able to fulfill the pending expectation. However, there is no further event, and the execution is declared as* closed*. Transitions* closure *and* E-violation *are applied in sequence, because the expectation cannot be fulfilled anymore.*

`sciff` *has therefore a single successful derivation, with an abductive explanation* $\Delta = \{\mathbf{E}(b, 10)\}$.

If the trace given in input would have included also a further event $\mathbf{H}(b, 15)$*, then also the derivation through node* N_{6b} *would have led to a successful derivation, with* $\Delta = \{\mathbf{E}(b, 15)\}$*. In fact, such an extended trace includes two executions of* b*, and both executions are good candidates to fulfill the generated expectation.*

9.2 FORMAL PROPERTIES OF THE SCIFF PROOF PROCEDURE

Three fundamental formal properties have been proven on `sciff`: *soundness*, *completeness* and *termination*. We briefly recall the general definition of such properties, then grounding the analysis on `sciff`, discussing which restrictions are needed on the framework in order to satisfy them. The interested reader may find the proofs of theorems in [7].

9.2.1 *Soundness*

Generally speaking, a logic-based system meets the *soundness* property if its inference rules prove only formulae valid w.r.t. its declarative semantics.

In the context of CLIMB, soundness states that whenever `sciff` has a successful derivation (as defined in Definition 9.5) for $\mathcal{S}_{\mathcal{T}}$, then $\mathcal{T}$ is compliant with $\mathcal{S}$ according to CLIMB declarative semantics (Definition 4.13).

THEOREM 9.1 (Soundness of `sciff`). *Given an arbitrary CLIMB instance* $\mathcal{S}_{\mathcal{T}}$,

$$\mathcal{S}_{\emptyset} \vdash_{\Delta}^{\mathcal{T}} \mathrm{true} \Rightarrow \textsc{compliant}(\mathcal{S}_{\mathcal{T}})$$

Proof. See [7]. □

9.2.2 *Completeness*

A logic-based system is *complete* if its inference rules are able to prove all formulae valid w.r.t. its declarative semantics.

In the CLIMB setting, completeness states that for each compliant CLIMB instance $\mathcal{S}_{\mathcal{T}}$, `sciff` has a successful derivation for $\mathcal{S}$ starting from the empty trace and leading to $\mathcal{T}$.

THEOREM 9.2 (Completeness of `sciff`). *Given an arbitrary CLIMB specification* $\mathcal{S}$,

$$\forall\, \mathcal{T},\ \textsc{compliant}(\mathcal{S}_{\mathcal{T}}) \Rightarrow \mathcal{S}_{\emptyset} \vdash_{\Delta}^{\mathcal{T}} \mathrm{true}$$

Proof. See [7]. □

9.2.3 *Termination*

A logic-based system meets the *termination* property every derivation produced by applying its inference rules is finite.

Semi-decidability of FOL

First Order Logic (FOL) does not always guarantee termination when proving if a certain formula is entailed by a theory. In particular, given a provable sentence, a successful derivation will be always provided in finite time, but when the sentence is not provable, the application of the inference rules may run forever. This issue is known as *semi-decidability*. Fortunately, decidable fragments of FOL, i.e., fragments which guarantee termination, can be found.

Since CLIMB belongs to the first order setting, it suffers from the semi-decidability issue as well. Therefore, suitable restrictions on the CLIMB language are needed to guarantee termination.

Termination and acyclicity conditions

Such restrictions are an extension of the classical *acyclicity conditions* imposed on logic programs [105] to ensure termination. Roughly speaking, acyclicity imposes the absence of loops in a logic program, i.e., it requires that if predicate p is defined in terms of q, then q cannot be defined by means of p. Acyclicity conditions need to be extended to deal also with Integrity Constraint (IC), as described by Xanthakos in [205].

We briefly recall such extended acyclicity conditions, which have been defined for IFF, and therefore also apply to the SCIFF framework[11].

Definition 9.6 (Level mapping [205]). A *Level mapping* for a logic program $\mathcal{P}$ is a function which maps $\bot$ to 0 and each ground atoms $\in \mathfrak{B}^{\mathcal{P}}$ to a positive integer, i.e., $|\cdot| : \mathfrak{B}^{\mathcal{P}} \to \mathbb{N}/\{0\}$, where $\mathfrak{B}^{\mathcal{P}}$ is the Herbrand base of $\mathcal{P}$. For $A \in \mathfrak{B}^{\mathcal{P}}$, $|A|$ denotes the *level* of A.

Definition 9.7 (Boundedness [205]). Given a level mapping $|\cdot|$, a literal L is *bounded w.r.t.* $|\cdot|$ iff $|\cdot|$ is bounded on the set of ground instances of L. In this case, we assume

$$|L| \triangleq \max\{|L_g| \text{ s.t. } L_g \text{ is a ground instance of } L\}$$

Definition 9.8 (Acyclic knowledge base [205]). Given a CLIMB knowledge base $\mathcal{KB}$, a clause $C \in \mathcal{KB}$ is *acyclic w.r.t. a level mapping* $|\cdot|$ if, for every ground instance $H \leftarrow B_1 \wedge \ldots \wedge B_n$ of C, it holds that $\forall\, i \in [1, \ldots, n]$, $|H| > |B_i|$. The entire $\mathcal{KB}$ is *acyclic w.r.t.* $|\cdot|$ if all its clauses are. $\mathcal{KB}$ is called *acyclic* if it is acyclic w.r.t. some level mapping.

Definition 9.9 (Acyclic CLIMB specification). A CLIMB specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$ is acyclic w.r.t. a level mapping $|\cdot|$ iff:

a. $\mathcal{KB}$ is acyclic w.r.t. $|\cdot|$;

11 For the sake of simplicity, we do not discuss here the case of negated literals; the interested reader is referred to [205].

B. each IC $\in \mathcal{IC}$ is acyclic w.r.t. $|\cdot|$; an IC is acyclic w.r.t. $|\cdot|$ iff for each ground instance $B_1 \wedge \ldots \wedge B_n \rightarrow H_1 \vee \ldots \vee H_m$, it holds that $\forall\, i \in [1, \ldots, n]\ \forall\, j \in [1, \ldots, m]$, $|B_i| > |H_j|$[12].

$\mathcal{S}$ is called *acyclic* if it is acyclic w.r.t. some level mapping.

The following theorem states that `sciff` is guaranteed to terminate the computation if the specification under study is acyclic and bounded.

THEOREM 9.3 (Termination of `sciff`). *Given a CLIMB instance $\mathcal{S}_{\mathcal{T}}$, if $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$ is acyclic and all the literals occurring in $\mathcal{KB}$ and $\mathcal{IC}$ are bounded, then every* `sciff` *derivation for $\mathcal{S}_{\mathcal{T}}$ is finite.*

Proof. See [7]. □

9.2.4 *ConDec Models and Termination of the SCIFF Proof Procedure*

Let us now consider the CLIMB specifications obtained by translating ConDec models, i.e., obtained by applying the t_{CLIMB} function introduced in Section 5.1. An important result is that `sciff` is guaranteed to terminate for such specifications.

THEOREM 9.4 (`sciff` terminates when reasoning upon ConDec specifications). *Given a ConDec model $\mathcal{CM}$,* `sciff` *is guaranteed to terminate when reasoning upon $t_{CLIMB}(\mathcal{CM})$.*

Proof. We prove that $t_{CLIMB}(\mathcal{CM}) = \langle \emptyset, \mathcal{IC} \rangle$ is acyclic. Since the knowledge base obtained by applying t_{CLIMB} is always empty (see Definition 5.1 – Page 84), it is always acyclic. Furthermore, all ICs belonging to $\mathcal{IC}$ contain only happened events in their body, and expectations in their head. Hence, the following level mapping can be established:

A. for each ground happened event $\mathbf{H}(e,t)$, $|\mathbf{H}(e,t)| = 2$;

B. for each ground positive expectation $\mathbf{E}(e,t)$, $|\mathbf{E}(e,t)| = 1$;

C. for each ground negative expectation $\mathbf{EN}(e,t)$, $|\mathbf{EN}(e,t)| = 1$.

From Definitions 9.7 and 9.9, $t_{CLIMB}(\mathcal{CM})$ is acyclic and bounded w.r.t. $|\cdot|$ and, from Theorem 9.3, termination is therefore guaranteed. □

9.3 THE G-SCIFF PROOF PROCEDURE

`g-sciff` is an abductive proof procedure specifically dedicated to verifying entailment of properties by SCIFF specifications. `g-sciff` relies on the data structures and transitions of `sciff`, extending them with a transition which transforms positive pending expectations into happened

12 If B_i is a predicate defined inside $\mathcal{KB}$ then also all the (ground) atoms appearing in its definition must have a level mapping greater than the one of all H_j.

events. In this way, given a CLIMB specification, `g-sciff` is able to *generate* execution traces compliant with the specification. We will describe how `g-sciff` concretely realizes this generative approach, delegating the discussion related to the verification of properties (in the ConDec setting) to the next chapter.

9.3.1 *Generation of Intensional Traces*

`g-sciff` is able to generate execution traces because *it considers happened event as abducibles*. In this way, abduction is used to simulate execution traces compliant with the specification under study.

Generative behaviour of `g-sciff`

To concretely realize such a feature, `g-sciff` incapsulates in a dedicated transition the following behaviour: "if there is a pending positive expectation, then abduce a corresponding happened event able to fulfill the expectation". Such abduced happened event is treated as a "normal" happened event, and it can therefore have an impact on the specification under study, matching with happened events contained in the body of its ICs and causing them triggering. Triggered ICs, in turn, will generate new expectations, leading to a new iteration of the generative process. In a nutshell, in each step `g-sciff` takes into account the current execution trace and "simulates by abduction" the occurrence of new events according to the ICs of the specification.

Intensional execution traces

Since pending positive expectations could contain variables (e.g., node N_2 of Example 9.1 has a pending positive expectation with a variable time), when a corresponding happened event is generated by `g-sciff`, it will contain variables as well. In this respect, execution traces generated by `g-sciff` are *intensional*: they represent classes of (ground) execution traces compliant with the specification under study.

In the general case, also an initial execution trace $\mathcal{T}_i$ could be provided as input for `g-sciff`: the proof procedure will try to extend $\mathcal{T}_i$ in order to comply with the specification under study.

9.3.2 *Data Structures Revisited*

The data structures and the concepts of initial and success node are identical to the ones of `sciff`, defined in Section 9.1.1. The only difference is that in the case of `g-sciff`, also happened event can be abduced, hence the definition of computed explanation must be revised accordingly.

DEFINITION 9.10 (`g-sciff` node and computed abductive explanation). A `g-sciff` node is defined by the tuple $\langle R, CS, ps\mathcal{IC}, \mathcal{T}, \Delta_A, \Delta_P, \Delta_F, \Delta_V \rangle$, where:

- R is the resolvent, i.e., a conjunction (of disjunctions) of literals – in CLIMB, literals are positive/negative expectations and predicates defined in the knowledge base of the specification;
- CS is the constraint store, containing CLP constraints and quantifier restrictions;

- $ps\mathcal{IC}$ is a set of partially solved ICs, i.e., rules whose body has been partially subject to match with occurred events;
- $\mathcal{T}$ is the generated execution trace, which contains the initial trace $\mathcal{T}_i$ given in the initial node;
- Δ_A is the set of abduced predicates but expectations, which are treated separately[13];
- Δ_P, Δ_F and Δ_V respectively represent the set of *pending*, *fulfilled* and *violated* expectations.

The *abductive explanation computed* in the node is the set Δ of all abduced atoms: $\Delta = \Delta_A \cup \Delta_P \cup \Delta_F \cup \Delta_V \cup (\mathcal{T}/\mathcal{T}_i)$.

DEFINITION 9.11 (g-sciff successful derivation). Given a CLIMB specification $\mathcal{S}$ and an initial execution trace $\mathcal{T}_i$, there exists a g-sciff *successful derivation* for $\mathcal{S}$ starting from $\mathcal{T}_i$ iff the proof tree with root node the initial node $I(\mathcal{S}_{\mathcal{T}_i})$ has at least one leaf success node. In this case, we write $\mathcal{S}_{\mathcal{T}_i} \vdash_{g\,\Delta}^{\mathcal{T}_f} \mathtt{true}$, where Δ is the computed abductive explanation in a success node (as defined in Definition 9.10) and $\mathcal{T}_f/\mathcal{T}_i \subseteq \Delta$ is the set of (intensional) happened events generated by g-sciff during the derivation.

Extraction of a ground execution trace

A ground compliant execution trace can be simply extracted from a success node by choosing a grounding for all the contained variables, s.t. all the involved CLP constraints are respected[14]. This is in accordance with the grounding of a computed explanation described in Section 9.1.1.

9.3.3 *Transitions Revisited*

sciff transitions are modified as follows:

HAPPENING IS REMOVED because dynamic acquisition of external occurred events is not needed anymore. Indeed, an initial execution trace could be provided to g-sciff at its start-up, but then the proof procedure works autonomously, trying to extend such an initial trace in order to make it compliant with the specification.

FULFILLER IS INSERTED to concretely implement the generative rule stating that "if an expectation cannot be fulfilled by a happened event in the current execution trace, then the trace is extended by generating a happened event able to fulfill the expectation". More specifically, *fulfiller* is applicabile in node N_k only if:

- $\Delta_{P\,k} \neq \emptyset$;

13 Also in the case of g-sciff, when dealing with CLIMB specifications such a set is always empty, because only expectations and happened events are abducible.
14 Indeed, when an happened event is abduced, its variables are existentially quantified

- *E-fulfillment* cannot be applied;
- the *closure* transition has not been applied before in the derivation.

Such a situation exactly represents the case in which there is a pending expectation which cannot be fulfilled, and new happened events can still be generated in order to fulfill it [15]. Checking whether *E-fulfillment* can be applied is a test executed for efficiency reasons: if *E-fulfillment* is applicable, there is no need to generate a happened event, because the execution trace already contains a good candidate.

When applied to node N_k, *fulfiller* extracts a pending expectation $\mathbf{E}(E,T)$ from the Δ_{Pk} set, moving it to the set of fulfilled expectations and inserting in the execution trace a corresponding happened event $\mathbf{H}(E,T)$:

$$\begin{aligned} \Delta_{Pk+1} &= \Delta_{Pk}/\{\mathbf{E}(E,T)\} \\ \Delta_{Fk+1} &= \Delta_{Fk} \cup \{\mathbf{E}(E,T)\} \\ \mathcal{T}_{k+1} &= \mathcal{T}_k \cup \{\mathbf{H}(E,T)\} \end{aligned}$$

EXAMPLE 9.2 (A `g-sciff` derivation). *Let us consider again Example 9.1, but using* `g-sciff` *as a reasoning engine. The first derivation is equal to the derivation of* `sciff` *starting from the initial node and leading to node* N_{6a}. *The difference between the two proof procedures is in node* N_{6b}*: while* `sciff` *states that the expectation* $\mathbf{E}(b,T'_2)$ *cannot be fulfilled if* $T'_2 \neq 10$, `g-sciff` *applies the* fulfiller *transition, generating a new node* N_{7b} *in which* $\mathbf{H}(b,T'_2)$ *is abduced and inserted in the execution trace, and* $\mathbf{E}(b,T'_2)$ *is declared fulfilled.* N_{7b} *is then a success node.*

The answers provided by `g-sciff` *mean that the initial execution trace* $\{\mathbf{H}(a,5), \mathbf{H}(b,10)\}$ *is compliant (node* N_{6a}*), or that a further* b *at a time different than* 10 *and greater than* 5 *is required to make the initial trace compliant (node* N_{7b}*). In this latter case, the computed explanation is* $\Delta = \{\mathbf{E}(b,T'_2),\mathbf{H}(b,T'_2\}$ *and the intensional generated execution trace is* $\{\mathbf{H}(a,5), \mathbf{H}(b,10), \mathbf{H}(b,T'_2)\}$*, with* $T'_2 > 5 \wedge T'_2 \neq 10$ *(these CLP constraints are contained in the constraint store of node* N_{7b}*). Each execution trace grounding* T'_2 *at a value which is greater than* 5 *and different than* 10 *is therefore compliant.*

9.3.4 *Comparison of the Proof Procedures*

`g-sciff` extends `sciff` because it abduces happened events, following the rule stating that each positive expectation must have a corresponding happened event. Since `sciff` is a general abductive proof procedure, it is able to abduce happened events as well, provided that the specification under study declares **H** as an abducible predicate. This is the

15 If *closure* has been applied before, then the application of *fulfiller* generate a $\perp$ child node: it tries to extend the trace with a new happened event, but *closure* forbids such a behaviour.

case of SCIFF-lite specifications, which employ happened events as abducibles in order to deal with composite events (see Section 4.2.6 – Page 66).

Given a CLIMB specification $\mathcal{S}$, `g-sciff` computations on $\mathcal{S}$ are, from a theoretical point of view, equivalent to `sciff` computations over the "generative" SCIFF-lite version of $\mathcal{S}$.

DEFINITION 9.12 (Generative extension). Given a CLIMB specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$, its SCIFF-lite *generative extension* is:

$$\mathsf{gen}(\mathcal{S}) = \langle \mathcal{KB}, \{\mathbf{E}, \mathbf{EN}, \mathbf{H}\}, \mathcal{IC} \cup \{\mathbf{E}(E,T) \rightarrow \mathbf{H}(E,T)\} \rangle$$

However, from a practical point of view `g-sciff` performs on $\mathcal{S}$ much better than `sciff` performs on $\mathsf{gen}(\mathcal{S})$, because of the efficient implementation of the *fulfiller* transition vs the general rule $\mathbf{E}(E,T) \rightarrow \mathbf{H}(E,T)$.

9.4 FORMAL PROPERTIES OF THE G-SCIFF PROOF PROCEDURE

Formal properties of `g-sciff` are investigated by relying on the results given for `sciff`.

9.4.1 *Soundness*

Soundness of `g-sciff` states that each trace generated by `g-sciff` starting from a specification and an initial trace is compliant with the specification, according to CLIMB declarative semantics.

THEOREM 9.5 (Soundness of `g-sciff`). *For each CLIMB specification $\mathcal{S}$ and for each (initial) trace $\mathcal{T}_i$:*

$$\forall\, \Delta, \mathcal{S}_{\mathcal{T}_i} \vdash_{g\,\Delta}^{\mathcal{T}_f} \mathtt{true} \Rightarrow \textsc{compliant}\left(\mathcal{S}_{\mathcal{T}_f}\right)$$

Proof. Soundness of `g-sciff` can be reduced to soundness of `sciff` by considering the generative extension of $\mathcal{S}$ (as defined in Definition 9.12). □

9.4.2 *Completeness W.r.t. Generation of Traces*

The completeness result shown in Theorem 9.2 for `sciff` trivially holds also for `g-sciff`. In fact, for an arbitrary CLIMB specification $\mathcal{S}$ and for each trace $\mathcal{T}$ compliant with that specification, a `g-sciff` derivation for $\mathcal{S}$ starting with $\mathcal{T}$ as the initial execution trace corresponds exactly to a `sciff` derivation: being $\mathcal{T}$ compliant, the *fulfiller* transition is never employed, because all the generated positive expectations can be fulfilled by happened events contained in $\mathcal{T}$. Formally, it holds that, given an arbitrary CLIMB specification $\mathcal{S}$:

$$\forall\, \mathcal{T},\ \textsc{compliant}(\mathcal{S}_{\mathcal{T}}) \Rightarrow \mathcal{S}_{\mathcal{T}} \vdash_{g\,\Delta}^{\mathcal{T}} \mathtt{true}$$

Completeness w.r.t. trace generation

However, in the context of `g-sciff` a more interesting notion of com-

pleteness concerns the generative nature of the proof. In this respect, completeness would state that, given a CLIMB specification $\mathcal{S}$ and an initial execution trace $\mathcal{T}_i$, g-sciff is able to generate *all* the possible execution traces which contain $\mathcal{T}_i$ and are compliant with $\mathcal{S}$:

$$\forall\, \mathcal{T}_i\ \forall\, \mathcal{T}_f \supseteq \mathcal{T}_i,\ \text{COMPLIANT}\left(\mathcal{S}_{\mathcal{T}_f}\right) \Rightarrow \mathcal{S}_{\mathcal{T}_i} \vdash_{g\Delta}^{\mathcal{T}_f} \text{true}$$

We will refer to this notion of completeness as *completeness w.r.t. trace generation*.

g-sciff does not meet such a completeness property: since its generative approach is driven by the ICs of $\mathcal{S}$ and by the initial execution trace, only a sub-set of these traces will be found.

EXAMPLE 9.3 (g-sciff is not complete w.r.t. trace generation). *As shown in Example 9.2, given the CLIMB specification* $\mathcal{S} = \langle \emptyset, \{\mathbf{H}(a, T) \rightarrow \mathbf{E}(b, T_2) \wedge T_2 > T\}\rangle$ *and the (initial) execution trace* $\mathcal{T}_i = \{\mathbf{H}(a, 5), \mathbf{H}(b, 10)\}$, g-sciff *intentionally computes the following infinite compliant execution traces:*

$\{\mathbf{H}(a, 5), \mathbf{H}(b, 10)\}$
$\{\mathbf{H}(a, 5), \mathbf{H}(b, 6), \mathbf{H}(b, 10)\}$
$\{\mathbf{H}(a, 5), \mathbf{H}(b, 7), \mathbf{H}(b, 10)\}$
...
$\{\mathbf{H}(a, 5), \mathbf{H}(b, 10), \mathbf{H}(b, 11)\}$
...

However, many other execution traces compliant with $\mathcal{S}$ *and including* $\mathcal{T}_i$ *exist, such as for example:*

$\{\mathbf{H}(a, 2), \mathbf{H}(a, 5), \mathbf{H}(a, 6), \mathbf{H}(b, 10)\}$
$\{\mathbf{H}(a, 2), \mathbf{H}(b, 4), \mathbf{H}(a, 5), \mathbf{H}(b, 10), \mathbf{H}(b, 12)\}$

Let us now consider the same specification $\mathcal{S}$*, but with an empty initial execution trace (*$\mathcal{T}_i = \emptyset$*). In this case, there is only one successful derivation produced by* g-sciff*, in which the final execution trace is the empty trace. In fact, the behaviour of* g-sciff *is driven by the initial trace and by the IC of* $\mathcal{S}$*: when the initial trace does not contain at least one execution of* a*, the IC does not trigger, no expectation is generated and no* fulfiller *transition is applied.*

Weak completeness w.r.t. trace generation

As shown by the example, g-sciff does not guarantee completeness w.r.t. trace generation. Nevertheless, it is guaranteed that if an initial execution trace can be extended to obtain a compliant execution trace, then g-sciff has at least one successful derivation; we call this property *weak completeness w.r.t. trace generation*. This is a key property: as we will see in the next chapter, g-sciff will be employed to verify whether a given specification is consistent, i.e., admits at least one compliant execution; weak completeness w.r.t. trace generation guarantees that g-sciff handles such an issue in a complete way.

THEOREM 9.6 (Weak completeness w.r.t. trace generation). *Given an arbitrary CLIMB specification $\mathcal{S}$:*

$$\forall\, \mathcal{T}_i,\ \left(\exists\, \mathcal{T}_f \supseteq \mathcal{T}_i,\ \text{COMPLIANT}\left(\mathcal{S}_{\mathcal{T}_f}\right)\right) \Rightarrow \left(\exists\, \Delta,\ \mathcal{S}_{\emptyset} \mathrel{\vdash^{\mathcal{T}'_f}_{g\,\Delta}} \mathtt{true}\right)$$

Proof. A compliant execution trace extending $\mathcal{T}_i$ can be found iff $\mathcal{S}_{\mathcal{T}_i}$ admits at least one abductive explanation which is **E**-consistent. `g-sciff` therefore starts from $\mathcal{S}_{\mathcal{T}_i}$ and computes an (intensional) **E**-consistent abductive explanation (in this first part of the computation, it behaves exactly as `sciff`). In the general case, such an explanation is not $\mathcal{T}_i$-fulfilled: further event occurrences are needed to comply with $\mathcal{S}$. `g-sciff` then fulfills such pending positive expectations by repeatedly applying the *fulfiller* transition, until a success node is reached. □

9.4.3 *Termination*

Termination of `g-sciff` is reduced to termination of `sciff` via the application of the generative extension of a specification (Definition 9.12).

THEOREM 9.7 (Termination of `g-sciff`). *Given a CLIMB specification $\mathcal{S}$, if* $\mathsf{gen}(\mathcal{S})$ *is acyclic and all the literals occurring in it are bounded, then every* `g-sciff` *derivation for $\mathcal{S}$ starting from an arbitrary execution trace is finite.*

Proof. A `g-sciff` derivation for $\mathcal{S}$ starting from an initial trace $\mathcal{T}_i$ corresponds to a `sciff` derivation for $\mathsf{gen}(\mathcal{S})_{\mathcal{T}_i}$, for which, under the hypotheses, termination is guaranteed by Theorem 9.3. □

9.4.4 *ConDec Models and Termination of the SCIFF Proof Procedure*

Acyclicity and boundedness conditions imposed on CLIMB specifications in order to guarantee `g-sciff` termination are much more restrictive than conditions required for `sciff`.

Interaction between the generative rule and level mappings

While Theorem 9.4 guarantees termination for all CLIMB formalizations of ConDec models, for `g-sciff` this is no more the case. In fact, ConDec formalizations must be augmented with the generative rule $\mathbf{E}(E,T) \rightarrow \mathbf{H}(E,T)$, which must be taken into account when defining a level mapping. Since the generative rule contains a (general) expectation in the body and a corresponding happened event in the head, a suitable level mapping cannot be defined for all such specifications, but must be adapted to each specific case.

Chapter 10 will introduce examples of ConDec model for which `g-sciff` does not terminate, discussing how such an issue can be overcome.

9.5 IMPLEMENTATION

Search strategy

`sciff` and `g-sciff` defines a set of transitions for building proof trees, leaving the search strategy to be defined at implementation level. The

basic implementation is based on a *depth-first* strategy. This choice, enabling us to tailor the implementation for the built-in computational features of Prolog, supports a simple and efficient implementation of the proof-procedure.

Adopted technologies

The proof procedures are implemented in SICStus 4[16] and its Constraint Handling Rules (CHR) library [80], and they are freely available[17]. The Constraint Handling Rules (CHR) library is used to implement the transitions of the proof procedures. The data structures are implemented as CHR constraints, so the transitions can be straightforwardly implemented as CHR rules.

CLP solvers

The implementation relies on constraint solvers to handle CLP constraints and quantifier restrictions. Both the CLP($\mathcal{FD}$) [70] and the CLP($\mathcal{R}$) [102] solvers embedded in SICStus have been integrated with the proof procedures. The user can thus choose the most suitable solver for the application at hand, which is an important issue in practice. It is well known, in fact, that no solver dominates the other, and we measured, in different applications, orders of magnitude of improvements by switching solver. Moreover, some domains require discrete time, other dense time. In the following experimentations we will report the results obtained with the CLP($\mathcal{R}$) solver, which is based on the simplex algorithm, and features a complete propagation of linear constraints.

16 http://www.sics.se/sicstus.html
17 http://www.lia.deis.unibo.it/research/sciff/

10

STATIC VERIFICATION OF CONDEC MODELS WITH G-SCIFF

Contents

10.1 Existential and Universal Entailment in CLIMB **174**
- 10.1.1 Specification of Properties with ConDec 174
- 10.1.2 Formalizing Existential and Universal Entailment 175

10.2 Verification of Existential Properties With g-SCIFF **176**
- 10.2.1 Conflict-freedom Checking Via g-SCIFF 176
- 10.2.2 Existential Entailment with g-SCIFF 177

10.3 Verification of Universal Properties With g-SCIFF **178**
- 10.3.1 Complementing Integrity Constraints 178
- 10.3.2 Reduction of Universal Entailment to Existential Entailment 179

10.4 ConDec Loops and Termination Issues **181**
- 10.4.1 Reformulation of ConDec relation constraints 183
- 10.4.2 Unbounded Specifications and Looping ConDec Models 185

10.5 Pre-processing of ConDec Models and Loop Detection **188**
- 10.5.1 Transformation of ConDec Models to AND/OR Graphs 188
- 10.5.2 Detection of $\wedge$- and $\vee$-loops 189
- 10.5.3 Pre-Processing Procedure 192

10.6 Dealing With an Infinite Number of Finite Derivations **195**
- 10.6.1 Succession Constraints and Infinite Branching Proof Trees 195
- 10.6.2 Solving the Infinite Branches Anomaly 197

In this Chapter we describe how the `g-sciff` proof procedure can be adopted for the static verification of ConDec models, respecting the five desiderata described in Section 8.1.

In particular, the first part of the Chapter demonstrates how existential and universal properties, as defined in Section 8.3.1, can be expressed in CLIMB and verified with `g-sciff`. Thanks to this possibility, `g-sciff` can be exploited to deal with all the static verification tasks introduced in Chapter 8, ranging from discovery of dead activities to checking composite models and conformance to a choreography. The second part of the Chapter is instead devoted to deal with termination issues; as sketched in Section 9.4.4, termination of `g-sciff` cannot be

generally guaranteed when reasoning upon CLIMB specifications, and therefore an ad-hoc solution for ConDec must be provided.

10.1 EXISTENTIAL AND UNIVERSAL ENTAILMENT IN CLIMB

In the CLIMB framework, properties are represented with the same language used for specifying models, i.e. by means of ICs. Verification of existential and universal properties is achieved by suitably quantifying on execution traces and combining the concepts of compliance with the specification and with the property.

10.1.1 *Specification of Properties with ConDec*

A CLIMB property is simply a set of CLIMB ICs.

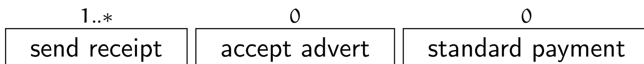
DEFINITION 10.1 (CLIMB property). A CLIMB *property* Ψ is a set of CLIMB ICs: $\Psi = \bigcup_i IC_i$. A CLIMB property is *simple* if it is expressed by a single IC, i.e., $i = 1$.

Expressing properties in ConDec

Since all the ConDec constraints are translatable to CLIMB IC, then also properties can be specified by means of ConDec models. Obviously, the CLIMB language is more expressive than the fragment needed to formalize ConDec constraints; however, specification of properties in ConDec takes advantage from the fact that also non-IT savvy can easily capture the intended requirements. In the following, we will consider ConDec as a graphical language to specify both interaction models and properties; all the provided results hold for general CLIMB specifications and properties.

Let us for example consider the four queries introduced in Section 8.3.4 to verify an order&payment protocol, interpreting them as ConDec models:

- Query 8.1, expressing that send receipt is a dead activity, can be represented by stating that universal entailment w.r.t. the ConDec model [0: send receipt] must be guaranteed.
- Query 8.2, checking whether a transaction can be completed s.t. the customer does not accept ads and the seller does not offer standard payment, can be represented by the ConDec model

  which must be existentially entailed. The `1..*` cardinality constraint models the correct completion of a transaction, where the concept of "correct completion" is related to the emission of a receipt, whereas the two `0` cardinality constraints are used to express that the customer refuses ads and that standard payment is not available.

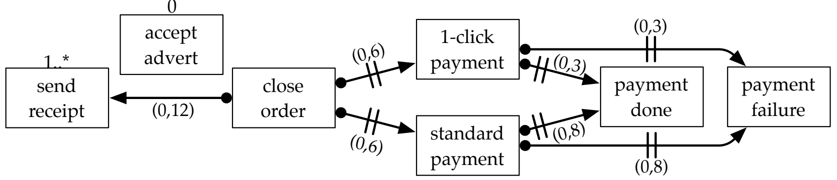


Figure 35: Representation of Query 8.4 – introduced in Section 8.3.4 – with extended ConDec.

- Query 8.3 is a universal query, which states that each execution involving 1-click payment must also guarantee that a receipt is sent only if the customer has previously accepted ads. It can be rephrased by saying that, for each execution instance, either
  - 1-click payment is not involved: [0: 1-click payment]
  - the precedence relationship between send receipt and accept advert is guaranteed: [accept advert] ——▶● [send receipt]
- Query 8.4 is an existential query involving latency times among activities. It can be easily expressed using the extended ConDec notation proposed in Section 6.1 to capture quantitative time constraints. The resulting model is shown in Figure 35. Note that [close order] ●—(0,12)—▶ [send receipt] is used to model the deadline by which the completion of the transaction is desired, while negation response constraints are used to represent latencies. For example, [close order] ●—‖—(0,6)—▶ [1-click payment] states that if the order is closed at a certain time T, then it is possible to execute 1-click payment only from time $T + 6$.

The first three models can be translated to CLIMB by applying the t_{CLIMB} or the $t^{\circlearrowright}_{CLIMB}$ functions[1] (depending on the model that must be verified, a translation with qualitative or quantitative time could be chosen). The last model contains quantitative temporal constraints, hence it can be translated to CLIMB only by means of the $t^{\circlearrowright}_{CLIMB}$ function. In the remainder of this Chapter, we will always consider simple ConDec model and the basic translation function t_{CLIMB}, but all the provided results also hold for extended ConDec models, and for the $t^{\circlearrowright}_{CLIMB}$ function in particular.

10.1.2 *Formalizing Existential and Universal Entailment*

The formalization of ∃- and ∀-entailment in CLIMB follows straightforwardly from the natural language characterization given in Section 8.3.1, using the formal notion of compliance provided by CLIMB (see Definition 4.13).

1 t_{CLIMB} has been defined in Section 5.1, while $t^{\circlearrowright}_{CLIMB}$ has been defined in Section 6.1.2.

∃-entailment in CLIMB

A CLIMB specification $\mathcal{S}$ ∃-entails a CLIMB property if there exists at least one execution trace which is at the same time compliant with $\mathcal{S}$ and the property. Since the property is constituted by a set of ICs but compliance is referred to an entire specification, when establishing compliance we assume that the property is embdedded inside a CLIMB specification such that the same knowledge base of $\mathcal{S}$ is used.

Definition 10.2 (∃-entailment). A CLIMB specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$ ∃-entails a CLIMB property Ψ ($\mathcal{S} \models_\exists \Psi$) iff:

$$\exists\ \mathcal{T}\ \textsc{compliant}\,(\mathcal{S}_\mathcal{T}) \wedge \textsc{compliant}\,(\langle \mathcal{KB}, \Psi \rangle_\mathcal{T})$$

$\mathcal{S}$ ∃-violates Ψ ($\mathcal{S} \not\models_\exists \Psi$) iff:

$$\nexists\ \mathcal{T}\ \textsc{compliant}\,(\mathcal{S}_\mathcal{T}) \wedge \textsc{compliant}\,(\langle \mathcal{KB}, \Psi \rangle_\mathcal{T})$$

∀-entailment in CLIMB

Similarly, a CLIMB specification ∀-entails a CLIMB property if every execution trace compliant with the specification is compliant with the property as well.

Definition 10.3 (∀-entailment). A CLIMB specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$ ∀-entails a CLIMB property Ψ ($\mathcal{S} \models_\forall \Psi$) iff:

$$\forall\ \mathcal{T},\ \textsc{compliant}\,(\mathcal{S}_\mathcal{T}) \Rightarrow \textsc{compliant}\,(\langle \mathcal{KB}, \Psi \rangle_\mathcal{T})$$

$\mathcal{S}$ ∀-violates Ψ ($\mathcal{S} \not\models_\forall \Psi$) iff:

$$\exists\ \mathcal{T},\ \textsc{compliant}\,(\mathcal{S}_\mathcal{T}) \wedge \neg\textsc{compliant}\,(\langle \mathcal{KB}, \Psi \rangle_\mathcal{T})$$

10.2 VERIFICATION OF EXISTENTIAL PROPERTIES WITH G-SCIFF

As discussed in Section 9.3, the `g-sciff` proof procedure is specifically dedicated to the static verification of CLIMB specifications: it realizes a "simulation by abduction" approach trying to generate compliant execution traces starting from an initial execution trace and the ICs of the specification.

We now exploit `g-sciff` to verify existential properties on ConDec models. To deal with this issue, we adopt the following approach: we first describe how `g-sciff` is able to verify *conflict-freedom* of ConDec specifications (see Definition 8.3), and then show how ∃-entailment of properties can be reduced to *conflict-freedom* checking.

10.2.1 *Conflict-freedom Checking Via g-SCIFF*

Conflict-freedom checking is the process of verifying whether a ConDec model $\mathcal{CM}$ supports at least one execution trace, i.e., ∃-entails the $\mathtt{true}$ property.

When checking for conflict-freedom, the user is interested in detecting whether at least one supported execution trace exists, but she is not interested in finding all the possible supported traces. Therefore, `g-sciff` is a suitable technology to deal with this problem: as stated by

Theorem 9.6, the formal property of *weak completeness w.r.t. trace generation* attests that if supported execution traces exist, then g-sciff is able to generate at least one of them.

By combining soundness and weak completeness it is guaranteed that at least one execution trace compliant with a SCIFF specification exists iff g-sciff has a successful derivation, and therefore g-sciff deals with the problem of checking conflict-freedom in a correct wayre.

THEOREM 10.1 (g-sciff can check conflict-freedom). *A ConDec model* $\mathcal{CM}$ *is conflict-free iff* g-sciff *has a successful derivation for* $t_{CLIMB}(\mathcal{CM})$ *(starting from the empty trace):*

$$t_{CLIMB}(\mathcal{CM}) \models_{\exists} \mathrm{true} \Leftrightarrow \exists\, \Delta,\ t_{CLIMB}(\mathcal{CM})_{\emptyset} \vdash_{g\,\Delta}^{\mathcal{T}} \mathrm{true}$$

In this case, the execution trace $\mathcal{T}$ *generated by* g-sciff *is an (intensional) example of execution attesting that* $\mathcal{CM}$ *is conflict free.*

Proof. Straightforward from the results of soundness and weak completeness w.r.t. trace generation (Theorems 9.1 and 9.6). □

10.2.2 *Existential Entailment with g-SCIFF*

As far as now, we have seen g-sciff derivations starting from a single specification. When relying on g-sciff for carrying out ∃-entailment, the following issue therefore arise: how is it possible to consider both the specification of the model and of the desired property when generating compliant execution traces? We prove that by composing the model and the property (see the composition operator COMP introduced in Definition 8.7), ∃-entailment can be reduced to checking whether the obtained model is conflict-free.

THEOREM 10.2 (∃-entailment with g-sciff). *Given a ConDec model* $\mathcal{CM}$ *and a ConDec property* Ψ*,*

$$t_{CLIMB}(\mathcal{CM}) \models_{\exists} t_{CLIMB}(\Psi) \Leftrightarrow \exists\, \Delta,\ t_{CLIMB}(\mathrm{COMP}(\mathcal{CM}, \Psi))_{\emptyset} \vdash_{g\,\Delta}^{\mathcal{T}} \mathrm{true}$$

In this case, the execution trace $\mathcal{T}$ *generated by* g-sciff *can be considered as an (intensional) example of execution which respects both the constraints of the model and the desired property.*

Proof. The proof relies on the compositionality of CLIMB specifications, which has been discussed in Section 4.4 and proven in Corollary 4.1, and on the definition of t_{CLIMB} (given in Section 5.1), which states that a ConDec model is translated to a CLIMB specification whose knowledge base is empty and whose ICs are the union of the translation of each mandatory constraint.

We have that:

$$\begin{aligned}
\exists\, \Delta,\ t_{\mathrm{CLIMB}}\,(\textsc{comp}\,(\mathcal{CM},\Psi))_{\emptyset} \vdash^{\mathcal{T}}_{g\,\Delta} \mathrm{true} &\Leftrightarrow && \text{(Th. 10.1)}\\
\textsc{compliant}(t_{\mathrm{CLIMB}}\,(\textsc{comp}\,(\mathcal{CM},\Psi))_{\mathcal{T}}) &\Leftrightarrow && \text{(Def. 5.1)}\\
\textsc{compliant}(\langle\emptyset, t_{\mathrm{IC}}\left(\mathcal{C}^{\mathcal{CM}}_{m} \cup \mathcal{C}^{\Psi}_{m}\right)\rangle_{\mathcal{T}}) &\Leftrightarrow && \text{(Sec. 5.1)}\\
\textsc{compliant}(\langle\emptyset, t_{\mathrm{IC}}\left(\mathcal{C}^{\mathcal{CM}}_{m}\right) \cup t_{\mathrm{IC}}\left(\mathcal{C}^{\Psi}_{m}\right)\rangle_{\mathcal{T}}) &\Leftrightarrow && \text{(Def. 4.15)}\\
\textsc{compliant}(\langle\emptyset, t_{\mathrm{IC}}\left(\mathcal{C}^{\mathcal{CM}}_{m}\right)\rangle_{\mathcal{T}}) \wedge &&&\\
\textsc{compliant}(\langle\emptyset, t_{\mathrm{IC}}\left(\mathcal{C}^{\Psi}_{m}\right)\rangle_{\mathcal{T}}) &\Leftrightarrow && \text{(Def. 5.1)}\\
\textsc{compliant}(t_{\mathrm{CLIMB}}\,(\mathcal{CM})_{\mathcal{T}}) \wedge &&&\\
\textsc{compliant}(t_{\mathrm{CLIMB}}\,(\Psi)_{\mathcal{T}}) &\Leftrightarrow && \text{(Def. 10.2)}\\
t_{\mathrm{CLIMB}}\,(\mathcal{CM}) \models_{\exists} t_{\mathrm{CLIMB}}\,(\Psi) & && \square
\end{aligned}$$

10.3 VERIFICATION OF UNIVERSAL PROPERTIES WITH G-SCIFF

Universal entailment of properties is carried out with `g-sciff` via a reduction to existential entailment, which in turn is again reduced to conflict-freedom checking.

10.3.1 *Complementing Integrity Constraints*

The idea behind the reduction of ∀-entailment to ∃-entailment resembles model checking [57][2]: a property is ∀-entailed iff the negated property is not ∃-entailed.

Complementary Integrity Constraints

In the CLIMB setting, the negation of a basic property (i.e., a single IC) cannot be directly applied on the property, due to syntactic restrictions. However, we notice that "negation" can be referred to the notion of compliance: given an arbitrary execution trace, the "negation" of an IC evaluates the trace as compliant iff the IC evaluates it as non-compliant. This intuitive concept is formalized by introducing the concept of *complementary* ICs.

DEFINITION 10.4 (Complementary Integrity Constraints (ICs)). Given a knowledge base $\mathcal{KB}$, two CLIMB ICs IC_1 and IC_2 are complementary w.r.t. $\mathcal{KB}$ (written $\mathrm{IC}_1 = \overline{\mathrm{IC}_2}^{\mathcal{KB}}$, $\mathrm{IC}_2 = \overline{\mathrm{IC}_1}^{\mathcal{KB}}$) iff:

$$\forall\, \mathcal{T},\ \textsc{compliant}\,(\langle\mathcal{KB}, \{\mathrm{IC}_1\}\rangle_{\mathcal{T}}) \Leftrightarrow \neg\textsc{compliant}\,(\langle\mathcal{KB}, \{\mathrm{IC}_2\}\rangle_{\mathcal{T}})$$

We will use the simplified notation $\mathrm{IC}_1 = \overline{\mathrm{IC}_2}$, $\mathrm{IC}_2 = \overline{\mathrm{IC}_1}$ where the context (i.e., $\mathcal{KB}$) is apparent or $\mathcal{KB}$ is empty.

Complementation function for ConDec

Given an IC, there exist many different ICs complementary to it. In the general case, the synthesis of a complementary IC is a task that must be accomplished manually. In the ConDec setting, the formalization of each constraint can be easily complemented; in this way, we

2 Model checking will be described in Section 11.3.1.

could imagine that the complemented formalization is encapsulated in a translation function $\overline{t_{IC}}$ which, given a ConDec constraint C, returns an IC complementary to $t_{IC}(C)$.

Example 10.1 (Complementing the response and negation response ConDec constraints). *Let us consider the response ConDec constraint between activities* a *and* b. *Such a constraint is violated by an execution trace if it contains at least one execution of activity* a *which is not followed by a consequent execution of* b. *Starting from this observation, an* IC *complementary to the one formalizing the response constraint can be obtain in a straightforward way:*

$$\overline{t_{IC}}\left(\boxed{a} \bullet\!\!\!\longrightarrow \boxed{b}\right) \triangleq true \rightarrow \mathbf{E}(exec(a), T_a) \\ \wedge \mathbf{EN}(exec(b), T_b) \wedge T_b > T_a.$$

$\overline{t_{IC}}\left(\boxed{a} \bullet\!\!-\!\!\|\!\!\!\longrightarrow \boxed{b}\right)$ *can be obtained in a very similar way, by simply substituting the negative expectation on* b *with a positive expectation:*

$$\overline{t_{IC}}\left(\boxed{a} \bullet\!\!-\!\!\|\!\!\!\longrightarrow \boxed{b}\right) \triangleq true \rightarrow \mathbf{E}(exec(a), T_a) \\ \wedge \mathbf{E}(exec(b), T_b) \wedge T_b > T_a.$$

Indeed, an execution trace violates $\boxed{a} \bullet\!\!-\!\!\|\!\!\!\longrightarrow \boxed{b}$ *if it contains at least one execution of activity* a *which is followed by an execution of activity* b.

10.3.2 *Reduction of Universal Entailment to Existential Entailment*

$\forall$-entailment can be reduced to $\exists$-entailment by exploiting the concept of complementary ICs. In particular, since all the constraints of Ψ must be $\forall$-entailed, the following theorem states that a ConDec property Ψ is $\forall$-entailed by a ConDec model $\mathcal{CM}$ iff there does not exist a mandatory constraint of Ψ whose complementation is $\exists$-entailed by $\mathcal{CM}$. Contrariwise, if at least one complemented constraint is $\exists$-entailed, the execution trace generated to prove the $\exists$-entailment amounts to a counter-example, which demonstrates that the original property is not guaranteed in any possible execution.

Theorem 10.3 (Reduction of $\forall$-entailment to $\exists$-entailment). *Given a ConDec model* $\mathcal{CM}$ *and a ConDec property* Ψ, *where* $\mathcal{C}_m^{\Psi} = \bigcup_{i=1}^{n} C_i$,

$$(\forall\, i \in [1, \ldots, n], t_{CLIMB}(\mathcal{CM}) \not\models_{\exists} \langle \emptyset, \overline{t_{IC}}(C_i) \rangle) \\ \Leftrightarrow t_{CLIMB}(\mathcal{CM}) \models_{\forall} t_{CLIMB}(\Psi)$$

Proof. The proof relies on the compositionality of CLIMB specifications, which has been discussed in Section 4.4 and proven in Corollary 4.1, and on the definition of t_{CLIMB} (given in Section 5.1), which states that a ConDec model is translated to a CLIMB specification whose knowledge base is empty and whose ICs are the union of the translation of each mandatory constraint.

$$
\begin{aligned}
& t_{CLIMB}(\mathcal{CM}) \models_\forall t_{CLIMB}(\Psi) \Leftrightarrow && \text{(Def. 10.3)}\\
& \forall \mathcal{T} \textsc{compliant}(t_{CLIMB}(\mathcal{CM})_\mathcal{T}) \\
& \quad \Rightarrow \textsc{compliant}(t_{CLIMB}(\Psi)) \Leftrightarrow && \text{(Def. 5.1)}\\
& \forall \mathcal{T} \textsc{compliant}(t_{CLIMB}(\mathcal{CM})_\mathcal{T}) \\
& \quad \Rightarrow \textsc{compliant}\left(\langle \emptyset, t_{IC}\left(\bigcup_{i=1}^{n} C_i\right)\rangle_\mathcal{T}\right) \Leftrightarrow && \text{(Sec. 5.1)}\\
& \forall \mathcal{T} \textsc{compliant}(t_{CLIMB}(\mathcal{CM})_\mathcal{T}) \\
& \quad \Rightarrow \textsc{compliant}\left(\langle \emptyset, \bigcup_{i=1}^{n} t_{IC}(C_i)\rangle_\mathcal{T}\right) \Leftrightarrow && \text{(Def. 4.15)}\\
& \forall \mathcal{T} \textsc{compliant}(t_{CLIMB}(\mathcal{CM})_\mathcal{T}) \\
& \quad \Rightarrow \bigwedge_{i=1}^{n} \textsc{compliant}(\langle \emptyset, t_{IC}(C_i)\rangle_\mathcal{T}) \Leftrightarrow \\
& \forall i \forall \mathcal{T} \textsc{compliant}(t_{CLIMB}(\mathcal{CM})_\mathcal{T}) \\
& \quad \Rightarrow \textsc{compliant}(\langle \emptyset, t_{IC}(C_i)\rangle_\mathcal{T}) \Leftrightarrow \\
& \forall i \ \nexists \mathcal{T} \textsc{compliant}(t_{CLIMB}(\mathcal{CM})_\mathcal{T}) \\
& \quad \wedge \neg \textsc{compliant}(\langle \emptyset, t_{IC}(C_i)\rangle_\mathcal{T}) \Leftrightarrow && \text{(Def. 10.4)}\\
& \forall i \ \nexists \mathcal{T} \textsc{compliant}(t_{CLIMB}(\mathcal{CM})_\mathcal{T}) \\
& \quad \wedge \textsc{compliant}(\langle \emptyset, \overline{t_{IC}}(C_i)\rangle_\mathcal{T}) \Leftrightarrow && \text{(Def. 10.2)}\\
& \forall i \ t_{CLIMB}(\mathcal{CM}) \not\models_\exists \langle \emptyset, \overline{t_{IC}}(C_i)\rangle
\end{aligned}
$$

□

Verification decomposition

When g-sciff succesfully computes a Δ showing that there exists a k for which $t_{CLIMB}(\mathcal{CM}) \models_\exists \langle \emptyset, \overline{t_{IC}}(C_k)\rangle$, the generated execution trace contained in Δ can be considered as a counter-example which amounts to a proof that Ψ is not $\forall$-entailed. Therefore, verification can be carried out by considering each single constraint belonging to the property separately. The constraint is first complemented by means of $\overline{t_{IC}}$, and the complemented version is then subject to $\exists$-entailment. The first complemented constraint which is $\exists$-entailed by the specification proves that the original property is not $\forall$-entailed, and thus the verification procedure terminates without needing to check the other constraints $\in \Psi$.

Examples of reduction of $\forall$-entailment to $\exists$-entailment

For example, Query 8.1 can be verified in an existential way by complementing the concept of dead activity, i.e., by looking for an execution trace in which send receipt is executed at least once [3]. If such an execution trace exists, then the activity is not dead.

Query 8.3 cannot be modeled by means of a single ConDec model: as shown in Section 10.1.1, it is in fact a disjunction of two properties, each one representable in ConDec. However, the property can be easily expressed in CLIMB. A possible way to complement it is the following:

3 Indeed, $\overline{t_{IC}}\left(\boxed{a}^{\,0}\right) \triangleq \mathtt{true} \rightarrow \mathbf{E}(exec(a), T) = t_{IC}\left(\boxed{a}^{\,1..*}\right)$ is complementary to $t_{IC}\left(\boxed{a}^{\,0}\right) \triangleq \mathtt{true} \rightarrow \mathbf{EN}(exec(a), T)$.

$$\mathrm{true} \rightarrow \mathbf{E}(exec\,(1-click_payment)\,,T_p) \wedge \mathbf{E}(exec\,(send_receipt)\,,T_s) \wedge \mathbf{EN}(exec\,(accept_advert)\,,T_a) \wedge T_a < T_s.$$

Intuitively, the IC states that 1-click payment must be chosen, and that advertising must not accepted *before* the receipt is sent.

To verify ∃-entailment of such a complemented property, g-sciff joins it with the specification representing the ConDec model under study (shown in Figure 59). By adopting a depth-first strategy, in its first successful derivation g-sciff generates the following intensional execution trace, which amounts as a counter-example:

$$\begin{aligned}&\mathbf{H}(choose_item, T_i),\\ &\mathbf{H}(register, T_r),\mathbf{H}(close_order, T_o) T_o > T_i,\\ &\mathbf{H}(1-click_payment, T_p) \wedge T_p > T_r \wedge T_p > T_o,\\ &\mathbf{H}(payment_done, T_d) \wedge T_d > T_p,\\ &\mathbf{H}(send_receipt, T_s) \wedge T_s > T_d, \mathbf{H}(accept_advert, T_a) \wedge T_a > T_s.\end{aligned}$$

The trace is intensional and the contained happened events are partially ordered; among such happened events, we can find that the chosen payment is 1-click, and that the customer accepts ads after the execution of the send_receipt activity.

10.4 condec loops and termination issues

In Section 9.4.4, we have pointed out that the acyclicity and boundedness conditions needed for guaranteeing g-sciff termination do not hold for an arbitrary ConDec model. In fact, the generative rule $\mathbf{E}(E,T) \rightarrow \mathbf{H}(E,T)$, which must be taken into account when defining a suitable level mapping, interferes with rules containing a happened event in the body and a positive expectation in the head; all the rules formalizing ConDec relation constraints are of this kind, as attested by Table 21 – Page 93.

Example 10.2 (A critic ConDec model). *Let us consider a simple ConDec model, which is translated to a CLIMB specification which is bounded w.r.t.* sciff *but unbounded w.r.t.* g-sciffs*. The model contains only two opposite response constraints between two activities: the first states that* b *is response of* a*, the second that* a *is response of* b*. Its CLIMB formalization is composed by two ICs:*

$$t_{IC}\left(\boxed{a} \rightleftarrows \boxed{b}\right) = \{\mathbf{H}(exec\,(a)\,,T_1) \rightarrow \mathbf{E}(exec\,(b)\,,T_2) \wedge T_2 > T_1,\quad \mathbf{H}(exec\,(b)\,,T_3) \rightarrow \mathbf{E}(exec\,(a)\,,T_4) \wedge T_4 > T_3\,\}$$

The definition of a level mapping w.r.t. g-sciff *must take into account the generative rule* $\mathbf{E}(E,T) \rightarrow \mathbf{H}(E,T)$*. The generative rule is general, i.e. it triggers for any expected event at any time. Therefore, when trying to define a level mapping* $|\cdot|$ *guaranteeing acyclicity and boundedness of the specification, the following conditions must be satisfied:*

- *for every* $T_2 > T_1$, $|\mathbf{H}(exec(a), T_1)| > |\mathbf{E}(exec(b), T_2)|$ *(first* IC *of the specification);*
- *for every* $T_4 > T_3$, $|\mathbf{H}(exec(b), T_3)| > |\mathbf{E}(exec(a), T_4)|$ *(second* IC *of the specification);*
- *for every* T_a, $|\mathbf{E}(exec(a), T_a)| > |\mathbf{H}(exec(a), T_a)|$ *(generative rule, grounded on* $exec(a)$*);*
- *for every* T_b, $|\mathbf{E}(exec(b), T_b)| > |\mathbf{H}(exec(b), T_b)|$ *(generative rule, grounded on* $exec(b)$*).*

Finding a bounded level mapping satisfying all these conditions is infeasible, because, e.g.:

$$|\mathbf{E}(exec(a), 10)| > |\mathbf{H}(exec(a), 10)| > |\mathbf{E}(exec(b), 11)| >$$
$$>|\mathbf{H}(exec(b), 11)| > |\mathbf{E}(exec(a), 12)| > \ldots$$

Therefore, termination is not guaranteed when `g-sciff` *reasons upon such a specification.*

When the empty execution trace is given at start-up, then `g-sciff` *immediately terminates, stating that the empty execution trace itself is compliant with the specification. Contrariwise, when the initial trace* $\{\mathbf{H}(exec(a), 1)\}$ *is provided,* `g-sciff` *loops, trying to explicitly generate the infinite trace*

$$\begin{aligned}
&\mathbf{H}(exec(a), 1),\\
&\mathbf{H}(exec(b), T') \wedge T' > 1,\\
&\mathbf{H}(exec(a), T'') \wedge T'' > T',\\
&\mathbf{H}(exec(b), T''') \wedge T''' > T'',\\
&\ldots
\end{aligned}$$

Note that the ConDec model [a ⇄ b] *is not correct: as soon as one among the activities* `a` *and* `b` *is executed, the implicit assumption that every execution must terminate in finite time is violated.*

A solution must be therefore provided to identify and treat ConDec models for which `g-sciff` termination is not guaranteed. Two possible approaches may be adopted:

A. the CLIMB formalization of ConDec relation constraints (i.e., the t_{CLIMB} function) must be revised, in order to avoid interferences with the generative rule;

B. the *structure* of the ConDec model under study (or of the underlying formalization) must be investigated in order to identify sources of (potential) non-termination.

We follow a hybrid approach. First of all, we revise the formalization of relation constraints. This solves the problem only partially, and therefore we also synthesize an analysis procedure able to identify and deal with problematic ConDec models.

10.4.1 *Reformulation of ConDec relation constraints*

We provide an alternative formulation for (some of the) relation constraints. After such a reformulation, the identification of ConDec models which can potentially cause non-termination of `g-sciff` becomes straigthforward. The possibility of replacing the CLIMB formalization of a single ConDec constraint with an equivalent one, without affecting the formalization of the other constraints, is guaranteed by Theorem 4.1 – Page 80.

Reformulation of the Responded Existence Constraint

The old formalization

$$t_{IC}\left(\boxed{a} \bullet\!\!-\!\!\!-\!\!\!-\!\!\boxed{b}\right) \triangleq \mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b).$$

is replaced by

$$t_{IC}\left(\boxed{a} \bullet\!\!-\!\!\!-\!\!\!-\!\!\boxed{b}\right) \triangleq true \rightarrow \mathbf{EN}(exec(a), T_a) \vee \mathbf{E}(exec(b), T_b).$$

In fact, saying that "if a is executed, then also b is executed" is the same as expressing that "either a is never executed, or b is executed". The proof of equivalence is straightforward, either by proving it directly, or by establishing the soundness of the new formalization w.r.t. the LTL formula representing the responded existence constraint.

The new formalization does not contain happened events in the body, and therefore does not interfere with the generative rule of `g-sciff` anymore. Such an advantage holds also for the coexistence constraint, which is defined as two opposite responded existence constraints.

Reformulation of the Alternate Constraints

Alternate constraints are reformulated for what concerns their interposition part (see Table 21). Such a reformulation enables the possibility to restructure the formalization of the whole constraints: they are not decomposed anymore in a response/precedence part plus the interposition part, but they are directly modeled as an augmented response/precedence.

Let us consider the ConDec constraint $\boxed{a} \bullet\!\!\Longrightarrow\!\!\blacktriangleright \boxed{b}$, which states that:

- if a occurs, then b must be executed afterwards;
- b must be executed between any two occurrences of a.

It can be equivalently formulated as:

- if a occurs, then b must be executed afterwards;
- if a occurs, then another consequent occurrence of a is forbidden until b is executed (otherwise, no occurrence of b would be present between the two as).

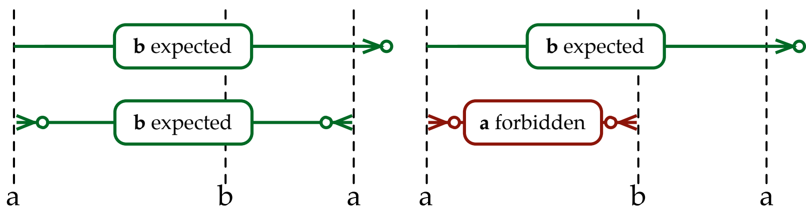


Figure 36: Shifting the perspective when modeling the alternate response constraint.

CONSTRAINT	STRENGTH	DESCRIPTION
a •—▶ b	weak	Between the two occurrences of a and b, *other* activities can be executed
a •═▶ b	medium	Between the two occurrences of a and b, other activities *but* a can be executed
a •≡▶ b	strong	Between the two occurrences of a and b, *no* activity can be executed

Table 24: Strength of "forward" relation ConDec constraints.

These two different but equivalent perspectives are depicted in Figure 36.

By adopting the new perspective, alternate response/precedence constraints can be simply formalized as follows:

$$t_{IC}\left(\boxed{a} \bullet\!\!\Longrightarrow \boxed{b}\right) \triangleq \mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b) \land T_b > T_a$$
$$\land\, \mathbf{EN}(exec(a), T_{a2})$$
$$\land\, T_{a2} > T_a \land T_{a2} < T_b.$$
$$t_{IC}\left(\boxed{a} \Longrightarrow\!\!\bullet \boxed{b}\right) \triangleq \mathbf{H}(exec(b), T_b) \rightarrow \mathbf{E}(exec(a), T_a) \land T_a < T_b$$
$$\land\, \mathbf{EN}(exec(b), T_{b2})$$
$$\land\, T_{b2} > T_a \land T_{b2} < T_b.$$

If we consider the "quantitative" representation of the chain response constraint (i.e., the formalization produced by applying $t^{\circlearrowright}_{CLIMB}$), this new formalization of the alternate constraints clearly shows how the "constraints' strength" increases moving from plain constraints to chain ones (see Table 24).

Reformulation of the "Quantitative" Negation Chain Response Constraint

When the negation chain response constraint is formalized by adopting a quantitative notion of time (i.e., by applying the $t^{\circlearrowright}_{IC}$ function), the produced IC involves a positive expectation in the head (see Sec-

tion 6.1.2–Page 110); in fact, the formalization of [a •‖→ b] expresses that between an occurrence of activity a and each following occurrence of b, at least one further activity must separate the two occurrences (interposing between them):

$$t_{IC}^{\circlearrowright}\left(\boxed{a} \bullet\!\!\Vert\!\!\!\rightarrow \boxed{b}\right) \triangleq \mathbf{H}(exec\,(a)\,,T_a) \land \mathbf{H}(exec\,(b)\,,T_b) \land T_b > T_a \rightarrow \mathbf{E}(exec\,(X)\,,T_x) \land T_x > T_a \land T_x < T_b.$$

Such a formalization can be restructured as an IC imposing expectations on the future (w.r.t. T_a, the time used in the body of the IC). The new equivalent formalization expresses the intuitive idea that saying "between a and b a further activity is expected to occur" is the same as saying "If a is executed, then either b is never executed afterwards, or it is executed sometimes in the future; in the latter case, at least one further activity (different from b and a) should occur inbetween":

$$t_{IC}^{\circlearrowright}\left(\boxed{a} \bullet\!\!\Vert\!\!\!\rightarrow \boxed{b}\right) \triangleq \mathbf{H}(exec\,(a)\,,T_a) \rightarrow \mathbf{EN}(exec\,(b)\,,T_b) \land T_b > T_a \lor \mathbf{E}(exec\,(b)\,,T_b) \land T_b > T_a \land \mathbf{EN}(exec\,(X)\,,T_x) \land X \neq b \land X \neq a \land T_x > T_a \land T_x < T_b.$$

10.4.2 *Unbounded Specifications and Looping ConDec Models*

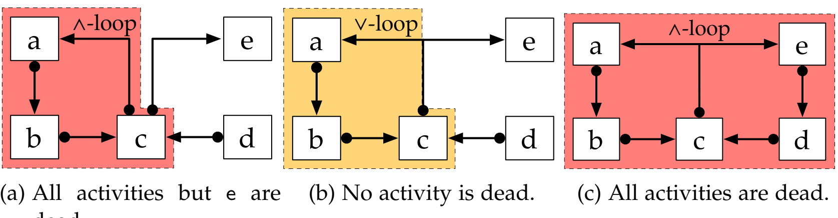
By adopting the revised formalization discussed in the previous Section, ICs containing happened events in the body and positive expectation in the head (which are the ones interfering with the generative rule of `g-sciff`) are of two kinds:

Forward vs backward constraints

FORWARD ICS impose a positive expectation at a time greater than the time of the happened event contained in the body. This is the case of response, alternate response and chain response constraints, whose forward behaviour is caused by their basic "response" part[4]. Following the nomenclature given for ICs, these constraints will be called *forward* constraints.

BACKWARD ICS impose a positive expectation at a time lower than the time of the happened event contained in the body. This is the case of precedence, alternate precedence and chain precedence constraints, whose backward behaviour is caused by their basic

4 If the quantitative CLIMB formalization of ConDec is used instead of the classical one (i.e., $t_{CLIMB}^{\circlearrowright}$ is applied instead of t_{CLIMB}), also the negation chain response must be considered among these constraints.



(a) All activities but e are dead. (b) No activity is dead. (c) All activities are dead.

Figure 37: Three ConDec models containing different kind of loops.

"precedence" part. These constraints will be called *backward* constraints.

Using this classification, unbounded specifications can be easily identified. In particular, a specification is unbounded if two activities x and y are involved in a "cyclic chain" of forward ICs, i.e., there exist a forward IC relating x with activity a_1, a forward IC relating a_1 with activity a_2, ..., and a forward IC relating a_N with y. The simplest case is the one presented in Example 10.2, where two activities are mutually connected with two response constraints. The same holds if the "cyclic chain" is constituted by backward ICs instead of forward ones.

The interesting fact is that such situations are identifiable by carrying out an analysis directly at the graphical level of ConDec: unbounded specifications are derived from ConDec models containing "cyclic chains" of forward or backward contraints. Figure 37 shows three models covering the different kind of loops that can be encountered in ConDec[5].

Termination of `g-sciff` *when reasoning upon different looping models*

For all these models, termination of `g-sciff` is not guaranteed. Let us for example consider the problem of checking if activity d is dead in the three models of Figure 37:

- `g-sciff` always loops when checking if d is a dead activity in Figure 37(a). To disprove the property, `g-sciff` checks if d can be executed in at least one instance; however, by simulating the execution of d, it enters in an endless computation, generating the execution trace $d \rightarrow c \rightarrow a \rightarrow b \rightarrow c \rightarrow \ldots$. Note that activity d is, in this model, dead, because each execution of a ConDec model must eventually terminate, while the execution of d leads to enter within an infinite loop. The same holds for activities a, b and c.

- In Figure 37(b), activity d is not dead; indeed, even if a loop is contained in the model, it is possible to exit the loop, by choosing to execute activity e after c instead of a. Termination of `g-sciff` when checking if d is dead is determined, in this case, by the syntactic structure of the IC which represents the disjunctive response spanning from activity c. Since `g-sciff` adopts a depth-first strategy,

5 Since the forward or backward nature of a relation constraint is determined only by its basic response/precedence component, it is not important which specific constraints are considered.

  - if a is the first choice, `g-sciff` loops as in the case of Figure 37(a);
  - if e is the first choice, `g-sciff` has a finite successful derivation which proves that d is not dead by producing the execution trace d → b → e as a counter-example.

  Adopting an iterative deepening strategy[6] would help in this specific case, but not in general.

- Figure 37(c) extends Figure 37(b) by introducing a further response constraint between e and d. In this situation, both choices spanning from b cause a loop. Therefore, d is a dead activity (and so are all the other activities), but `g-sciff` is unable to prove it.

Classification of loops

The three models depicted in Figure 37 clearly show that two kind of loops could be created by combining relation constraints, and that the difference lies in whether the model supports the possibility of exiting from the loop or not. If such a possibility is supported, the loop will be denoted as an $\vee$-loop, otherwise the loop will be denoted as an $\wedge$-loop.

Definition 10.5 ($\wedge$-loop). Given a ConDec model $\mathcal{CM}$, an activity $A \in \mathcal{A}^{\mathcal{CM}}$ belongs to an $\wedge$-*loop* if when it is executed, the mandatory constraints $\mathcal{C}_m^{\mathcal{CM}}$ force the re-execution of the same activity afterwards (*forward* $\wedge$-loop) or before (*backward* $\wedge$-loop).

Definition 10.6 ($\vee$-loop). Given a ConDec model $\mathcal{CM}$, an activity $A \in \mathcal{A}^{\mathcal{CM}}$ belongs to a $\vee$-*loop* if when it is executed, the mandatory constraints $\mathcal{C}_m^{\mathcal{CM}}$ may force the re-execution of the same activity afterwards (*forward* $\vee$-loop) or before (*backward* $\vee$-loop), depending on the made choices.

In Figure 37(a), activities a, b and c belong to an $\wedge$-loop. In Figure 37(b), such activities belong to an $\vee$-loop, because only if activity a is chosen after c the loop is entered. Finally, in Figure 37(c) all activities belong to an $\wedge$-loop: no matter what choice is taken after having performed c, the user is forced to re-execute the same activity again. Instead, the ConDec model shown in Figure 59 is loop-free.

Relationship between $\wedge$-loops and dead activities

The most important feature of this classification is summarized in the following remark.

Remark 10.1 ($\wedge$-loops contain dead activities). When the execution enters inside an $\wedge$-loop, there is no possibility to exit from it, which means that it becomes impossible to terminate the execution in finite time. Therefore, each activity belonging to an $\wedge$-loop is a dead activity.

The remark points out that once a method to discover $\wedge$-loops is provided, then all the involved activities can be automatically marked as dead, by explicitly adding an absence constraint on them.

6 Where, at each iteration, the maximum depth is related to the maximum number of happened events that can be generated.

10.5 PRE-PROCESSING OF CONDEC MODELS AND LOOP DETECTION

We introduce a set of algorithms able to detect the presence of $\wedge$- and $\vee$-loops inside a ConDec model. Such algorithms rely on a translation function which maps the ConDec model to an AND/OR graphs, enabling the possibility of carrying out the loop detection task.

The loop detection procedure is then embedded inside a pre-processing analysis which must be applied:

FOR $\exists$-CONSISTENCY to the composition of the model and the property. In fact, event if the model and the property are loop free, their composition is not guaranteed to be loop-free.

FOR $\forall$-CONSISTENCY to the model. Indeed, $\forall$-consistency requires each single element of the property to be complemented, and the complementation function $\overline{t_{IC}}$ does not produce ICs which contain happened events in the body and positive expectations in the head; therefore, no complemented component can participate in the formation of a loop, and it is sufficient to test the model alone.

10.5.1 *Transformation of ConDec Models to AND/OR Graphs*

Given a ConDec model, two corresponding AND/OR Graphs $\overrightarrow{\mathcal{G}}$ and $\overleftarrow{\mathcal{G}}$ are generated, taking into account its forward and backward relation constraints respectively[7].

DEFINITION 10.7 (AND/OR graph). An AND/OR graph is a triple $\langle \mathcal{A}, \mathcal{O}, \mathcal{E} \rangle$, where:

- $\mathcal{A}$ is a set of $\wedge$-nodes;
- $\mathcal{O}$ is a set of $\vee$-nodes;
- $\mathcal{E} \subseteq \mathcal{A} \cup \mathcal{O} \times \mathcal{A} \cup \mathcal{O}$ is a set of *directed edges*, i.e., ordered pairs of nodes.

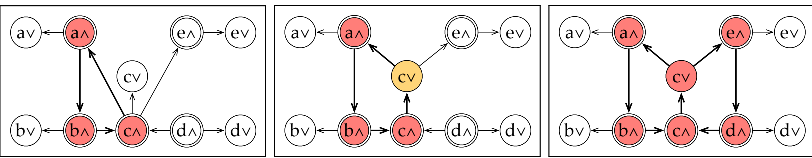
$\wedge$-nodes will be represented inside single-lined circles, $\vee$-nodes will be represented inside double-lined circles and edges will be depicted by arrows.

Transformation of a single activity to an AND/OR subgraph

In the transformation process, each ConDec activity a is mapped to a subgraph containing an $\wedge$-node $a_\wedge$ connected with an $\vee$-node $a_\vee$:

- $a_\wedge$ is used to attach incoming forward (backward resp.) relation constraints, and to handle outgoing forward (backward resp.) relation constraints which do not involve branches on the target;

7 Forward vs backward relation constraints have been defined in Section 10.4.2.



(a) Presence of an $\wedge$-loop. (b) Presence of an $\vee$-loop. (c) Presence of an $\wedge$-loop.

Figure 38: The three AND/OR forward graphs corresponding to the ConDec models shown in Figure 37.

- $a_\vee$ is used to handle outgoing forward (backward resp.) relation constraints which involve branches on the target (i.e., have a disjunctive behaviour).

We will denote the produced graph as a *forward* (*backward* resp.) graph if forward (backward resp.) relation constraints are considered. Activities that are explicitly subject to an absence constraint, i.e., for which it is explicitly known that they are dead, are not considered in the transformation process: these activities cannot be executed, and therefore they "break" all the loops containing them.

The generation of an AND/OR graph starting from a ConDec model $\mathcal{CM}$ is shown in Listing TransformToAndOrGraph($\mathcal{CM}$, $\mathtt{Forward}$), where $\mathtt{Forward}$ is a boolean representing whether the AND/OR graph must be built by considering the forward or backward relation constraints of $\mathcal{CM}$.

The application of this translation procedure to the three diagrams shown in Figure 37 is depicted in Figure 38[8]. In particular, forward AND/OR graphs are presented: since the models of Figure 37 contain only forward constraints, the application of the translation method on backward constraints would produce loop-free graphs with isolated nodes.

Characterization of $\wedge$- and $\vee$-loops on the graph

As the three graphs of Figure 38 suggest, an $\wedge$- and an $\vee$-loop differ from each other in that an $\vee$-loop is a loop which contains at least one $\vee$-node, and this $\vee$-node is connected with at least one node which does not belong to an $\wedge$-loop (the presence of this node reflects the possibility of exiting from the loop in the corresponding ConDec model). The following Section handles the discovery of $\wedge$- and $\vee$-loops in the general case.

10.5.2 *Detection of $\wedge$- and $\vee$-loops*

Detection of $\wedge$- and $\vee$-loops is tackled by a cascaded application of the two algorithms embedded into Listing FindAndLoops($\mathcal{G}$) and Listing ContainsLoops($\mathcal{G}$).

$\wedge$-loop detection algorithm

The algorithm for detecting $\wedge$-loops is recursively defined as follows. Given a set $\mathcal{P}$ of already visited nodes and the current node n:

- if $n \in \mathcal{P}$, then n belongs to an $\wedge$-loop;

8 The models must be previously composed with the property, which in the case study simply adds a $1..*$ constraint on activity d.

```
function: TransformToAndOrGraph(ConDec model CM, boolean Forward)
returns : An AND/OR graph mapping CM by taking into account
          its forward or backward constraints, depending on the value
          of Forward (true →forward constraints, false →backward
          constraints)
1  begin
2    A ← ∅;
3    O ← ∅;
4    E ← ∅;
     /* ReduceModel(CM) returns a reduced version of CM, by
        eliminating each activity a s.t. [a]^0 ∈ C_m^CM, and by
        modifying constraints accordingly                          */
5    CMRed ← ReduceModel(CM);
6    foreach activity a ∈ A^CMRed do
7      A ← A ∪ {a_∧};
8      O ← O ∪ {a_∨};
9      E ← E ∪ {(a_∧, a_∨)};
10     cur ← ∅;                        // current set of constraints
11     if Forward then // forward graph
12       cur = {c(S,T) ∈ C_m^CMRed | c is forward ∧ a ∈ S};
13     else// backward graph
14       cur = {c(S,T) ∈ C_m^CMRed | c is backward ∧ a ∈ S};
15     end
16     foreach c(Source,Target) ∈ cur do
17       if ||Target|| = 1 then // "normal" constraint
18         E ← E ∪ {(a_∧, t)}, having Target = {t};
19       else// disjunctive constraint
20         foreach activity t ∈ Target do
21           E ← E ∪ {(a_∨, t)};
22         end
23       end
24     end
25   end
26   return ⟨A, O, E⟩;
27 end
```

Function TransformToAndOrGraph($\mathcal{CM}$,Forward)

- if n is an $\wedge$-node, then it belongs to an $\wedge$-loop if at least one node to which n is connected belongs to an $\wedge$-loop (detected updating the set of visited nodes by considering also n);
- if n is an $\vee$-node, then it belongs to an $\wedge$-loop if all the nodes to which n is connected belong to an $\wedge$-loop (detected updating the set of visited nodes by considering also n).

Model augmentation

Then, we hypothesize that after the detection of $\wedge$-loops, the ConDec model is *augmented* by explicitly inserting an absence constraint on each $\wedge$-looping activity, and that the $\vee$-loop detection algorithm is applied on this augmented model (after its transformation to an AND/OR graph). In this way, before $\vee$-loop detection all the $\wedge$-loops

```
   function: FindAndLoops(AND/OR graph 𝒢 = ⟨𝒜, 𝒪, ℰ⟩)
   returns : The set of nodes belonging to ∧-loops
1  begin
2      ℒ ← ∅;
3      foreach n ∈ 𝒜 ∪ 𝒪 do
4          if BelongsToAndLoop(n, ∅) then
5              ℒ ← ℒ ∪ {n};
6          end
7      end
8      return ℒ;
9  end

   function: BelongsToAndLoop(node n, set 𝒫 of already visited
             nodes, AND/OR graph 𝒢 = ⟨𝒜, 𝒪, ℰ⟩)
   returns : true if n belongs to a ∧-loop; false otherwise
10 begin
11     if n ∈ 𝒫 then
12         return true;
13     else
14         𝒫′ ← 𝒫 ∪ n;
15         if n ∈ 𝒜 then
16             foreach m | (n, m) ∈ ℰ do
17                 if BelongsToAndLoop(m, 𝒫′) then
18                     return true;
19                 end
20             end
21             return false;
22         end
23         if n ∈ 𝒪 then
24             foreach m | (n, m) ∈ ℰ do
25                 if ¬BelongsToAndLoop(m, 𝒫′) then
26                     return false;
27                 end
28             end
29             return true;
30         end
31     end
32 end
```

Function FindAndLoops($\mathcal{G}$)

have been already eliminated from the graph[9], hence $\vee$-loops can be detected by simply looking for "normal" loops.

General loop detection algorithm

The algorithm for discovering "normal" loops is recursively defined as follows. Given a set $\mathcal{P}$ of already visited nodes and the current node n:

- if $n \in \mathcal{P}$, then n belongs to an loop;

9 Indeed, remember that activities associated to an absence constraint are ruled out when the graph is built.

```
   function: ContainsLoops(AND/OR graph 𝒢 = ⟨𝒜, 𝒪, ℰ⟩)
   returns : true if 𝒢 contains loops; false otherwise
 1 begin
 2   ℒ ← ∅;
 3   foreach n ∈ 𝒜 ∪ 𝒪 do
 4     if BelongsToLoop(n, ∅) then
 5       return true;
 6     end
 7   end
 8   return false;
 9 end

   function: BelongsToLoop(node n, set 𝒫 of already visited nodes,
             AND/OR graph 𝒢 = ⟨𝒜, 𝒪, ℰ⟩)
   returns : true if n belongs to a loop; false otherwise
10 begin
11   if n ∈ 𝒫 then
12     return true;
13   else
14     𝒫′ ← 𝒫 ∪ n;
15     foreach m | (n, m) ∈ ℰ do
16       if BelongsToLoop(m, 𝒫′) then
17         return true;
18       end
19     end
20     return false;
21   end
22 end
```

Function ContainsLoops($\mathcal{G}$)

- otherwise, n belongs to a loop if at least one node to which n is connected belongs to a loop (detected updating the set of visited nodes by considering also n).

10.5.3 *Pre-Processing Procedure*

The schema of the pre-processing procedure is depicted in Figure 39 and formalized in Listing PreProcess($\mathcal{CM}$), where $\mathcal{CM}$ represents the model under study (composition of the model and the property in case of $\exists$-entailment). The procedure consists of the following steps:

A. The ConDec model is translated to the two corresponding forward and backward AND/OR graphs, using the TransformToAndOrGraph function, described in Section 10.5.1.

B. $\wedge$-loop detection is carried out on the obtained graphs, using the FindAndLoops function described in Section 10.5.2.

C. If the model is $\wedge$-loop free, then move to step F; otherwise, move to the following step.

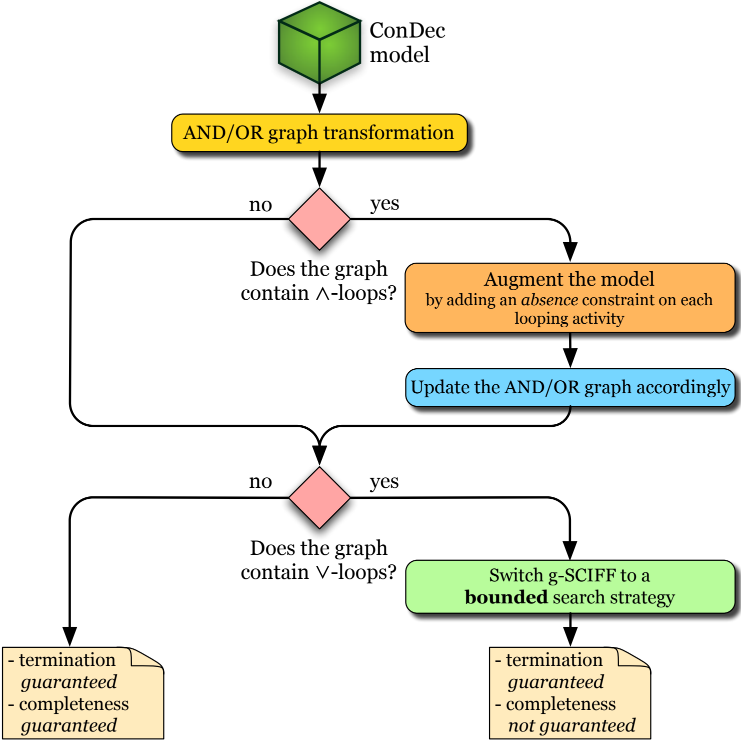


Figure 39: Pre-processing analysis of ConDec models for detecting and handling the presence of loops.

D. Each activity belonging to an $\bigwedge$-loop is augmented with an absence ConDec costraint, used to explicitly denote that the activity is dead, according to Remark 10.1. In this way, g-sciff can correctly deal with models containing $\bigwedge$-loops without experiencing non-termination issues; indeed, as soon as g-sciff expects the execution of an activity belonging to a loop, the explicit presence of an absence constraint (which is formalized in g-sciff with a negative expectation) leads immediately to a failure, because **E**-fulfillment is not respected.

E. The AND/OR graphs are recalculated by reflecting such an augmentation.

F. $\bigvee$-loop detection is carried out on the (possibly recalculated) AND/ OR graphs, using the ContainsLoops function described in Section 10.5.2.

G. The outcome of $\bigvee$-loop detection is exploited as follows:

- If the model is $\bigvee$-loops free, then g-sciff will correctly reason upon the (possibly augmented) model.
- If instead the model contains a $\bigvee$-loop, then termination cannot be guaranteed anymore (see Figure 37(b) and the corresponding discussion in Section 10.4.2). Nevertheless,

function: `PreProcess`(*ConDec model* $\mathcal{CM}$)
returns : An augmented version of $\mathcal{CM}$, where all activities belonging to an $\bigwedge$-loop are subject to an absence constraint, and a boolean stating whether `g-sciff` must switch to a bounded search strategy to guarantee termination.

1 **begin**
2 $\quad \overrightarrow{\mathcal{G}} \leftarrow$ `TransformToAndOrGraph`($\mathcal{CM}$, *true*) ; `// forward graph`
3 $\quad \overleftarrow{\mathcal{G}} \leftarrow$ `TransformToAndOrGraph`($\mathcal{CM}$, *false*) ; `// backward graph`
4 $\quad \mathcal{L}^{\bigwedge} \leftarrow$ `FindAndLoops`($\overrightarrow{\mathcal{G}}$);
5 $\quad \mathcal{L}^{\bigwedge} \leftarrow \mathcal{L}^{\bigwedge} \cup$ `FindAndLoops`($\overleftarrow{\mathcal{G}}$);
6 $\quad$ **if** $\mathcal{L}^{\bigwedge} = \emptyset$ **then**
7 $\quad\quad$ **return** *[*$\mathcal{CM}$*,true]*;
8 $\quad$ **else**
9 $\quad\quad \mathcal{CM}Aug \leftarrow \mathcal{CM}$;
10 $\quad\quad$ **foreach** *activity* $a \in \mathcal{A}$ **do**
11 $\quad\quad\quad$ **if** $a \in \mathcal{L}^{\bigwedge}$ **then**
12 $\quad\quad\quad\quad \mathcal{C}_m^{\mathcal{CM}Aug} \leftarrow \mathcal{C}_m^{\mathcal{CM}Aug} \cup \boxed{a}^{\,0}$;
13 $\quad\quad\quad$ **end**
14 $\quad\quad$ **end**
$\quad\quad$ `/* Recalculation of AND/OR graphs */`
15 $\quad\quad \overrightarrow{\mathcal{G}} \leftarrow$ `TransformToAndOrGraph`($\mathcal{CM}Aug$, *true*);
16 $\quad\quad \overleftarrow{\mathcal{G}} \leftarrow$ `TransformToAndOrGraph`($\mathcal{CM}Aug$, *false*);
17 $\quad\quad Switch \leftarrow$ `ContainsLoops`($\overrightarrow{\mathcal{G}}$);
18 $\quad\quad$ **if** $\neg$ $Switch$ **then**
19 $\quad\quad\quad Switch \leftarrow$ `ContainsLoops`($\overleftarrow{\mathcal{G}}$);
20 $\quad\quad$ **end**
21 $\quad\quad$ **return** *[*$\mathcal{CM}Aug$*, *$Switch$*]*;
22 $\quad$ **end**
23 **end**

Function `PreProcess`($\mathcal{CM}$)

a maximum bound on the length of the possible execution traces produced by `g-sciff` can be imposed, choosing a *bounded depth-first* search strategy for `g-sciff`[10] (similarly to bounded model checking[29][11]). The bound can be tuned by the user, reflecting the "size" of the domain under study. Anyway, the user must be alerted that the search strategy has been modified, because even if *bounded depth-first* preserves termination, it undermines completeness. Indeed, by adopting a bounded search strategy, a positive answer is correct, whereas a *no* answer does not prove that the ConDec model contains a conflict, but only that there is

10 An alternative solution would be to choose suitable heuristics for driving the search strategy.
11 Bounded model checking addresses the problem of verifying the validity of a formula within a predefinite number of transitions of a system.

no compliant execution trace whose length is up to the bound[12].

10.6 dealing with an infinite number of finite derivations

The pre-processing procedure described in Section 10.5.3 guarantees that when g-sciff is applied to an arbitrary ConDec model, termination is always preserved (at the price of loosing completeness when ∨-loops are contained in the model).

The infinite branches anomaly

Termination, in turn, guarantees that each derivation produced by g-sciff has a finite length. However, it cannot guarantee that also the *number of computed derivations is finite*. In other words, it could be possible that there is an infinite number of finite derivations computed by g-sciff, i.e., that the generated proof tree has infinite branches. Since we are interested in finding whether a successful derivation actually exists, if (some of) these derivations are successful, then g-sciff will always answer in finite time. On the contrary, if all these derivations lead to a failure node, then g-sciff will try to explore them all, running forever. We will call this problem the *infinite branches anomaly*.

10.6.1 *Succession Constraints and Infinite Branching Proof Trees*

The case of succession constraints, i.e., a •—▶• b, a •═▶• b and a •≡▶• b could bring g-sciff to experience the infinite branches anomaly.

The following example shows how a simple ConDec model containing a succession constraint is verified by g-sciff producing an infinite number of finite derivations, i.e., infinite generated traces.

Example 10.3 (Succession constraints and the infinite branching anomaly). *Let us consider the simple ConDec model* a (1..∗) •—▶• b, *stating that:*

- *activity* a *must be executed at least once;*
- *if* a *is executed, then* b *must be executed afterwards;*
- *if* b *is executed, then* a *must have been executed before.*

When asking g-sciff *if the model contains conflict, the proof procedure operates as follows:*

A. g-sciff *starts with a pending expectation* $\mathbf{E}(\mathsf{exec}(a), T_{a1})$*, due to constraint* a (1..∗)*;*

---

12 For example, g-sciff would state that the ConDec model a (3..∗) contains a conflict when verification is carried out by choosing a bounded depth-first strategy with bound=2. If the bound is changed to 3, then g-sciff correctly states that the model is conflict-free, providing the sample execution trace a → a → a.

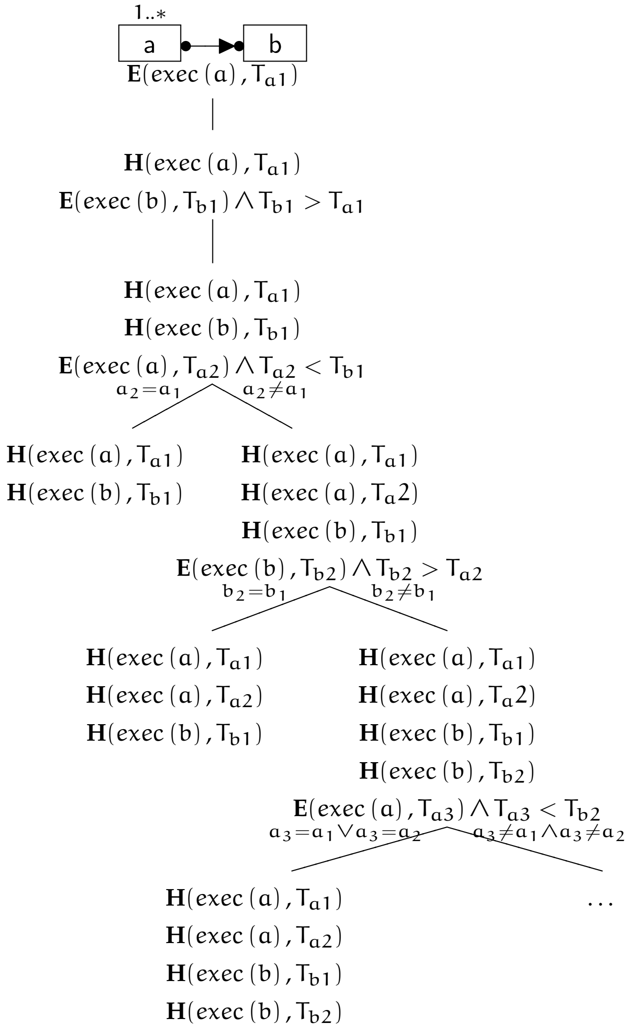


Figure 40: Part of the infinite branching proof tree produced by `g-sciff` when the model contains a succession constraint.

B. *it then fulfills such an expectation, generating a corresponding happened event* $\mathbf{H}(\mathsf{exec}(a), T_{a1})$ *and triggering the response part of the succession constraint* [a •—▶• b]*;*

C. *such a triggering leads to generate an expectation about a consequent execution of* b – $\mathbf{E}(\mathsf{exec}(b), T_{b1}) \wedge T_{b1} > T_a$*;*

D. `g-sciff` *fulfills this expectation (with a generation of* $\mathbf{H}(\mathsf{exec}(b), T_{b1})$*), triggering the precedence part of the succession constraint and generating a further expectation about a previous execution of activity* a – $\mathbf{E}(\mathsf{exec}(a), T_{a2}) \wedge T_{a2} < T_{b1}$*;*

E. *this expectation can be fulfilled by the intensional happened event generated at step* B – `g-sciff` *therefore applies the* E-fulfilllment *transition, generating two child nodes:*

a) *in the first node, fulfillment is imposed* ($T_{a2} = T_{a1}$) *– this is a success node, associated with the generated execution trace* a $\rightarrow$ b*;*

b) *in the second node, fulfillment is avoided (by imposing* $T_{a2} \neq T_{a1}$*), and a new happened event* $\mathbf{H}(\text{exec}(a), T_{a2})$ *is generated by applying the* fulfiller *transition.*

Since `g-sciff` *adopts a depth-first strategy, it terminates the computation in node* Ea*, with an open choice point pointing to node* E2*. If another solution is requested, i.e.,* `g-sciff` *is forced to explore node* Eb*, a new trace is generated, with another choice point left open. Every derivation leaves an open choice point which can be explored to provide a new execution trace, that inserts a new execution of* a *or* b *because of the inequalities constraints imposed by the* E-fulfillment *transition.*

The proof tree generated by `g-sciff` *resembles the structure shown in Figure 40*[13]*.*

Let us now suppose that [a] •—▶• [b] (1..*) *is part of a bigger ConDec model, and that this model contain a conflict. For a conflicting model, all the derivations produced by* `g-sciff` *are failure derivations, and therefore* `g-sciff` *tries to explore all the infinite branches of a proof tree similar to the one shown in Figure 40, experiencing the infinite branches anomaly and running forever.*

10.6.2 *Solving the Infinite Branches Anomaly*

To solve this issue, we provide a further revision to the formalization of forward and backward relation constraints. The idea is to insert inside each IC a negative expectation, such that the semantics of the IC w.r.t. compliance is not affected, but **E**-consistency can be exploited by `g-sciff` to prune the proof tree, making it finite.

In particular, we insert a negative expectation to formalize the intuitive fact that when the constraint triggers, the target is expected to happen, but the expectation will match only with the *nearest* event occurrence able to fulfill it. In this respect, constraint [a] •—▶ [b] is reformulated as "when activity a is executed, then b is expected to be executed afterwards, and the expectation will match with the first consequent occurrence of b":

$$t_{IC}\left([a] \bullet\!\!\longrightarrow\!\blacktriangleright [b]\right) \triangleq$$
$$\mathbf{H}(\text{exec}(a), T_a) \rightarrow \mathbf{E}(\text{exec}(b), T_b) \wedge T_b > T_a \wedge \mathbf{EN}(\text{exec}(b), T_{bn}) \wedge T_{bn} > T_a \wedge T_{bn} < T_b.$$

Similarly, the precedence constraint can be revised as follows:

$$t_{IC}\left([a] \longrightarrow\!\blacktriangleright\!\bullet [b]\right) \triangleq$$
$$\mathbf{H}(\text{exec}(b), T_b) \rightarrow \mathbf{E}(\text{exec}(a), T_a) \wedge T_a < T_b \wedge \mathbf{EN}(\text{exec}(a), T_{an}) \wedge T_{an} > T_a \wedge T_{an} < T_b.$$

13 The tree is simplified, and shows only the application of the *E-fulfillment* and *fulfiller* transitions.

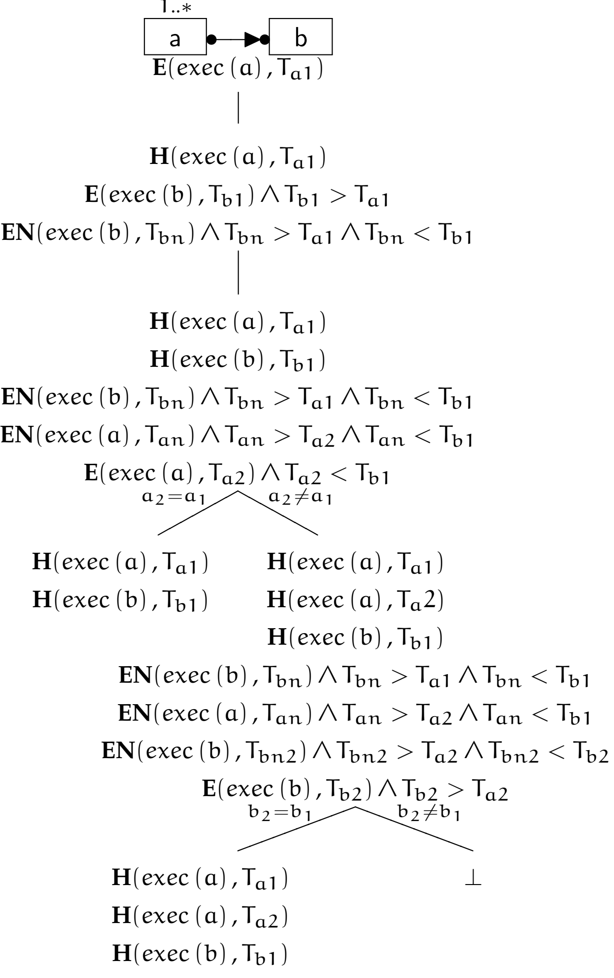


Figure 41: Complete proof tree produced by `g-sciff` when the model contains a succession constraint, and the revised formalizations is adopted. The infinite branches anomaly is not experienced anymore.

Such a revised formalization solves the infinite branching anomaly. Figure 41 shows how the infinite proof tree produced by `g-sciff` in Example 10.3 is modified when such a revised formalization is adopted.

It is worth noting that such a revised formalization can be seamlessly (and must be) applied to alternate constraints and to constraints with quantitative temporal conditions. For example, in case of a response with quantitative conditions, the updated formalization would be:

$$t_{IC}\left(\boxed{a} \xrightarrow{(n,m)} \boxed{b}\right) \triangleq$$

$$\mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b) \wedge T_b > T_a + n \wedge T_b < T_a + m$$
$$\wedge\, \mathbf{EN}(exec(b), T_{bn}) \wedge T_{bn} > T_a + n \wedge T_{bn} < T_b.$$

The revised formalization states that "the expectation will match the *first* occurrence of activity b *happening inside the requested time interval*".

11

EXPERIMENTAL EVALUATION

Contents

11.1 Verification Procedure with g-SCIFF **200**
11.2 Scalability of the g-SCIFF Proof Procedure **201**
11.2.1 The Branching Responses Benchmark 202
11.2.2 The Alternate Responses Benchmark 203
11.2.3 The Chain Responses Benchmark 206
11.3 Using Model Checking For the Static Verification of ConDec Models **209**
11.3.1 Model Checking 209
11.3.2 Verification of ConDec Properties By Satisfiability and Validity Checking 211
11.3.3 Reduction of Validity and Satisfiability Checking to Model Checking 213
11.3.4 Verification Procedure by Model Checking 214
11.4 Comparative Evaluation **215**
11.4.1 Evaluation Benchmarks 215
11.4.2 Experimental Results 216
11.5 Discussion **217**

Chapter 10 has discussed how `g-sciff` can deal with the static verification of ConDec models. In order to assess the usability of the proposed approach, a key point is to evaluate its performance and scalability, comparing it with other state-of-the-art verification techniques.

This Chapter is focused on such a topic: it aims at showing that `g-sciff` is an *effective* technology for dealing with static verification. An extensive experimental evaluation is presented, to stress `g-sciff` and emphasize its performance results in both favorable and unfavorable cases. After having discussed how the static verification task can be also interpreted as a model checking problem, we compare `g-sciff` with state-of-the-art explicit and symbolic model checkers, providing an empirical discussion on their advantages and drawbacks.

The benchmarks will stress scalability and performance of the verification techniques along two significative dimensions:

Dimensions used to evaluate verification technologies

SIZE OF THE MODEL *Number of mandatory constraints* contained in the model;

CARDINALITY ON ACTIVITIES Presence of existence constraints imposing a *minimum number of required executions* on the activities of the model.

```
function: CallGSCIFF(CLIMB specification S, initial trace T_i, integer
          Bound)
returns  : true, together with a sample execution trace T_f if there is a
          g-sciff successful derivation for S starting from T_i, false
          otherwise; the result is produced by adopting a depth-first
          strategy if Bound = −1, a bounded depth-first strategy with
          bound Bound otherwise
1  begin
2      if Bound ⩾ 0 then
3          Set the search strategy of g-sciff to
           bounded-depth-first(Bound);
4      else
5          Set the search strategy of g-sciff to depth-first;
6      end
7      if S_{T_i} ⊢^{T_f}_{g Δ} true then
8          return [true, T_f];
9      end
10     return [false, -];
11 end
```

Function g-sciff($\mathcal{S}$,$\mathcal{T}_i$,Bound)

Hardware used for the experiments

All experiments have been performed on a MacBook Intel CoreDuo 2 GHz machine.

11.1 VERIFICATION PROCEDURE WITH G-SCIFF

In Chapter 10, we have shown how the issue of $\exists$- and $\forall$-entailment of properties in the ConDec setting can be tackled by the g-sciff proof procedure, reducing them to conflict-freedom checking.

Starting from these theoretical results (Theorems 10.2 and 10.3 in particular), we now synthesize an effective procedure to verify $\exists$- and $\forall$-entailment of properties by g-sciff. The verification procedure are embedded in Functions $\exists$-entailment($\mathcal{M}$,Ψ) and $\forall$-entailment($\mathcal{CM}$,Ψ). The two procedures rely on a function CallGSCIFF($\mathcal{S}$,$\mathcal{T}_i$,Bound), which encapsulates the g-sciff computation on the specification $\mathcal{S}$, starting with $\mathcal{T}_i$ as initial trace and possibly providing a bound Bound in order to make g-sciff selecting a bounded depth-first search strategy.

The pre-processing procedure PreProcess is devoted to perform loop detection on ConDec models, augmenting them with absence constraints when $\wedge$-loops are contained, and returning whether a bounded search strategy must be chosen to guarantee termination (this happens when the model contains $\vee$-loop). All the details and the usage guidelines of this function have been deeply investigated in Section 10.5.3; the $\exists$-entailment and $\forall$-entailment functions strictly adheres to such guidelines.

```
   function: ∃-entailment(ConDec model CM, ConDec property Ψ)
   returns  : true, together with a sample execution trace if
              t_CLIMB(CM) ⊨∃ t_CLIMB(Ψ), false otherwise
1  begin
2  |  CMΨ ← COMP (CM, Ψ);
3  |  [CMΨ_Aug,Switch] ← PreProcess(CMΨ);
4  |  S ← t_CLIMB(CMΨ_Aug);
5  |  if ¬Switch then
6  |  |  return CallGSCIFF(S, ∅, −1);
7  |  else
8  |  |  Bound ← ask a bound to the user;
9  |  |  return CallGSCIFF(S, ∅, Bound);
10 |  end
11 end
```

Function ∃-entailment(CM,Ψ)

```
   function: ∀-entailment(ConDec model CM, ConDec property Ψ)
   returns  : true if t_CLIMB(CM) ⊨∀ Ψ, false, together with a
              counter-example execution trace, otherwise
1  begin
2  |  [CM_Aug,Switch] ← PreProcess(CM);
3  |  if Switch then
4  |  |  Bound ← ask a bound to the user;
5  |  else
6  |  |  Bound ← −1;
7  |  end
8  |  ⟨∅,ICAug⟩ ← t_CLIMB(CM_Aug);
9  |  foreach C_i ∈ Ψ do
10 |  |  IC_compl ← t_IC‾(C_i);
11 |  |  S ← ⟨∅,ICAug ∪ IC_compl⟩;
12 |  |  [Success, T] ← CallGSCIFF(S,∅,Bound);
13 |  |  if Success then
14 |  |  |  return [false, T];
15 |  |  end
16 |  end
17 |  return [true, -];
18 end
```

Function ∀-entailment(CM,Ψ)

11.2 SCALABILITY OF THE G-SCIFF PROOF PROCEDURE

A first quantitative evaluation is focused on "artificial" yet significative ConDec models, which employ different kind of relation constraints.

Why relation constraints impact on `g-sciff` *more than other constraints*

The focus on relation constraints rather than negation constraints is due to the nature of `g-sciff`: since it adopts a generative approach, the most time-consuming task is related to the generation of happened events from positive expectations and to the management of the consequent situation; positive expectations are involved only in the formalization of the existence constraint and of all the relation constraints.

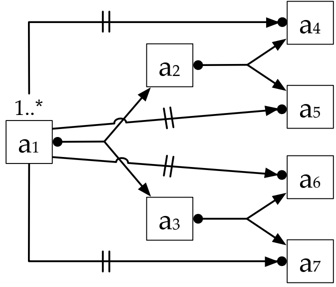


Figure 42: The *branching responses* benchmark when 7 activities and 8 constraints are employed.

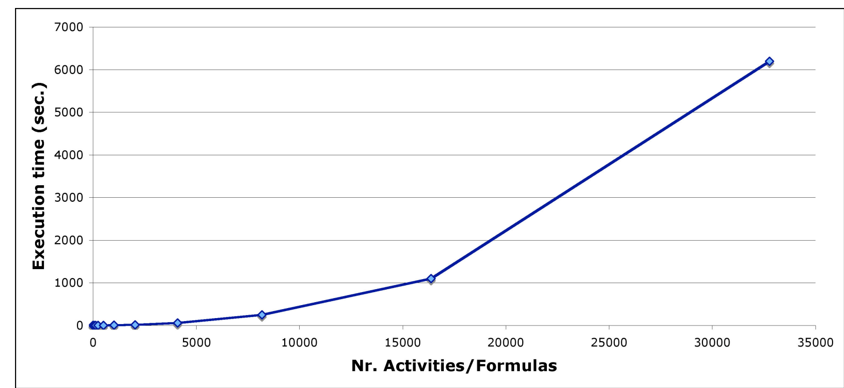


Figure 43: Trend of `g-sciff` when reasoning upon the *branching responses* benchmark.

# OF CONSTRAINTS	TIMES (SEC.)	# OF CONSTRAINTS	TIMES (SEC.)
4	0.00	512	1.04
8	0.01	1024	3.64
16	0.01	2048	14.01
32	0.02	4096	56.85
64	0.04	8192	249.77
128	0.10	16384	1100.26
256	0.34	32768	6149.99

Table 25: Timings employed by `g-sciff` to verify the *branching responses* benchmark.

11.2.1 *The Branching Responses Benchmark*

The first benchmark aims at evaluating the scalability of `g-sciff` when a growing number of branching response constraints is used. For this

reason, it will be referred as the *branching responses* benchmark. Each response has a branching-factor of 2. The structure of the model is as follows:

- the model contains an activity a_1 which is expected to be executed at least once;
- a_1 is source of a branching response, which states that one among activities a_2 and a_3 must be executed after a_1;
- both a_2 and a_3 are sources of a branching response as well, and this structure is repeated until a "frontier" is reached;
- all the activities belonging to this "frontier" have an outgoing negation precedence constraint pointing to a_1.

The benchmark is then built by increasing the number of activities connected by means of branching response constraints – Figure 42 shows the benchmark when 7 activities and 8 constraints are employed. Note that each instance of the model contains a conflict: after having executed activity a_1, no matter what choice is made to fulfill each triggered branching response, one among the activities in the "frontier" must be eventually executed; however, the execution of an activity in the "frontier" forbids the presence of a previous a_1.

The proof tree built by `g-sciff` when reasoning upon this benchmark strictly resembles the structure of the model itself, due to the depth-first strategy of the proof procedure. In the instance of the benchmark shown in Figure 42, `g-sciff` tries to generate, in order, the following execution traces:

A. $a_1 \rightarrow a_2 \rightarrow a_4$;

B. $a_1 \rightarrow a_2 \rightarrow a_5$;

C. $a_1 \rightarrow a_3 \rightarrow a_6$;

D. $a_1 \rightarrow a_3 \rightarrow a_7$.

All these attempts lead to a failure, because the execution of a_i ($i = 4, \ldots, 7$) forbids a previous execution of a_1; each failure is detected by `g-sciff` through the *consistency* transition.

In general, `g-sciff` must explore the entire model to detect the conflict, trying each possible combination of choices. Nevertheless, as attested by the timings reported in Table 25 and Figure 43, `g-sciff` scales very well: it is able to detect the presence of conflict on a model containing 4096 constraints in less than one minute.

11.2.2 *The Alternate Responses Benchmark*

This benchmark focuses on alternate response constraints and their interplay with the existence constraint; it will be referred as the *alternate responses* benchmark. Its structure is shown in Figure 44; it is parametric w.r.t. two values, N and K, and is characterized as follows:

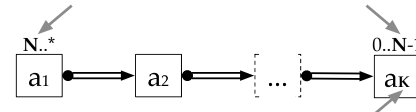


Figure 44: The *alternate responses* benchmark, parametrized on N and K.

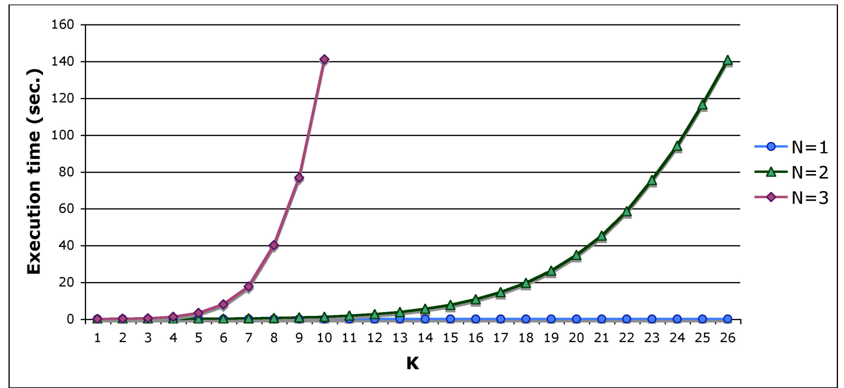


Figure 45: Trend of g-sciff when reasoning upon the *alternate responses* benchmark.

A. $K-1$ alternate response constraints are adopted to connect a sequence of K activities;

B. the first activity of the sequence is associated with an existence N..* constraint;

C. the last activity of the sequence is associated with an absence 0..N-1 constraint.

An interesting property of alternate response constraints is that the target activity must be executed at least as many times as the source activity is executed: each occurrence of the source requires a "dedicated" occurrence of the target activity. Hence, in the benchmark the number of executions of activity a_2 must be greater than the one of a_1 (i.e., always greater than N), the number of executions of activity a_3 must be greater than the one of a_2, and so on. This implies that a_K must be executed at least N times, but this contrasts with the absence constraint attached to it. As a consequence, each instance of the benchmark contains a conflict.

When reasoning upon the *alternate responses* benchmark, g-sciff must intensively apply the *consistency* transition and the underlying CLP solver. In fact, each generated occurrence triggers an expectation about the next one, together with a negative expectation expressing the interposition, i.e., stating that the activity cannot be executed again until the next activity is effectively executed (see the formalization provided in Section 10.4.1). By looking at the timings reported in Table 25 and the trends depicted in Figure 43, it is apparent that the N value is critical.

	time (sec.)				time (sec.)		
K	N=1	N=2	N=3	K	N=1	N=2	N=3
1	0.00	0.01	0.02	14	0.01	5.47	1263.34
2	0.00	0.02	0.10	15	0.01	7.65	2184.68
3	0.00	0.03	0.35	16	0.01	10.62	3010.70
4	0.00	0.05	1.12	17	0.01	14.53	>1 h
5	0.00	0.1	3.17	18	0.01	19.58	>1 h
6	0.00	0.16	7.96	19	0.01	26.16	>1 h
7	0.00	0.28	17.54	20	0.01	34.73	>1 h
8	0.01	0.45	40.03	21	0.01	45.28	>1 h
9	0.00	0.74	76.65	22	0.02	58.43	>1 h
10	0.01	1.11	140.95	23	0.02	75.37	>1 h
11	0.00	1.74	279.48	24	0.02	93.94	>1 h
12	0.00	2.58	460.38	25	0.02	116.44	>1 h
13	0.01	3.76	754.50	26	0.02	140.60	>1 h

Table 26: Timings employed by `g-sciff` to verify the *alternate responses* benchmark.

When $N = 1$, verification reduces to the case in which simple response constraints are adopted instead of alternate ones, because the interposition part is not used: for each activity, there is only a single generated occurrence. Hence, `g-sciff` answers almost immediately even for large values of K.

On the contrary, when N is increased, each one of the N required executions generates its own "interposition" expectation, which must be combined with the expectations about the other $N - 1$ occurrences of the same activity, to the aim of preserving **E**-consistency. In total, $N \times (N - 1) \times (K - 1)$ combinations must be considered. Such combinations require to impose, handle and propagate a huge amount of temporal constraints. In particular, `g-sciff` must try to compute each possible partial ordering among the activities in order to respect all the interpositions, before becoming aware that no solution actually exists.

Anyway, notice that it is rather uncommon to find models in which a certain activity is a-priori constrained by an existence formula to be executed always a huge amount of times; furthermore, when the modeler adopts many strict relationships (such as alternate response constraints) in the same diagram, she is breaking the ConDec philosophy, which aims at leaving the execution as unconstrained as possible. When a ConDec model tends to contain too many strict constraints, then it could be the case that the modeler has not chosen the right

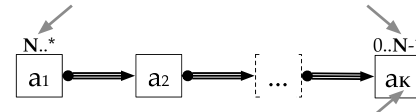


Figure 46: The *chain responses* benchmark, parametrized on N and K.

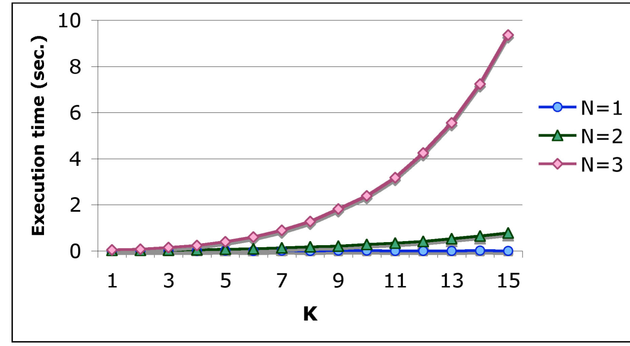


Figure 47: Trend of g-sciff when reasoning upon the *chain responses* benchmark by adopting a qualitative notion of time.

notation, and that a procedural approach would fit better with the problem she is trying to solve.

11.2.3 *The Chain Responses Benchmark*

The *chain responses* benchmark is a modification of the *alternate responses* one, where all the alternate response constraints are substituted with chain response constraints.

As pointed out in Sections 5.3 and 6.1, chain response constraints are the only elements of ConDec which need to be reformulated when switching from a qualitative to a quantitative notion of time, due to the different intended meaning of the *temporal contiguity* concept. We therefore carry out a comparison between the two formulations, to estimate the impact of adopting quantitative time instead of qualitative time.

By adopting qualitative time, the CLIMB formalization of a chain response constraint simply states that the target activity must be executed immediately after the source one, i.e., at the next time:

$$t_{IC}\left(\boxed{a} \Longrightarrow \boxed{b}\right) \triangleq \mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b) \wedge T_b == T_a + 1.$$

In this way, when g-sciff generates an (intensional) occurrence of the source activity, say, $\mathbf{H}(exec(a), T_{a1})$, then the generation of a corresponding occurrence of the target activity is generated with a fixed time equal to $T_{a1} + 1$. As a consequence, the behaviour of g-sciff when reasoning on the *chain responses* benchmark is the following:

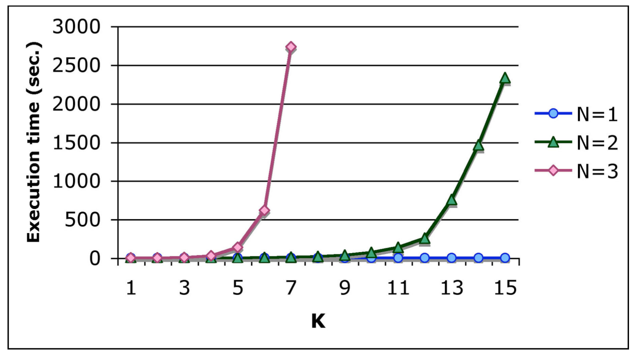


Figure 48: Trend of g-sciff when reasoning upon the *chain responses* benchmark by adopting a quantitative notion of time.

QUALITATIVE TIME (t_{CLIMB})			
	TIME (SEC.)		
K	N=1	N=2	N=3
1	0.00	0.01	0.04
2	0.00	0.02	0.07
3	0.00	0.03	0.15
4	0.00	0.05	0.24
5	0.10	0.07	0.39
6	0.00	0.09	0.60
7	0.01	0.13	0.89
8	0.00	0.17	1.27
9	0.01	0.21	1.81
10	0.01	0.28	2.38
11	0.00	0.33	3.18
12	0.00	0.41	4.25
13	0.00	0.53	5.55
14	0.01	0.65	7.24
15	0.00	0.78	9.36

(a) Qualitative time.

QUANTITATIVE TIME ($t^{\circlearrowright}_{CLIMB}$)			
	TIME (SEC.)		
K	N=1	N=2	N=3
1	0.00	0.01	0.03
2	0.00	0.02	0.35
3	0.00	0.09	4.04
4	0.00	0.34	27.12
5	0.01	1.13	134.77
6	0.00	3.02	616.94
7	0.00	7.63	2733.64
8	0.00	17.10	>1 h
9	0.01	37.09	>1 h
10	0.01	71.73	>1 h
11	0.01	137.03	>1 h
12	0.00	256.48	>1 h
13	0.01	756.34	>1 h
14	0.01	1460.47	>1 h
15	0.01	2332.65	>1 h

(b) Quantitative time.

Table 27: Timings employed by g-sciff to verify the *chain responses* benchmark.

A. an occurrence of a_1 is generated, and an expectation about the next activity is raised;

B. a sequence of expectations and of corresponding occurrences involving a_2-a_K is generated, where the time of each occurrence is completely determined by the occurrence of the previous activity;

C. the process is repeated until the N-th expectation about a_K is generated;

D. as soon as this last expectation is generated, `g-sciff` detects the presence of a conflict, caused by the **E**-inconsistency between such an expectation and the negative expectation expressing that at most $N-1$ executions of a_K are allowed.

Since each generated occurrence of activities a_2-a_K has a completely determined time, such a time is not subject to case analysis and non-determinsm when the *E-fulfillment* transition is applied.

As shown in Figure 47 and Table 27(a), `g-sciff` is therefore very fast and scalable when reasoning upon the benchmark by adopting the qualitative formalization of chain responses: it employs less than 10 seconds when the model contains 15 chain response constraints and imposes that the first activity must be executed at least 3 times. As in the case of the *alternate responses* benchmark, the most influencing parameter is N: when it changes from 2 to 3, the performance of `g-sciff` degrades of one order of magnitude.

When temporal contiguity is modeled in a quantitative manner, the chain response constraint is modeled by including a negative expectation forbidding the occurrence of *all* activities between the executions of the source and the target activity:

$$t_{IC}^{\circ}\left(\boxed{a} \bullet\!\!\Longrightarrow \boxed{b}\right) \triangleq \mathbf{H}(exec(a), T_a) \rightarrow \mathbf{E}(exec(b), T_b) \wedge T_b > T_a \wedge \mathbf{EN}(exec(X), T_x) \wedge T_x > T_a \wedge T_x < T_b.$$

Such a negative expectation is even more difficult to be handled than the one used to express the "interposition" behaviour of alternate constraints, because its involved activity is a variable (X), and thus matches with all the possible ground activities. Timings and trend experienced by `g-sciff` when reasoning upon the *chain responses* benchmark are reported in Figure 48 and Table 27(b), and clearly points out such a difficulty.

Indeed, each occurrence of activity a_1 triggers a sequence of $K-1$ positive expectations (and corresponding executions) on a_1-a_K, together with $K-1$ negative expectations used to impose the temporal contiguities. When $N > 1$, multiple sequences are generated, and all the negative expectations of a sequence must be combined with the positive expectations of the other sequences, to guarantee that all temporal contiguities are respected. `g-sciff` must therefore suitably carry out $N \times (N-1) \times (K-1)^2$ combinations, making an extensive use of the underlying CLP solver.

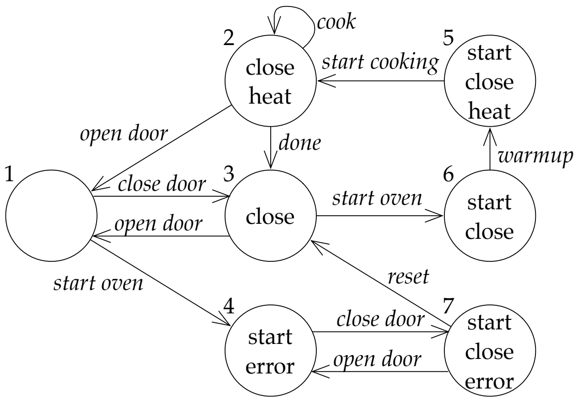


Figure 49: Kripke structure of a micro-wave oven (from [57]).

11.3 USING MODEL CHECKING FOR THE STATIC VERIFICATION OF CONDEC MODELS

By relying on the original formalization of ConDec in terms of propositional Linear Temporal Logic (LTL) formulas[157, 158], we discuss how the static verification of ConDec models can be carried out by means of model checking techniques. This opens the possibility of comparing the performance and scalability of `g-sciff` with the ones of state-of-the-art model checkers.

11.3.1 *Model Checking*

Generally speaking, *model checking*[57] designates a collection of technique for the automatic verification of finite state concurrent systems. Model checking started in the 1980s from seminal research conducted by Clarke and Emerson and followed by many others, and has found widespread application in the hardware and software industries, ranging from verification of security and cryptographic protocols to debugging of software programs and digital circuits.

The model checking process

The model checking process consists of three steps[57]:

MODELING The first task that must be accomplished is to convert the design of the system into a formalism accepted by the model checking tool. The most widespread formalism is the Kripke structure, a non-deterministic finite state machine in which each state is labeled with the set of propositions true in that state. An example of Kripke structure is given in Figure 49. Kripke structures may support infinite execution traces, and can be translated to automata on infinite words.

SPECIFICATION The properties that the design must satisfy are expresses in some logical formalism. Temporal logics, such as LTL, are commonly employed to accomplish this task, because they

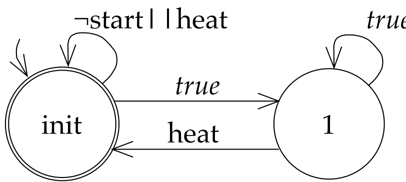


Figure 50: A non-deterministic Büchi automaton representing the LTL formula $\Box(\mathsf{start} \Rightarrow \Diamond \mathsf{heat})$, produced by the LTL2BA algorithm[83].

support a powerful yet intuitive way to characterize the evolution of the system over time.

VERIFICATION Verification is carried out *automatically*, by checking whether the desired properties are entailed by the model of the system. If a property is not met, then the model checker provides an error execution trace, which amounts as a counter-example useful to identify where the error is located. For example, the Kripke structure of Figure 49 could be verified against the property $\Box(\mathsf{start} \Rightarrow \Diamond \mathsf{heat})$, specifying that it must always be guaranteed that every time the oven is started, then it must eventually heat. A model checker would state that the property does not hold, providing e.g. an error trace starting from state 1 and containing the infinite sequence start oven → close door → reset → open door → start oven → ....

Formally, given a Kripke structure $\mathcal{M}$ representing the model of the system and a temporal property φ, the model checking problem is[1]

$$\mathcal{M}, s \models \varphi$$

where s represents a state of the model from which the property must hold (usually, s is one of the initial states of the Kripke structure – in this case, it can be omitted). If $\mathcal{M} \not\models \varphi$, then a counter-example is returned by the model checker as a proof.

Major advantages of model checking

Among the advantages of model checking techniques, we find that:

A. they do not require the intervention of the user during the verification phase, i.e., they are fully automated;

B. they can handle properties on infinite-length executions;

C. they always guarantee that an answer is generated in finite time.

Explicit model checking

Classical model checkers, called *explicit*-state, face the model checking problem by relying on the translation of a temporal logic formula onto a Büchi automaton, and by intersecting such an automaton with the automaton obtained from the Kripke structure of the model [99]. Figure Figure 50 shows how the LTL formula $\Box(\mathsf{start} \Rightarrow \Diamond \mathsf{heat})$ can be

1 It is worth noting that the $\models$ symbol does not denote, in this setting, entailment or logical consequence.

translated to a non-deterministic Büchi automaton. SPIN[2][99] is one of the most popular state-of-the-art explicit model checker.

The state explosion problem

The major drawback of explicit model checking is known as the *state-explosion problem*[58], which essentially states that the Kripke structure is exponential in the size of the system description, and thus that modeling even small systems leads to produce intractable Kripke structures; for example, in a concurrent system the Kripke structure is exponential in the number of interacting processes. State explosion is also experienced when the property to be verified is too large: the construction of a Büchi automaton starting from an LTL formula is exponential in the size of the formula [57, 64]. From a theoretical point of view, the time and space complexity of LTL model checking is

$$\mathcal{O}\left(|\mathcal{M}| \times 2^{|\varphi|}\right)$$

where $\mathcal{M}$ is the Kripke structure of the model, and φ the property to be verified.

Abstraction techniques and symbolic model checking

The state explosion problem motivated research about how to represent the states in a compact way. Two major mainstream approaches emerged in the last decades[58]:

ABSTRACTION TECHNIQUES exploit the domain knowledge on the system and the desired properties to insert in the Kripke structure only the relevant portions of the system.

SYMBOLIC VERIFICATION employs more compact data structures to store the Kripke structure and the translation of the properties, without experiencing information loss. As stated in [58]: " *A fundamental breakthrough* [w.r.t. the state explosion problem] *was made in the fall of 1987 by Ken McMillan, who was then a graduate student at Carnegie Mellon. He argued that larger systems could be handled if transition relations were represented implicitly with ordered Binary Decision Diagrams (BDDs) [37]. By using the original model checking algorithm with the new representation for transition relations, he was able to verify some examples that had more than states [39]*". NuSMV [55] is one of the most popular state-of-the-art symbolic model checkers.

11.3.2 *Verification of ConDec Properties By Satisfiability and Validity Checking*

In the ConDec setting, LTL properties represent the formalization of the ConDec model under study, which is a conjunction formula built by applying the t_{LTL} function (see Section 3.7). In other words, the model of the system is not represented by a (procedural) Kripke structure, but it is itself specified by means of (declarative) LTL formulae. In order to employ state-of-the-art model checkers for the static verification of ConDec models, it is therefore necessary to precisely identify which

2 `http://spinroot.com`

kind of verification must be carried out, and how it can be encoded as a model checking problem.

In the ConDec setting, the model checking problem is grounded as follows:

$$\mathcal{M}, s \models t_{LTL}(\mathcal{CM})$$

where $t_{LTL}(\mathcal{CM})$ is the LTL formalization of the ConDec model $\mathcal{CM}$, and $\mathcal{M}$ is a Kripke structure modeling possible execution traces; a positive answer is obtained iff all the traces which can be produced by $\mathcal{M}$ are supported by $\mathcal{CM}$.

Conflict-freedom as a satisfiability problem

In this respect, the problem of checking if a ConDec model is conflict-free can be reduced to LTL *satisfiability* checking.

Definition 11.1 (Satisfiability problem). An LTL formula is *satisfiable* iff there exists *at least one* Kripke structure which satisfies the formula.

However, it is important to remember that model checking and temporal logics work on infinite-length execution traces, while supported ConDec executions must always eventually terminate. E.g., a conflict-freedom test providing an infinite trace as a sample showing that it is possible to meet all the ConDec constraints in at least one way, would be a wrong response, because such a trace cannot be reproduced in reality. The LTL encoding of such a *termination property* has been provided in Definition 8.5 – Page 139; we briefly recall the definition for the sake of readability; given a ConDec model $\mathcal{CM} = \langle \mathcal{A}, \mathcal{C}_m, \mathcal{C}_o \rangle$, the LTL *termination property* $\mathtt{term}(\mathcal{CM})$ states that a termination event $e \notin \mathcal{A}$, incompatible with all other activities, must eventually occur, and then is executed infinitely often in the future:

$$\mathtt{term}(\mathcal{CM}) \triangleq \Diamond e \wedge \Box(e \Rightarrow \bigcirc e) \wedge \forall\ a \in \mathcal{A},\ \Box(e \Rightarrow \neg a)$$

The Conlict-freedom issue is then faced by checking if the conjunction of the LTL formula representing the model with its termination property is satisfiable.

Definition 11.2 (Conflict-freedom checking in LTL). A ConDec model $\mathcal{CM}$ is conflict-free iff $t_{LTL}(\mathcal{CM}) \wedge \mathtt{term}(\mathcal{CM})$ is satisfiable.

∃-entailment as a satisfiability problem

When carrying out properties verification, it is therefore not sufficient to combine the LTL formula representing the ConDec model with the LTL formula encoding the property, but the implicit *termination* property must be explicitly taken into account as well.

Definition 11.3 (∃-entailment in LTL). Given a ConDec model $\mathcal{CM}$ and a ConDec property Ψ,

$$t_{LTL}(\mathcal{CM}) \models_{\exists} t_{LTL}(\Psi) \Leftrightarrow t_{LTL}(\textsc{comp}(\mathcal{CM}, \Psi)) \wedge \mathtt{term}(\mathcal{CM}) \text{ is satisfiable}$$

∀-entailment as a validity problem

Differently from ∃-entailment, ∀-entailment of a property Ψ instead states that *all* the possible Kripke structures supported by the ConDec

model augmented with the termination property, also satisfy Ψ. In the LTL setting, this issue is called *validity* problem[3].

DEFINITION 11.4 (Validity problem). An LTL formula is valid iff every possible Kripke structure satisfies the formula.

$\forall$-entailment is therefore encoded as a validity problem where the formula is an implication whose body is the LTL formula of the ConDec model plus the termination property and the head is the LTL formula of the ConDec property.

DEFINITION 11.5 ($\forall$-entailment in LTL). Given a ConDec model $\mathcal{CM}$ and a ConDec property Ψ,

$$\mathtt{t}_{LTL}(\mathcal{CM}) \models_{\forall} \mathtt{t}_{LTL}(\Psi) \Leftrightarrow (\mathtt{t}_{LTL}(\mathcal{CM}) \wedge \mathtt{term}(\mathcal{CM}) \Rightarrow \mathtt{t}_{LTL}(\Psi)) \text{ is valid}$$

For example, the process of checking if d is a dead activity in the model shown in Figure 37(a), is tackled as follows. First, the $\overset{0}{\boxed{\mathtt{d}}}$ property is expressed in LTL as $\Box\neg d$. Second, we check whether the formula

$$\mathtt{t}_{LTL}(\mathcal{CM}) \wedge \mathtt{term}(\mathcal{CM}) \Rightarrow \Box\neg \mathtt{d}$$

is valid. The answer is positive: there does not exist a finite execution trace supported by the model which contains d.

If we erroneously forget to take into account the $\mathtt{term}(\mathcal{CM})$ property, then we obtain a wrong answer: formula

$$\mathtt{t}_{LTL}(\mathcal{CM}) \Rightarrow \Box\neg \mathtt{d}$$

is not valid, because the infinite-lenght trace $\mathtt{d} \to \mathtt{c} \to \mathtt{a} \to \mathtt{b} \to \mathtt{c} \to \ldots$ contains an execution of activity d. The problem is that such a sample cannot be reproduced in reality.

11.3.3 *Reduction of Validity and Satisfiability Checking to Model Checking*

In order to deal with static verification of ConDec by exploiting state-of-the-art model checkers, satisfiability and validity checking must be expressed as model checking problems.

Universal Kripke structure

Rozier and Vardi tackled this issue in [169], showing how LTL satisfiability and validity checking can be reduced to model checking. The reduction process centers around the construction of an *universal* Kripke structure.

DEFINITION 11.6 (Universal Kripke structure). Given a set of proposition symbols $\mathcal{S}$, the *universal Kripke structure* on $\mathcal{S}$ (written $\mathcal{U}(\mathcal{S})$) is a Kripke structure able to generate all the possible execution traces over the proposition symbols belonging to $\mathcal{S}$.

3 Note that validity and satisfiability have a strong correspondence: an LTL formula φ is valid if $\neg\varphi$ is unsatisfiable.

```
    function: ∃-entailment_MC(ConDec model CM, ConDec property Ψ)
    returns : true, together with a sample LTL execution trace if
              t_LTL(CM) ⊨_∃ t_LTL(Ψ), false otherwise
 1  begin
 2  |  ψμ ← t_LTL(COMP(CM, Ψ));
 3  |  ← t_LTL(Ψ);
 4  |  U ← universal(P(ψμ) ∪ P(term(CM)));
 5  |  [Success, T_L] ← model_checking(U, ¬(ψμ ∧ term(CM)));
 6  |  if ¬Success then
 7  |  |  return [true, T_L];
 8  |  else
 9  |  |  return [false, -];
10  |  end
11  end
```

Function $\exists$-entailment$_{MC}$($\mathcal{CM}$,Ψ)

Thus, given the LTL formula μ representing a ConDec model, the corresponding universal Kripke structure is obtained by considering all the proposition symbols appearing in μ, i.e., $\mathcal{P}(\mu)$ (see Definition 5.10 – Page 101).

Given an LTL formula φ,

- validity checking can be tackled by model checking φ against $\mathcal{U}(\varphi)$: if the model checker returns a negative answer, the generated counter-example shows that there exists a possible execution trace violating φ, i.e., that φ is not valid;

- satisfiability checking can be instead tackled by model checking $\neg\varphi$ against $\mathcal{U}(\varphi)$: if the model checker returns a negative answer, the generated counter-example is actually a positive example for the original formula φ, proving that there exists at least one possible execution satisfying φ, i.e., that φ is satisfiable.

Definition 11.7 (Validity checking via model checking[169]). An LTL formula φ is valid iff $\mathcal{U}(\varphi) \models \varphi$.

Definition 11.8 (Satisfiability checking via model checking[169]). An LTL formula φ is satisfiable iff $\mathcal{U}(\varphi) \not\models \neg\varphi$.

11.3.4 *Verification Procedure by Model Checking*

Functions $\exists$-entailment$_{MC}$($\mathcal{CM}$,Ψ) and $\forall$-entailment$_{MC}$($\mathcal{CM}$,Ψ) implement the $\exists$- and $\forall$-entailment of properties in the LTL setting, by reducing satisfiability and validity to model checking.

Besides the function t_{LTL}, which maps a ConDec model to the corresponding LTL conjunction formula, two others support functions are used:

- universal(φ) builds a universal Kripke structure able to generate all the execution traces over the proposition symbols of φ, i.e.,

```
function: ∀-entailment_MC(ConDec model CM, ConDec property Ψ)
returns : true if t_LTL(CM) ⊨_∀ t_LTL(Ψ), false, together with an
          execution trace amounting as a counter-example, otherwise
1 begin
2   μ ← t_LTL(CM);
3   ψ ← t_LTL(Ψ);
4   U ← universal(P(CM) ∪ P(ψ) ∪ P(term(CM)));
5   return model_checking(U, μ ∧ term(CM) ⇒ ψ);
6 end
```

Function $\forall$-entailment$_{MC}$($\mathcal{CM}$,Ψ)

over all the activities of the ConDec model and the property under study;

- model_checking($\mathcal{M}$,φ) model checks $\mathcal{M}$ againts φ, returning true if $\mathcal{M}$ meets φ in every possible execution, false, together with a counter-example, otherwise.

11.4 COMPARATIVE EVALUATION

In order to choose a suitable model checker and run a comparative evaluation of g-sciff with the state of the art, we referred to the results of an experimental investigation conducted by Rozier and Vardi on *LTL* satisfiability checking [169]. The authors found that the symbolic approach is clearly superior to the explicit approach, and that NuSMV is the best performing model checker for the benchmarks they considered. We thus chose NuSMV and ran our benchmarks to compare g-sciff with it.

In fact, some preliminary tests we carried out by using the SPIN explicit model checker confirmed the results of Rozier and Vardi: SPIN could not handle in reasonable time even a ConDec chart and properties as simple as the ones we described in Section 8.3.4. g-sciff, instead, correctly handles all the queries described in Section 8.3.4 with the following timings: Query 8.1 in 10 ms, Query 8.2 in 20 ms, Query 8.3 in 420ms and Query 8.4 in 80ms.

Unfortunately, the comparison could not cover all relevant aspects of the language, such as quantitative temporal aspects (presented in Query 8.4), because neither NuSMV nor any other model checker cited in [169] supports Metric Temporal Logic (MTL). Anyway, since existing MTL tools seem to use explicit model checking and not symbolic model checking, our feeling is that g-sciff would largely outperform them on these instances.

11.4.1 *Evaluation Benchmarks*

To obtain our evaluation benchmarks, we complicated the model shown in Section 8.3.4 – Figure 59 so as to stress g-sciff and emphasize its performance results in both favorable and unfavorable cases. In particular,

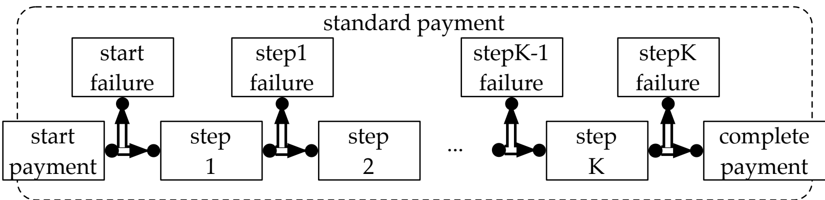
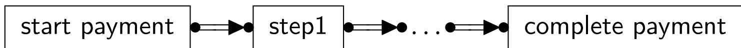


Figure 51: Parametric extension to the model presented in Figure 59.

as in the *alternate responses* and *chain responses* benchmark (see Sections 11.2.2 and 11.2.3), verification techniques are stressed along two axes:

SIZE OF THE MODEL Instead of a single activity, standard payment consists of a chain of K activities, in which every two consecutive steps are linked by an alternate succession relation:



Furthermore, we model a possible failure at each of these steps (start failure, step 1 failure, ...), thus making each alternate succession disjunctive. This extension to the model is depicted in Figure 51.

NUMBER OF REQUIRED EXECUTIONS We add an existence N..* constraint on activity payment failure, simulating that payment failure must occur at least N times.

11.4.2 *Experimental Results*

The comparison between g-sciff and NuSMV has been carried out on the parametrized version of Figure 59, by focusing on two sets of benchmarks:

1. Query 8.1, augmented with an absence constraint on the start payment activity [0 start payment], to make the property unsatisfiable (benchmark EXP_1);
2. Query 8.1 (benchmark EXP_2).

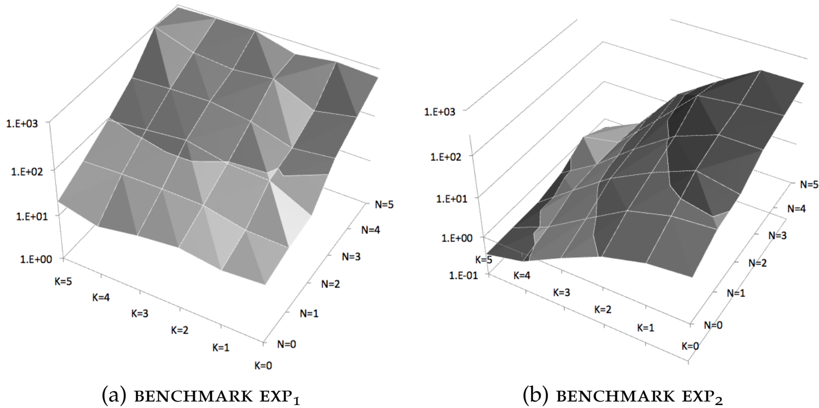
Of the two benchmarks, the first one concerns verification of an unsatisfiable property and the second one verification of an ∃-entailed property. The latter requires producing an example demonstrating ∃-entailment, which generally increases the runtime. The input files are available on a Web site[4]. The runtime resulting from the benchmarks has been presented in [144] and is reported in Table 28. Figure 52 shows the ratio NuSMV/g-sciff runtime, in Log scale.

It turns out that g-sciff outperforms NuSMV in most cases, up to several orders of magnitude. This is especially true for the first benchmark, for which g-sciff is able to complete the verification task always

4 See http://www.lia.deis.unibo.it/research/climb/iclp08benchmarks.zip.

N \ K	0	1	2	3	4	5
			benchmark exp_1			
0	0.01/0.20	0.02/0.57	0.03/1.01	0.02/3.04	0.02/6.45	0.03/20.1
1	0.02/0.35	0.03/0.91	0.03/2.68	0.04/4.80	0.04/8.72	0.04/29.8
2	0.02/0.46	0.04/1.86	0.05/4.84	0.05/10.8	0.07/36.6	0.07/40.0
3	0.03/0.54	0.05/2.40	0.06/8.75	0.07/20.1	0.09/38.6	0.10/94.8
4	0.05/0.63	0.05/2.34	0.08/9.51	0.10/27.1	0.11/56.63	0.14/132
5	0.05/1.02	0.07/2.96	0.09/8.58	0.12/29.0	0.14/136	0.15/134
			benchmark exp_2			
0	0.02/0.28	0.03/1.02	0.04/1.82	0.05/5.69	0.07/12.7	0.08/37.9
1	0.06/0.66	0.06/1.67	0.07/4.92	0.08/9.21	0.11/17.3	0.15/57.39
2	0.14/0.82	0.23/3.44	0.33/8.94	0.45/22.1	0.61/75.4	0.91/72.86
3	0.51/1.01	1.17/4.46	1.87/15.87	3.77/41.2	5.36/79.2	11.4/215
4	1.97/1.17	4.79/4.43	10.10/17.7	26.8/52.2	61.9/116	166/268
5	5.78/2.00	16.5/5.71	48.23/16.7	120/60.5	244/296	446/259

Table 28: Results of the benchmarks (SCIFF/NuSMV), in seconds [144].



(a) benchmark exp_1 (b) benchmark exp_2

Figure 52: Charts showing the ratio NuSMV/`g-sciff` runtime, in Log scale.

in less than 0.15s, while NuSMV takes up to 136s. For the second benchmark, `g-sciff` does comparatively better as K increases, for a given N, whereas NuSMV improves w.r.t. `g-sciff` and eventually outperforms it, for a given K, as N increases.

11.5 discussion

Exploration of the search space

The main difference of `g-sciff` with model checking is that queries are evaluated top-down, i.e., starting from expectations and using abduc-

tion as a mechanism to simulate events. No intermediate format needs to be generated, which eliminates a computationally expensive step. By going top-down, the verification algorithm only considers relevant portions of the search space, which can boost performance. On the downside, the performance strongly depends on the way CLIMB programs are written w.r.t. the property. Due to the left-most, depth-first search tree exploration strategy that `g-sciff` inherits from Prolog, the order of clauses influences performance, and so does the ordering of atoms inside the clauses. However, this does not impact on soundness, completeness and termination.

In particular, since verification is performed by `g-sciff` starting from expectations, its performances are heavily influenced by the presence of existence/choice constraints in the ConDec model and query. Indeed, existence and choice constraints are the ones that, translated to CLIMB, impose expectations about the execution of a certain activity independently from the other activities and constraints. At the beginning of the `g-sciff` computation, these expectations are transformed to happened events via the *fulfiller* transition. These happened events, in turn, trigger new parts of the model under study, leading to the generation of new expectations and happened events. This is why the performance of `g-sciff` decreases as the N value of the benchmarks increases: all the N expected executions are simulated, triggering the outgoing relationships N times, and so on. On the other side, if a portion of the ConDec model is not affected by this propagation, i.e., its execution is not mandatory, then its constraints do not affect verification at all. The extreme case is the one in which no activity is expected to be executed by the ConDec model nor by the (complemented) query: in this situation, independently of the size of the model, `g-sciff` answers immediately by returning the void execution trace as an example. This is the case, for example, when presence of conflict is checked on the order&payment diagram shown in Figure 59. This smart exploration of the search space motivates why when K (i.e., the number of constraints) increases, performances of `g-sciff` degrade gracefully. Furthermore, it suggests that suitable heuristics that choose how to explore the search tree could help to improve the `g-sciff` performance. This is subject for future research.

Differently from `g-sciff`, model checking techniques first translate the formula representing the ConDec model and the query to an intermediate structure: a Büchi automaton in the case of explicit model checking, a BDD in the case of symbolic model checking. As we have already pointed out, this translation is exponential in the size of the formula [57, 64]. When model checking is adopted to check if declarative specifications are satisfiable, this is a critical point, because both the model of the system and the property are represented by means of LTL formulae. Even if symbolic model checking experiences a more graceful degradation rather than explicit model checking (mainly due to the compactness of BDDs), it cannot avoid state explosion when verifying declarative specifications such as the ones produced by translating ConDec models to LTL. Furthermore, practical experiments show

that the performance of symbolic methods is highly unpredictable; this phenomenon can be partially explained by complexity theoretical results which state that the use of BDDs does not improve worst case complexity [58].

It then comes as no surprise that NuSMV experiences an exponential degradation as the K value of our benchmark increases: K reflects the number of constraints contained in the ConDec model, and thus indirectly also the size of the underlying LTL formalization. When N increases, the degradation is more graceful; indeed, an existence constraint stating that activity a must be executed at least N times is modeled in LTL by applying the (recursive) translation

$$t_{LTL}\left(\overset{N..*}{\boxed{a}}\right) \triangleq \Diamond\left(a \wedge \Diamond t_{LTL}\left(\overset{N-1..*}{\boxed{a}}\right)\right)$$

and the the size of the formula does not dramatically increase from $N-1$ to N.

Termination issues

A great advantage of model checking w.r.t. `g-sciff` is that it always guarantees that an answer is provided in finite time. In this respect, while `g-sciff` needs to be accompanied by a pre-processing procedure to avoid non-termination issues (for the specific case of ConDec), and it must necessarily switch to a bounded (incomplete) search strategy when the ConDec model contains $\bigvee$-loops, model checking techniques can seamlessly deal with infinite-lenght computations: the implict requirement that "each supported execution must eventually terminate" can be made explicit in the logic, and combined with the formulae formalizing the model and the property. If ConDec had been extended with new constraints, then `g-sciff` would require to update the preprocessing procedure, while model checking techniques would be able to seamlessly reason upon such a new capabilities.

Expressiveness of the formalisms

However, it is important to point out that non-termination issues experienced by `g-sciff` are due to its first-order nature: the CLIMB language is able to express metric and data-related constraints by exploiting variables, a feature which is not available for state-of-the-art model checkers, which employ propositional temporal logics. All the features added in ConDec^{++} can be expressed in CLIMB and verified by applying `g-sciff`, while they cannot be tackled by LTL, thus making model checking inapplicable. Enlarging the range of features that can be expressed by the temporal logic leads model checking to experience the same drawbacks of `g-sciff`; just to cite an example, the translation of a temporal logic formula to the underlying automaton becomes undecidable for variants of temporal logic with explicit time, such as Metric Temporal Logic (MTL) with dense time[9].

Last but not least, as we will see in the next part of this dissertation, the same CLIMB formalization produced for `g-sciff` can be used in conjunction with the `sciff` proof procedure enabling monitoring and run-time verification capabilities. This eliminates the problem of having to produce two sets of specifications (one for static and one for run-time verification) and of verifying that they are equivalent.

12

RELATED WORK AND SUMMARY

Contents

12.1 Related Work **221**
12.1.1 Verification of Properties 221
12.1.2 A-priori Compliance Verification 225
12.1.3 Model Composition 227
12.1.4 Interoperability and Choreography Conformance 228
12.2 Summary of the Part **229**

In this Chapter, related work concerning the topic of static verification of interaction models and their composition is presented. Then, the major contributions of this part of the dissertation are briefly summarized.

12.1 RELATED WORK

We present related work by dedicating particular attention to the following research areas:

- static verification of properties;
- a-priori compliance verification;
- formalisms and reasoning techniques for model composition;
- approaches dealing with interoperability and choreography conformance issues.

12.1.1 *Verification of Properties*

Existing formal verification tools rely on model checking or theorem proving. A drawback of most model checking tools is that they typically only accommodate discrete time and range on finite domains, and that the cardinality of domains impacts heavily on their performance, especially in relation to the production of an automaton (or other more compact structures such as BDDs) starting from a temporal logic formula. On the other hand, theorem proving in general has a low level of automation, and it may be hard to use, because it heavily relies on the user's expertise [97]. `g-sciff` presents interesting features from

both approaches. Like theorem proving, its performance is not heavily affected by domain cardinality, and it accommodates domains with infinite elements, such as dense time. Similarly to model checking, it works in a push-button style, thus offering a high level of automation.

In [159], Pesic et al. describe an integrated framework in which ConDec models can be graphically specified, automatically obtaining the underlying LTL formalization, which is then used for enactment and verification purposes. Verification is limited to conflict-freedom and discovery of dead activities, and does not support ∃- nor ∀-entailment of properties. Instead of adopting standard model checking techniques, verification is carried out by exploiting the finite-trace model checking approach proposed by Giannakopoulou and Havelund in [86]. The approach reviews the semantics of LTL in a finite setting, adding an (ideal, i.e., not fixed) upper bound on the application of temporal operators. For example, the semantics of the $\mathcal{U}$ operator is revised as follows (n is the upper bound):

$$\begin{aligned} \mathcal{T}_{\mathcal{L}}, i \models_{\mathcal{L}} \psi \mathcal{U} \phi \text{ iff } \exists\, k,\ i \leqslant k \leqslant n \text{ s.t. } (\mathcal{T}_{\mathcal{L}}, k \models_{\mathcal{L}} \phi) \\ \wedge \forall i \leqslant j < k\ (\mathcal{T}_{\mathcal{L}}, j \models_{\mathcal{L}} \psi) \end{aligned}$$

The authors then modify a translation algorithm, which produces a Büchi automaton starting from an LTL formula, to reflect such a finite-trace semantics. The modified automaton embeds by construction the ConDec termination condition stating that each supported execution of the model must eventually terminate. This enables the possibility of performing verification without explicitly asserting the ConDec termination condition in the logic. Nevertheless, the approach presents the same computational drawbacks of explicit model checking: the translation phase has a cost which is exponential in the size of the formula, thus making it not suitable for the verification of even small-sized models.

As pointed out in Section 11.4, the quantitative evaluation presented in Chapter 11 does not cover all aspects of ConDec^{++}, which contains data-related constraints, supports a non-atomic model of activities and accomodates quantitative time constraints. Different extensions of propositional LTL have been proposed to explicitly referencing time and expressing quantitative time constraints. For example, the timed requirement of Query 8.4, Figure 35, stating that a receipt is expeted by 12 time units after having executed accept advert, can be expressed in MTL[9] as:

$$\Box(accept_advert \Rightarrow \Diamond_{\leqslant 12} send_receipt)$$

which is equivalent to the Timed Propositional Temporal Logic (TPTL)[10] formula:

$$\Box x.(accept_advert \Rightarrow \Diamond y.(y - x \leqslant 12 \wedge send_receipt))$$

Many tools have been developed to verify real-time systems w.r.t. timed temporal logics, relying on timed automata [25]. For example, UPPAAL [11] is an integrated environment for modeling and verifiying

real-time systems as networks of timed automata; it supports a limited set of temporal logic properties to perform reachibility tests. A timed automaton is a Büchi automaton extended with a set of real-valued (constrained) variables modeling clocks. As in standard explicit model checking, building and exploring (product of) timed automata is a very time and space-consuming task, made even more complex due to presence of such clocks. `g-sciff` incorporates CLP solvers to deal with metric temporal constraints, and therefore the complexity of verification does not change if quantitative time constraints are involved.

Among the temporal logic-based languages able to specify quantitative time constraints, we cite TRIO[84], a language, based on a metric extension of first-order temporal logic, for modeling critical real-time systems. Similarly to ConDec, TRIO systems are modeled in a declarative manner, i.e., as a conjunction of TRIO formulae. Different approaches have been investigated for model checking TRIO specifications, but many important features of the initial language are lost in the effort to obtain a decidable and tractable specification language. For example, the time domain is reduced to natural numbers, there is no quantification over time variables, and the language can range only on finite domains. Such a restricted language can be translated onto a Promela alternating Büchi automaton using the Trio2Promela tool[28], or encoded as a SAT problem in Zot [163]. Zot is specifically focused on TRIO satisfiability checking, without exploiting the reduction method proposed by Rozier and Vardi [169] and used in this paper. As `g-sciff`, Zot is not only able to perform satisfiability checking, but it also supports the possibility of inserting an initial partial execution trace of the system, which is then completed, if possible, in order to make it compliant with the formulae of the model. As stated in [163], it is in general slower than NuSMV.

Zot exploits SAT-based technologies to perform bounded satisfiability checking of TRIO specifications. SAT-based technologies have been introduced to overcome the state-explosion problem of classical bounded and unbounded model checking. One represents with boolean formulas the initial state of the system $I(Y_0)$, the transition relation between two consecutive states $T(Y_i, Y_{i+1})$, and the (denied safety) property $F(Y_i)$. Then, the property is verified in the set of states $0 \dots k$ iff the formula [132]

$$I(Y_0) \wedge \left(\bigwedge_{i=0}^{k} T(Y_i, Y_{i+1}) \right) \wedge \left(\bigvee_{i=0}^{k} F(Y_i) \right)$$

is unsatisfiable. Bounded model checking is obviously not complete, in that a fixed bound is imposed. Furthermore, in a metric setting the bound represents a maximum time, and thus, differently from CLP, verification of temporal constraints is affected by the time granularity (e.g., verifying a deadline of 120 time units is more difficult than verifying a deadline of 12 time units).

SAT-based unbounded model checking is based on analogous formulae, but it also adds formulae that verify loop freeness (as in induction-based unbounded model checking [178]) or use SAT specific features

(as in interpolant-based unbounded model checking [138]). In all cases, the transition function should be unfolded for a set of possible states, which makes the boolean formula quite large. Indeed, modern SAT solvers can handle millions of boolean variables, but even generating a large SAT can be very costly.

In [76], Fisher and Dixon propose a clausal temporal resolution method to prove satisfiability of arbitrary LTL formulae. The approach is two-fold: first, the LTL formula is translated into the SNF form (which has been recalled in this manuscript – Section 5.6.1); then a resolution method, encompassing classical as well as temporal resolution rules, is applied until either no further resolvents can be generated or $\perp$ is derived. In this latter case, the formula is unsatisfiable. From a theoretical point of view, clausal temporal resolution always terminates, while avoiding the state-explosion problem; however, the translation to SNF produces large formulas, and finding suitable candidates for applying a temporal resolution step makes the resolution procedure exponential in the size of the formula. Furthermore, in case of satisfiability no example is produced.

Many approaches have been developed also in the field of LP to statically verify interaction models. For example, Alessandra Russo et al. [171] exploit abduction for the verification of declarative specifications expressed in terms of required reactions to events. They use the Event Calculus (EC) and include an explicit time structure. Global systems invariants are proven by refutation, and adopting a goal-driven approach similar to ours. The main difference concerns the underlying specification language: while Russo et al. rely on a general purpose abductive proof procedure to handle EC specifications and requirements, we adopt a language which directly captures the notion of occurred events and expectations, and whose temporal relationships are expressed as CLP constraints.

Another system aimed at proving properties of graphical specifications translated to LP formalisms is West2East [42], where interaction protocols modeled in Agent UML are translated to a Prolog program representing the corresponding finite state machine, whose properties can be verified exploiting the Prolog meta-programming facilities. However, the focus of that work is more on agent oriented software engineering, rather than verification: the system allows (conjunctions of) existential or universal queries about the exchanged messages (i.e., to check if a given message is guaranteed to be exchanged in at least one or all of the possible protocol instatiations) or guard conditions, and it is not obvious how to express and verify more complex properties.

Differently from the approach here presented, in other works LP and CLP have been exploited to implement model checking techniques. Of course, since they mimic model checking, they inherit the same computational drawbacks of classical model checkers when applied for the static verification of ConDec models. For example, Delzanno and Podelski [63] propose to translate a procedural system specification into a CLP program. Safety and liveness properties, expressed in Computation Tree Logic, are checked by composing them with the trans-

lated program, and by calculating the least and the greatest fix-point sets. In [94], Gupta and Pontelli model the observed system through an automaton, and convert it into CLP. As in our approach, they cannot handle infinite sequences without the intervention of the user: the user must provide a predicate that generates a finite number of event sequences, representing all the possible finite evolutions of the system.

12.1.2 *A-priori Compliance Verification*

A vast literature is focused on the a-priori compliance verification of interaction models, with particular attention to the BPM field. Indeed, after the outbreak of high-profile financial scandals, new legislation such as the Sarbanes-Oxley Act has been produced to regulate the behaviour of companies, and it has therefore become fundamental to statically verify if a BP design effectively complies with such regulations.

The specification of BPs by means of declarative approaches such as ConDec is a very recent topic; they are usually modeled by means of graphical procedural specifications such as BPMN [203]. Business rules, regulations and policies have instead an inherent declarative nature. As a consequence, most of the literature on this topic adopts procedural specifications to represent BPs, and declarative (usually logic-based) languages to capture the regulatory models against which BPs must be verified.

Two major mainstream approaches emerge:

- logic-based approaches which formalize regulatory models by relying on the deontic notions of obligations and permissions; verification is then carried out by exploiting resolution techniques underlying the logic.
- approaches which adopt model checking techniques for verification; the procedural specification of the BP is formalized as a Kripke structure, while business rules are captured as temporal logics formulae.

In [90], the authors introduce the Formal Contract Language (FCL) to formalize business contracts. FCL combines concepts from the logic of violations and normative positions based on Deontic Logic with Directed Obligations. FCL is reduced to a normal form which makes it possible to reason upon execution traces, verifying if all the obligations are satisfied. Possible execution traces are extracted from BPMN models before verification.

Another approach relying on obligation and permissions is presented in [88]. The authors introduce PENELOPE as a declarative language to capture obligations and permissions imposed by business policies (sequencing and timing constraints between activities). Instead of using these policies to verify an external process model, the policies itself are then employed to automatically generate BPs that are, by construction, compliant with them.

Although CLIMB does not directly rely on the deontic notions of obligations and permission, a relationship between these concepts and the ones of positive and negative expectations has been established in [5]. The suitability of SCIFF as a framework to represent and verify business contracts has been instead discussed in [8].

In [14], Awad et al. adapt the BPMN-Q graphical notation (based on the BPMN elements), to express business rules and queries, and exploit BPMN as a modeling notation for BPs. The process model is then translated to a Petri Net via a multi-step methodology able to isolate its relevant sub-parts, while the BPMN-Q query is expressed as a past-LTL formula. Model checking is then employed to verify if the formula is satisfied by the Petri Net.

A similar approach is proposed in [78], where UML Activity Diagrams are used to specify the BP, and a graphical language inspired by Activity Diagrams, called PPSL, is used to represent the regulatory business rules. Model checking is exploited by mapping PPSL onto past-LTL formulae.

It is worth noting that both BPMN-Q and PPSL support constructs very similar to a sub-set of ConDec, such as for example response, precedence and existence/absence constraints.

In [121], Liu et al. present a static compliance-checking framework in which BPEL[12] specifications are model checked against rules specified with the BPSL notation, a graphical language able to express orderings among activities possibly associated to metric constraints. BPEL specifications are then mapped to Pi-calculus, which is in turn translated to a finite state machine. BPSL requirements are instead mapped to LTL; temporal constraints are translated by nesting the $\bigcirc$ operator. For example, a BPSL requirement stating that "if a request is received, then an answer must be provided exactly after 3 time units" is formalized by means of the following LTL formula:

$$\Box(request \Rightarrow \bigcirc\bigcirc\bigcirc answer)$$

A slightly different approach is the one of [85], in which a methodology to check compliance of business processes and the resolution of violations is presented. Semantic Process Networks (SPN) are introduced to formalize BPs specified with the BPMN notation, augmented with annotations used to associate effect predicates to activities. Business rules are then verified against the produced network.

Verification of procedural specifications w.r.t. a ConDec regulatory model could be seamlessly carried out with model checking techniques, by exploiting the mapping of ConDec to LTL. With `g-sciff`, the problem is related to the specification of the procedural BP, which cannot be easily captured by means of CLIMB rules. In this respect, two possibilities could be exploited:

- the BP model is maintained separated from CLIMB. Simulations are then performed on the model to produce different possible execution traces (this assumption is made in many other works, such as for example [90]); then, the simulated execution traces

are subject to compliance verification with CLIMB rules, by adopting the `sciff` proof procedure. Such a verification is exactly the same that is carried out a-posteriori, i.e., by considering concrete actual executions of the BP. The discussion on this topic is therefore postponed to Chapter 15.

- the BP is translated to CLIMB, and the obtained specification is then combined with the CLIMB specification formalizing the ConDec regulatory model. A simple structured BP notation used to specify clinical guidelines has been translated onto CLIMB rules in [143]; however, when the focus is on static verification, to guarantee termination of `g-sciff` the BP model must must avoid the presence of loops.

12.1.3 *Model Composition*

The issue of composing local models to obtain a unique global model has been subject to extensive research in the last years, especially in the fields of Component-Based Engineering, Business Process Management (BPM) and Service Oriented Computing (SOC).

In [147], Moschoyiannis and Shields provide a set-theoretic mathematical framework for modeling components and their composition. In particular, each component is characterized by a set of behavioural constraints which associate a sequence of operation calls to each interface exposed by the component itself. Based on this notion, a formal definition of composition is provided, examining its effect on the individual components. A well-behaved composition is defined as a composition which preserves the sequencing of operation calls supported by each component. The proposed framework can be used for guiding the composition of components as it advocates formal reasoning about the composite before the actual composition takes place.

In [89], a framework for component-based composition encompassing heterogeneous interaction and execution is described. It adopts an abstract layered model of components, where each component is described by its dynamic behaviour, the architectural constraints on that behaviour, and its concrete execution model. Such descriptions are captured by using timed automata with dynamic priorities, which are then composed by means of a commutative and associative composition operator which preserves deadlock-freedom.

Petri nets are used for design-time conformance and compatibility in [133, 134, 136, 175]. For example, [134] focuses on the problem of consistency between executable and abstract processes while [136] presents an approach where for a given composite service the required other services are generated. Also related is [79], were Message Sequence Charts (MSCs) are compiled into the "Finite State Process" notation to describe and reason about web service compositions. Automatic service composition has been addressed in OWL-S [135] which looks how atomic services interact with the real world, the Roman model approach [26] that uses finite state machines, and Mealy ma-

chine [38] that focuses on message exchange between services. Compatibility of synchronous communication via message exchange in web services is investigated in [30, 27, 24, 162], while ConDec allows asynchronous communication and focuses on the process perspective, rather than message exchange. ConDec contributes to this area are with the verification techniques aimed at checking if a composition of local models meets certain properties, such as conflict-freedom and absence of dead activities. However, while the cited approaches focus on automatic composition of services (i.e., automatic choreography generation from participating services), ConDec assumes that all relevant process models of the composition are available and then verifies their interoperability.

To the best of our knowledge, the work presented in [146], is the first attempt to automatically verify declarative service models. Such models are graphically described by means of the DecSerFlow notation [186], which has been developed by the same authors of ConDec and presents many similarities with it. Verification of service composition is carried out by exploiting the possibility of mapping DecSerFlow onto LTL as well as onto the SCIFF framework, in the same way that has been described in this dissertation.

12.1.4 *Interoperability and Choreography Conformance*

As in the case of model composition, the notions of conformance and interoperability have been, and still are, deeply investigated, especially in the Service Oriented Computing setting. In the research literature it is possible to find several deÞnitions of *interoperability* and *conformance*, and there is not a complete agreement about its exactly meaning. For example, in [17, 127] the authors state that choreography conformance aims to check if a service, described by its behavioural interface, can play a given role within a choreography. The proposed conformance test is less restrictive than classic bi-simulation techniques [41]. Differently from the notion of conformance presented in Chapter 8, Baldoni et al. guarantee that if a service is evaluated as conformant with the choreography, then it will be able to interoperate with any other service conformant to the choreography. Verification is therefore carried out on each specific service independently from the other concrete services. The advantage of this approach is that any single service can be replaced by another conformant service playing the same role, without needing to touch the other concrete services; the drawback is that this notion of conformance is sometimes too strong: it rules out many possible allowed compositions whose services cover only a part of the whole choreography. The approach proposed in this dissertation embraces the opposite philosophy: verification is carried out on a specific service composition as a whole, checking if the composed services can fruitfully interact while respecting the choreography constraints. It is worth noting that, in [2], the SCIFF framework has been successfully adapted to address the type of interoperability and conformance veri-

fication described in [17]. An interesting research activity would be to study whether this adaptation could fit with the ConDec setting.

A different notion of interoperability is given in [53], where the authors represent global choreographies and local services in terms of state transition systems (and their composition as the product of the two transition systems). They define a notion of interoperability as a set of features that the resulting transition system should guarantee. Although their idea of interoperability is in some sense "broader" that the one given in [17, 2], it is still related to the procedural aspects of the interaction.

Differently from all these approaches, the notion of conformance and interoperability provided in this dissertation is instead more related to ensuring that declarative constraints specified in terms of ConDec are indeed satisfied, given the ConDec representation of both a global choreography and of a composition of local models. ConDec, in fact, focuses on the declarative aspects and features of global choreographies, leaving the interaction unconstrained as much as possible. The introduction of more restrictive forms of conformance and interoperability would clash with the openness philosophy of ConDec.

The composition of local models and their conformance with a choreography, as described in this dissertation, is tightly related to the problem of service contracting. In Semantic Web technologies, searching for a service means to identify components that can potentially satisfy the user needs in terms of outputs and effects (discovery), and that, when invoked by the customer, can fruitfully interact with her (contracting). Differently from the composition problem described in this dissertation, contracting could involve a negotiation phase, in which two parties partially disclose and negotiate their interaction policies, private information and mutual requirements.

In this setting, Roman and Kifer have proposed a framework which relies on Concurrent Transaction Logic (CTR) to model and reason about (semantic) web services [167]. Local services, as well as the global choreography, are represented by a set of CTR declarative formulae, which are able to express all the DecSerFlow/ConDec constraints. Then, a proof theory is given to solve a twofold problem:

CONTRACTING Verification is carried out to verify that at least one execution exists, s.t. the policies of a service and a client as well as the choreography constraints are guaranteed.

ENACTMENT A variation of contracting, in which the aim is to build a constructive proof for such an execution. The obtained proof can be seen as a possible way to enact the interaction while respecting the constraints of all the involved parties.

12.2 SUMMARY OF THE PART

In this part of the dissertation, we have faced the problem of static verification of open declarative interaction models, specified using the ConDec notation.

We have introduced many interesting kind of verifications that can be applied during the design phase of a ConDec model, ensuring its correctness/consistency and verifying its compliance to regulations and policies, again expressed in ConDec. We have also introduced the concepts of local and global ConDec models, discussing how compositions of local models as well as conformance of a composition w.r.t. a global model can be verified. All these verifications have been reduced to ∃- and ∀-entailment of properties, which respectively state that a ConDec (possibly composite) model meets a given property in at least one/all its supported executions.

We have recalled the `sciff` and `g-sciff` proof procedures, which can be employed for the verification of CLIMB specifications. `sciff` is devoted to checking whether a given execution trace actually complies with a CLIMB model, while `g-sciff` is a generative extension of `sciff` able to generate intensional execution traces compliant with a CLIMB model, exploiting its constraints.

We have then shown how `g-sciff` is able to effectively deal with ∃- and ∀-entailment of properties in the ConDec setting. We have provided a discussion relating the non-termination issued of `g-sciff` with the presence of loops in the ConDec model under study. This issue has been overcome with a loop detection procedure used to pre-process ConDec models taking suitable countermeasures in the presence of loops.

The effectiveness of the approach has been quantitatively assessed by means of different benchmarks, emphasizing the performance results of `g-sciff` in both favorable and unfavorable cases. Timings and scalability of `g-sciff` have been compared with the ones of model-checkers, after having shown how the ∃- and ∀-entailment of properties can be also encoded as a model checking problem.

Part III

RUN-TIME AND A-POSTERIORI VERIFICATION

13

RUN-TIME VERIFICATION

"Begin at the beginning",
the King said, very gravely,
"and go on till you come to the end: then stop"

— Lewis Carroll

Contents

13.1 The Run-Time Verification Task **234**
13.2 Run-time Verification with the SCIFF Proof Procedure **235**
13.2.1 Reactive Behaviour of the SCIFF Proof Procedure 235
13.2.2 Open Derivations 236
13.2.3 Semi-Open Reasoning 238
13.3 The SOCS-SI Tool **240**
13.4 Speculative Run-Time Verification **241**
13.4.1 Speculative Verification with the `g-sciff` Proof Procedure 242
13.4.2 Interleaving the `sciff` and `g-sciff` Proof Procedures 243

Static verification techniques are useful to assess the correctness, safety and consistency of the designed models. However, they suppose that the model is completely accessible, i.e., that it is a *white box*.

There are many situations in which such an assumption is not reasonable. For example, a ConDec choreography could be applied to an already existing service composition. Here, the internal implementation of concrete services taking part to the composition is unaccessible, i.e., they are *black boxes*. Also the case in which a service composition is built to realize a desired choreography cannot relies only on static verification to ensure that the composition correctly deals with all the choreographic constraintd. In fact, the composition is built by looking at the behavioural interface of the selected parties, without having access to their complete internal implementation (they are *gray boxes*). In this respect, there is no guarantee that the behaviour exposed by a third-party service really corresponds to its internal implementation. Potential mismatches between the exposed and the implemented behaviours could produce unexpected/undesired interactions, breaking the choreographic global contract.

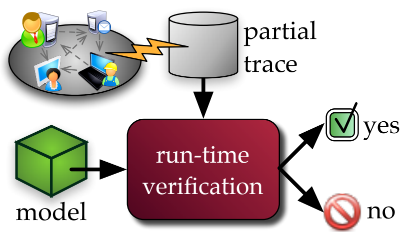


Figure 53: ConDec and run-time verification.

In all these cases, verification must shift from static-time to run-time, where the events generated by the (concrete implementation of) interacting entities can be collected an analyzed, checking if they actually adhere to the global choreographic model.

In other words, static verification must be complemented by *run-time verification*.

13.1 THE RUN-TIME VERIFICATION TASK

As defined in [59]:

> Run-time verification techniques are used to dynamically verify the behavior observed in the target system with respect to given requirements. Verification is used both to check that the observed behavior matches specified properties and to explicitly recognize undesirable behaviors in the target system.

In the ConDec setting, run-time verification aims at checking whether the current execution trace complies with all the constraints contained in the model.

During the execution of an instance of the system, the interacting entities generate events which are collected in a corresponding execution trace. Hence, run-time verification is carried out on a partial execution trace, represented by the course of interaction reached so far (see Figure 53).

Reasoning with incomplete information

A run-time verifier must be able to deal with incomplete information: at each time, the observed execution trace is only a partial version of the whole trace describing the instance, and therefore verification cannot provide a definitive answer. For example, if the model prescribes that a certain event e must eventually occur within the instance, and at the current time such an event has not yet been generated by the interacting entities, the verifier should not infer that the execution is wrong, but only that the requirement has not yet been satisfied so far.

On-the-fly detection of violations

On the other hand, a run-time verifier must be able to detect on-the-fly whether the behaviour of the interacting entities is respecting the prescriptions of the model, detecting violations as soon as possible.

Open, semi-open and closed reasoning

In the following, we will identify three kind of reasoning capabilities

needed for run-time verification:

OPEN REASONING to deal with partial execution traces; when the interaction is ongoing, new events will occur, and therefore reasoning must be "open" w.r.t. the acquisition of new knowledge.

CLOSED REASONING to conclude the reasoning process when the instance reaches an end, i.e., when the current execution trace is actually a complete execution trace, providing an exhaustive knowledge about the evolution of the instance.

SEMI-OPEN REASONING to properly combine the two kind of reasoning when certain assumptions can be made on the system. As we will see, a typical example is the one in which events always occur in ascending order: reasoning is, in this case, closed on the past and open w.r.t. the future.

13.2 RUN-TIME VERIFICATION WITH THE SCIFF PROOF PROCEDURE

State of the art approaches face the run-time verification problem by proposing ad-hoc solutions, which rely on specific software implementations or customized decision procedures, lacking a strong formal basis. This makes it impossible to prove formal properties about the verifiers, such as soundness and completeness. However, this properties are of key importance; only if they are met, the correctness of the results produced by the verifier is guaranteed.

As far as we are concerned, the `sciff` proof procedure [7] is one of the few methods which deal with the run-time verification task by relying on a strong formal basis. As we have pointed out in Section 9.2, `sciff` is sound and complete, and meets termination for any CLIMB specification, i.e., it will always provide an answer in finite time. Differently from many approaches found in the literature, the SCIFF framework has been originally thought for specifically targeting the run-time verification problem, and has been later extended to deal with static verification. The run-time verification facilities of `sciff` have been applied to many different systems, including Multi-Agent Systems [7], business contracts [8], web service choreographies [3] and clinical guidelines [143].

13.2.1 *Reactive Behaviour of the SCIFF Proof Procedure*

As described in Section 9.1, `sciff` checks wether an execution trace complies with a given CLIMB specification. This task can be seamlessly carried out on a complete trace, or by dynamically reasoning on a partial trace. In fact, the transition system used to implement `sciff` (see Section 9.1) is specifically thought for dealing with the dynamic occurrence of events:

Open vs closed `sciff` *reasoning*

A. `sciff` supports both an "open" and a "closed" reasoning modal-

ity, reflecting whether the current trace is partial or complete. By default, `sciff` works in an open modality: when a positive expectation is not fulfilled by any event occurred in the (current) trace, it is stored in the set of "pending" expectations, waiting for a further occurrence able to fulfill it. When the course of interaction reaches its end, then `sciff` is alerted, by setting a flag, that no more events will occur; `sciff` then applies the *closure* transition, which in turn makes it possible to evaluate the expectations which are still pending[1].

B. `sciff` has a transition, called *happening*, specifically dedicated to acquire a new event occurrence from an external queue.

Reactivity of SCIFF

In other words, `sciff` exhibits a *reactive behaviour*. As soon as a happened event, say, h, is inserted in the external queue of `sciff`, then it is imported by means of the *happening* transition. This starts a new reasoning phase in which `sciff` (i) checks whether h fulfills a pending positive expectation or violates a pending negative expectation, and (ii) combines h with the partially solved ICs of the model, generating new expectations if h has the ability of triggering some of them. By supposing that `sciff` is working in open modality, two possibilities may then arise:

A. `sciff` reaches a failure node; it then performs backtracking, trying a different derivation[2]. If all the possible branches of the proof tree lead to a failure node, then `sciff` states that the current partial trace violates the specification.

B. `sciff` reaches a non-failure node in which no further transition is applicable; in this case, we say that the proof procedure reaches *quiescence*.

13.2.2 *Open Derivations*

The fact that `sciff` reaches a non-failure node, where no further transition is applicable, attests that the partial execution trace analyzed so far does not violate the constraints prescribed by the model. In this case, we say that `sciff` has a successful open derivation for the partial trace w.r.t. the model.

Definition 13.1 (`sciff` open successful derivation). Given a CLIMB specification $\mathcal{S}$, an initial execution trace $\mathcal{T}_i$ and a current partial execution trace $\mathcal{T}_p \supseteq \mathcal{T}_i$, there exists an *open successful derivation* for $\mathcal{S}_{\mathcal{T}_p}$ starting from $\mathcal{T}_i$ iff the proof tree with root node the initial node $I(\mathcal{S}_{\mathcal{T}_i})$ has at least one non-failure node containing $\mathcal{T}_p$. In this case, we write $\mathcal{S}_{\mathcal{T}_i} \vdash^{\mathcal{T}_p}_{\Delta} \mathtt{true}$, where Δ is the abductive explanation computed in that non-failure node.

1 Each pending positive expectation is declared as violated; each pending negative expectation is declared as fulfilled.

2 Remember that `sciff` adopts a depth-first search strategy

Let us now suppose that `sciff` has an open successful derivation for $\mathcal{S}_{\mathcal{T}_p}$. This means that, after having processed the last happened event of $\mathcal{T}_p$, `sciff` has reached a node in which no more transition is applicable. Here, two further possibilities may arise:

- The current instance reaches its termination, i.e., $\mathcal{T}_p$ is actually a complete execution trace. The "closure flag" is set on `sciff`, which can then evaluate the expectations which are still pending; in particular, if at least one positive expectation is still pending, then a failure node is generated, and the open successful derivation leads, in fact, to a closed failure derivation. Contrariwise, if no positive expectation is pending at the closure time, then a success node is reached.
- A new event occurs and is inserted into the external queue of `sciff`. `sciff` imports it through the *happening* transition, which in turn triggers a new reasoning phase.

We illustrate how `sciff` works on two simple examples. The examples concern different execution instances which must obey to the following ConDec diagram:

query ●═▶ inform

Such a model can benefit from the run-time verification facilities of `sciff`, thanks to the ConDec-CLIMB mapping. In particular, the application of the (revised) translation function described in Section 10.4.1 produces a CLIMB specification $\mathcal{QI} = \langle \emptyset, \{(\dagger)\} \rangle$, where (†) is:

$$\begin{aligned} \mathbf{H}(exec(query), T_q) \rightarrow & \mathbf{E}(exec(inform), T_i) \wedge T_i > T_q \\ & \wedge \mathbf{EN}(exec(query), T_{q2}) \\ & \wedge T_{q2} > T_q \wedge T_{q2} < T_i. \end{aligned} \qquad (\dagger)$$

EXAMPLE 13.1 (Violation at closure). *Let us suppose that* `sciff` *is used to verify an execution instance against* $\mathcal{QI}$*. At the beginning of the computation, the execution trace is empty.* (†) *does not trigger, and therefore* `sciff` *reaches immediately the quiescence, waiting for an event occurrence.*

Let us now suppose that a query is sent at time 5*, i.e., that the new partial execution trace is* $\mathcal{T}_1 = \{\mathbf{H}(exec(query), 5)\}$*. The happening of the* query *activity triggers* (†)*, leading to generate a positive expectation about a consequent* inform*, and a negative expectation about a further* query *inbetween. Then, a new quiescence node* N_1 *is reached, and therefore* `sciff` *has a successful open derivation for* $\mathcal{T}_1$*:* $\mathcal{S}_{\emptyset} \vdash_{\Delta_1}^{\mathcal{T}_1} true$*, where*

$$\begin{aligned} \Delta_1 = \{ & \mathbf{E}(exec(inform), T'_i) \wedge T'_i > 5, \\ & \mathbf{EN}(exec(query), T'_{q2}) \wedge T'_{q2} > 5 \wedge T'_{q2} < T'_i \} \end{aligned}$$

The two generated expectations are put in the pending set, and `sciff` *waits for a new event occurrence.*

Now `sciff` *is notified that no further event will happen in the current instance (i.e., that* $\mathcal{T}_1$ *is a complete trace). The* closure *transition is consequently applied, leading to a failure node, which states that the positive expectation about the execution of* inform *is violated. In other words,* $\mathcal{S}_\emptyset \not\vdash_{\Delta_1}^{\mathcal{T}_1} \mathtt{true}$*, and thus* $\mathcal{T}_1$ *is not compliant with* $\mathcal{S}$*.*

Example 13.2 (Successful verification). *As in Example 13.1, a query is sent at time* 5. `sciff` *reaches quiescence in a node whose abductive explanation is the set* Δ_1 *shown in Example 13.1.*

Then, a new event occurs within the instance, attesting that the inform *activity is executed at time* 10*. Such an activity is able to fulfill the positive pending expectation, and therefore* `sciff` *generates two successor nodes (*$N_{2,1}$ *and* $N_{2,2}$*), one in which fulfillment is applied, one in which fulfillment is avoided. Then,* `sciff` *selects the first node. Such a node is characterized by the following elements:*

$$\begin{aligned}
\mathcal{T}_{2,1} &= \{\mathbf{H}(\mathtt{exec}\,(\mathtt{query})\,,5), \mathbf{H}(\mathtt{exec}\,(\mathtt{inform})\,,10)\} \\
\Delta_{F2,1} &= \{\mathbf{E}(\mathtt{exec}\,(\mathtt{inform})\,,10)\} \\
\Delta_{P2,1} &= \{\mathbf{EN}(\mathtt{exec}\,(\mathtt{query})\,,T'_{q2}) \wedge T'_{q2} > 5 \wedge T'_{q2} < 10\}
\end{aligned}$$

Now `sciff` *is notified that no further event will happen in the current instance (i.e., that* $\mathcal{T}_{2,1}$ *is a complete trace).* `sciff` *applies* closure*, followed by* EN-fulfillment*: since no further event will happen, the negative expectation concerning the query is satisfied.* `sciff` *finally reaches a success node, attesting that* $\mathcal{T}_{2,1}$ *is compliant with* $\mathcal{QI}$*.*

13.2.3 *Semi-Open Reasoning*

In its basic form, `sciff` does not make any assumption about the orderings in which events occur and are fetched. This is a desired feature, because in a distributed system, there is no guarantee that events generated by two different parties are delivered to `sciff` in the same order as they occurred. In other words, during the execution `sciff` is open w.r.t. both the past and the future.

Ordering of event occurrences

However, there exist also many systems in which such an ordering is preserved (especially when the time granularity is coarse-grained). If it is the case, `sciff` can seamlessly suppose that event occurrences are fetched in ascending order: when a new event occurring at time t is fetched, then it is guaranteed that all the event fetched in the future will involve times greater than t.

Such an information can be exploited to "close" the reasoning w.r.t. the past: since happened events will not happen at a time lower than t, all the time variables associated to pending positive expectations can be updated, stating that they must be greater or equal than t. This behavior is encapsulated in the *times update* transition, which has been described in Section 9.1.2 - Page 159.

By taking into account the following definition of *current time*, if the current execution trace is $\mathcal{T}_p$, the transition has the effect of adding a CLP constraint on each positive pending expectation, imposing that its corresponding time variable must be greater or equal than $\mathtt{ct}(\mathcal{T}_p)$.

DEFINITION 13.2 (Current time). Given an execution trace $\mathcal{T}$, the *current time* associated to $\mathcal{T}$ is:

$$ct(\mathcal{T}) \triangleq \max\{t \mid \mathbf{H}(e, t) \in \mathcal{T}\}$$

Thanks to the *times update* transition, `sciff` becomes "eager" to evaluate expectations. Such an eager evaluation is exploited to detect violations of ConDec constraints as soon as possible.

EXAMPLE 13.3 (Eager evaluation of the precedence constraint). *Let us consider a ConDec model involving a precedence constraint*

pay ——►● delivery

which is formalized in CLIMB *as:*

$$\mathbf{H}(exec\,(delivery), T_d) \rightarrow \mathbf{E}(exec\,(pay), T_p) \wedge T_p < T_d.$$

Let us now suppose that the execution of activity delivery *at time* 8 *is dynamically fetched by* `sciff`*. Such happened event matches with the antecedent of the* IC *formalizing the precedence constraint, and thus the* IC *is triggered, generating the expectation* $\mathbf{E}(exec\,(pay), T'_p) \wedge T'_p < 8$.

After the generation of this expectation, the default behaviour of `sciff` *would be to wait for the occurrence of a next event, or for the notification stating that the execution has reached an end. If, instead, the* times update *transition is enabled,* `sciff` *will infer that* T'_p *must be greater or equal than* 8*. The insertion of this* CLP *constraint into the constraint store immediately leads to a failure, due to the inconsistency of* $T'_p < 8 \wedge T'_p \geqslant 8$.

EXAMPLE 13.4 (Eager evaluation of deadline expiration). *Let us consider a ConDec^{++} model involving a metric response constraint*

pay ●—(−,48)—► delivery

which is formalized in CLIMB *as:*

$$\mathbf{H}(exec\,(pay), T_p) \rightarrow \mathbf{E}(exec\,(delivery), T_d) \wedge T_d > T_p \wedge T_d < T_p + 48.$$

Let us now suppose that the execution of activity pay *at time* 21 *is dynamically fetched by* `sciff`*. Such happened event matches with the antecedent of the* IC *formalizing the metric response constraint, and thus the* IC *is triggered, generating the expectation* $\mathbf{E}(exec\,(delivery), T'_d) \wedge T'_d > 21 \wedge T'_d < 69$.

If the times update *transition is enabled, then the deadline expiration can be detected by* `sciff` *as soon as the first event occurring after time* 69 *is fetched. Indeed, let us suppose that the deadline expires, and that a new activity* req info *is performed at time* 72*.* `sciff` *applies the* times update *transition, imposing* $T'_d \geqslant 72$*, which is however in conflict with* $T'_d < 69$ *and thus leads* `sciff` *to a failure.*

Semi-open derivation

In conclusion, the *times update* transition implements a sort of Closed World Assumption on the past: given a trace $\mathcal{T}$, it states that no further

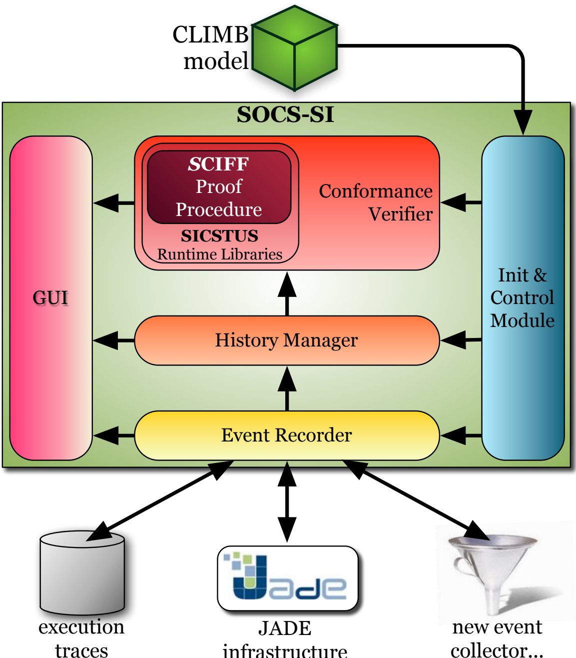


Figure 54: SOCS-SI architecture.

event can happen at a time lower than $\mathtt{ct}(\mathcal{T})$ – the execution trace is complete until the current time – but admits future event occurrences, i.e., events occurring at a time greater or equal than $\mathtt{ct}(\mathcal{T})$. In this respect, sciff performs *semi-open* derivations, i.e., derivations which are closed on the past and open w.r.t. the future. We identify semi-open derivations with notation $\vDash$.

13.3 THE SOCS-SI TOOL

SOCS-SI [4] is a tool fur run-time compliance verification of interacting entities. The tool is composed by the sciff proof procedure, interfaced to a graphical user interface and to a component for the observation of the interaction. SOCS-SI has a twofold purpose:

- providing the possibility of intercepting event occurrences, playing the role of external queue for sciff;
- returning to the user the results computed by sciff, showing the evolution of pending, fulfilled and violated expectations.

As shown in Figure 54, the SOCS-SI software application is composed by a set of modules[3]:

EVENT RECORDER fetches events from different sources and stores them inside the history manager.

3 All components, except the proof procedure, are implemented in the Java language.

HISTORY MANAGER receives events from the event recorder and composes them into an external queue for `sciff`.

SOCIAL COMPLIANCE VERIFIER wraps the `sciff` proof procedure, fetching events from the history manager. As soon as `sciff` is ready to process a new event, it imports one from the history manager. The event is processed and the results of the computation are returned to the GUI. The proof procedure then continues its computation by fetching another event if there is any available. If not, it suspends, waiting for new events, or for a communication that no more events will arrive.

INIT&CONTROL MODULE provides for initialization of all the components in the proper order. It receives as initial input a set of specifications defined by the user. Therefore, thanks to the translation of ConDec to CLIMB, SOCS-SI can be adopted for the run-time verification of interacting entities w.r.t. a ConDec model.

The event recorder is the core of the tool. It fetches events from an external system through specialized modules. Each module is dedicated to fetch events from a specific event source. The tool can be easily extended by inserting as many specialized modules as desired: to insert a new module, the user must implement it by extending a `RecorderInterface`, which exposes the basic methods for fetching events. The new module can then be selected through the GUI.

13.4 SPECULATIVE RUN-TIME VERIFICATION

We now discuss a problem related to the run-time verification of ConDec models, showing how a proper integration between `sciff` and `g-sciff` can be successfully exploited to overcome it.

Although `sciff` is able to reason upon a dynamically incoming execution trace, in a reactive manner, it is not always able to detect a violation as soon as possible. To point out such an issue, we use the ConDec model shown in Figure 55 as a running example. The model illustrates an order management process consisting of a sequence of four steps:

A. first of all, an order is selected from the list of pending orders;

B. the price of the order is calculated;

C. a package for the order's shipment is prepared;

D. the order is finally delivered to the customer.

The process contains a further activity, modeling the exceptional situation in which the stock is empty. In this case, no delivery can be executed afterwards. The relation between the empty stock and the deliver order activities is therefore modeled as a negation response constraint.

Now let us consider the following (partial) execution: the warehouse realizes that the stock is empty at time 1, and then the seller selects an

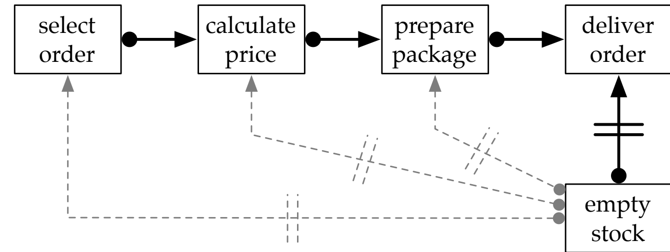


Figure 55: An order management ConDec model with hidden dependencies.

order at time 5. At time 5, the execution should be evaluated as non-compliant: the seller has selected an order, and therefore such order must be eventually delivered; but this contrasts with the fact that being the stock empty, no delivery can happen. We have identified this problem as "presence of hidden dependencies" in ConDec models (see Section 3.5).

Delayed violation detection

`sciff` is not able to detect non-compliance at time 5, because the inconsistency point has not yet been reached. However, such an inconsistency point will surely appear *in the future*, either because one of the remaining three steps will not be executed, or because the execution eventually comes to the deliver order activity, which is however forbidden.

13.4.1 *Speculative Verification with the* `g-sciff` *Proof Procedure*

In order to make `sciff` able to deal with this issue, it must be extended with a *speculative* part, able to carry out hypothetical reasoning forward in the future, checking if at least one possible extension of the current trace exists, s.t. all the constraints of the model are respected. If it is not the case, then a violation can be identified in advance. In our running example, speculative reasoning could complement `sciff`, providing a negative answer, at time 5, to the question: is it possible to go on by respecting the ConDec model?

We argue that this speculative form of reasoning resembles very strictly the conflict-freedom checking issue, introduced in Definition 8.3 – Page 137. In its original formulation, conflict-freedom checking aims at statically evaluating whether a ConDec supports at least one possible execution. In this context, instead, the question is whether the ConDec model supports at least one possible execution containing the events occurred so far.

Speculative reasoning as checking conflict-freedom

In Theorem 10.1 – Page 177, we have shown that the conflict-freedom problem can be suitably tackled by `g-sciff`: a ConDec model $\mathcal{CM}$ is conflict-free iff $t_{\text{CLIMB}}(\mathcal{CM})_{\emptyset} \vdash_{g\Delta}^{\mathcal{T}} \mathtt{true}$. However, it is worth noting that `g-sciff` seamlessly supports the possibility of starting from an initial execution trace: in this case, `g-sciff` tries to extend such a trace so as to make it compliant with the ConDec model. Therefore, `g-sciff` can be effectively employed to accomplish speculative reasoning.

```
function: SpeculativeRunTimeVerification(ConDec model CM)
returns : false, if a violation is detected during the verification, true
          otherwise
1  begin
2      enable the times update transition in g-sciff;
3      [CMΨ_Aug,Switch] ← PreProcess(CM);
4      S ← t_CLIMB(CMΨ_Aug);
5      if ¬Switch then
6          Bound ← -1;
7      else
8          Bound ← ask a bound to the user;
9      end
10     T_cur ← ∅;
11     while Open do
12         T_old ← T_cur;
13         Occurrence ← extract an event occurrence from the queue;
14         T_cur ← T_old ∪ {Occurrence};
15         if S_{T_old} ⊬_Δ^{T_cur} true then
16             return false;
17         end
18         [OK_future,_] ← CallGSCIFF(S, T_cur, Bound );
19         if ¬OK_future then
20             return false;
21         end
22     end
23     if S_{T_old} ⊢_Δ^{T_cur} true then
24         return true;
25     else
26         return false;
27     end
28 end
```

Function SpeculativeRunTimeVerification($\mathcal{CM}$)

13.4.2 *Interleaving the* sciff *and* g-sciff *Proof Procedures*

Speculative run-time verification can then be suitably tackled by interleaving the application of sciff and g-sciff:

A. first, a sciff open derivation is used to check if the execution trace collected so far does not violate the constraints of the model;

B. then, g-sciff is employed to check whether there exists at least one course of interaction, extending the current one, which complies with the model.

The algorithm enclosed into the SpeculativeRunTimeVerification($\mathcal{CM}$) function relies on this interleaving for performing speculative run-time verification. It supposes that events are guaranteed to be fetched in ascending order, but such a hypotheses is not a mandatory requirement for the algorithm. The verification procedure is organized as follows:

A. first of all, the *times update* transition is enabled in `g-sciff` – in this way, `g-sciff` generates happened events only in the future, i.e., after the current time;

B. then, the ConDec model given as input is pre-processed in order to guarantee the termination of `g-sciff`[4];

C. while the interaction is open (i.e., accepts new event occurrences)
   a) a new event occurrence is extracted from the external queue (if the queue is empty, then the operation waits until an event is inserted in the queue);
   b) `sciff` is used to evaluate whether the trace, extended with the last fetched event, is violating the ConDec model;
   c) if there does not exist a semi-open successful derivation for `sciff`, then a violation is detected, and the procedure terminates;
   d) otherwise, `g-sciff` is used to accomplish speculative reasoning[5];
   e) if the speculative reasoning task returns a negative answer, then a violation is detected, and the procedure terminates.

D. when the execution has reached an end s.t. no violation has been detected, a `sciff` closed derivation is performed to evaluate still pending expectations, and computing the final verification result.

4 The pre-processing procedure has been described in Section 10.5.
5 The `g-sciff` call is encapsulated in a dedicated function, which has been described in Section 11.1.

14

MONITORING AND ENACTMENT WITH REACTIVE EVENT CALCULUS

Contents

14.1 Event Calculus **246**
 14.1.1 The Event Calculus Ontology 246
 14.1.2 Domain-Dependent vs Domain-Independent Axioms 247
 14.1.3 Reasoning with Event Calculus 248
14.2 The Reactive Event Calculus **249**
14.3 REC Illustrated: A Personnel Monitoring Facility **251**
 14.3.1 Formalizing the Personnel Monitoring Facility in REC 252
 14.3.2 Monitoring a Concrete Instance 253
 14.3.3 The Irrevocability Issue 254
14.4 Formal properties of REC **255**
 14.4.1 Irrevocability of REC 255
14.5 Monitoring Optional Constraints with REC **261**
 14.5.1 Representing ConDec Optional Constraints in REC 261
 14.5.2 Identification and Reification of Violations 264
 14.5.3 Compensating Violations 266
 14.5.4 Monitoring Example 266
14.6 Enactment of ConDec Models **269**
 14.6.1 Showing Temporarily Unsatisfied Constraints 271
 14.6.2 Blocking Unexecutable Activities 271
 14.6.3 Termination of the Execution 273

In Chapter 13, we have shown how `sciff` can be used to perform run-time verification of an execution instace w.r.t. a given ConDec model. The main drawback of the approach is that, when a violation is detected, then `sciff` generates a "no" answer, terminating the computation.

Differently from run-time verification, we identify *monitoring* as a task aimed at dynamically checking the behaviour of interacting entities, by *capturing* the detected violations without terminating its computation (see Figure 56); a captured violation could trigger a corresponding alarm, or warning the system adiminstrator.

In the ConDec setting, monitoring comes in support for dealing with optional constraints. Optional constraints express preferred scenario; in other words, it would be preferable that the interacting entities respect them, but their violation does not undermine compliance [157].

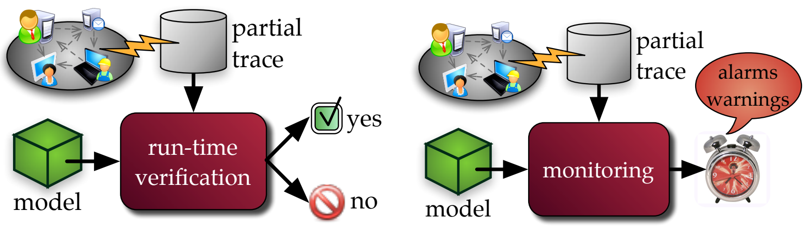


Figure 56: Run-time verification vs monitoring.

As pointed out by Pesic in [157], "allowing users to violate an optional constraint without any warning would totally hide the existence of the constraint". Monitoring optional constraints is therefore important to report warning when the behaviour of interacting entities deviates from optional constraints.

In this chapter, we show how a reactive form of the Event Calculus (EC) [113] can be encoded as a SCIFF-lite program, enabling the possibility of

- monitoring optional constraints;
- inserting *compensation* constructs in ConDec, aimed at expressing what countermeasures should be taken when an optional constraint is violated;
- tracking the evolution of constraints.

The latter point is of key importance when dealing with the enactment of ConDec models, which is discussed in the last part of the chapter.

14.1 EVENT CALCULUS

More than 20 years ago, Kowalski and Sergot [113] introduced the Event Calculus (EC) as a general framework to reason about time and events. EC overcomes limitations of other previous approaches, such as the Situation Calculus, providing a solid theoretical basis to express and reasoning upon complex requirements in Event-Based Systems (EBSs). Basic concepts are that of *event*, happening at a point in time, and *property* (or *fluent*), holding during time intervals.

EC has many interesting features. Among them: an extremely simple and compact representation, symmetry of past and future, generality with respect to time orderings, executability and direct mapping with computational logic frameworks, modelling of concurrent events, immunity from the frame problem, and explicit treatment of time and events. It has therefore been applied in a variety of domains.

14.1.1 *The Event Calculus Ontology*

As described in [177], EC "is a logical mechanism that infers what is true when given what happens when and what actions do".

$happens_at(Ev, T)$	Event Ev happens at time T
$mvi(F, T_i, T_f)$	Fluent F begins to hold from time T_i and persists to hold until time T_f: $(T_i, T_f]$ is a maximum validity interval for F
$holds_at(F, T)$	Fluent F holds at time T
$initially_holds(F)$	Fluent F holds from the initial time
$initiates(Ev, F, T)$	Event Ev initiates fluent F at time T
$terminates(Ev, F, T)$	Event Ev terminates fluent F at time T

Table 29: The EC ontology.

"What actions do" is the background knowledge about actions and their effects. This background knowledge formalizes what properties becomes true or false when actions are performed, i.e., events occur. In the EC terminology, properties/effects are called *fluents*, and the capability of an event to make a fluent true (false respectively) at a certain time is formalized by stating that the event is able to *initiate* (*terminate* resp.) the fluent.

"What happens when" is represented by the execution trace of a specific instance of the system under study. Fluents status is affected by the occurring events of the instance, according to the background knowledge. When the *happening* of an event causes a fluent to not hold (hold resp.), we say that the fluent becomes *(de)clipped*. Fluents, together with the evolution of their validity over time, describe partial states of the instance; this is the main difference between EC and Situation Calculus [137], where *situations* capture complete state of affairs.

By combining the background knowledge about actions and effects and a concrete execution trace of the system, "what is true when", i.e., the intervals inside which fluents *hold*, can be inferred. In the literature, a fluent does not hold at the time it is declipped, but it holds at the time it is clipped: fluents validity intervals are open on the left and closed on the right.

The EC ontology is shown in Table 29; differently from the classical EC ontology, this one reports the concept of Maximum Validity Interval (MVI [51]), represented by the $mvi/3$ predicate. MVIs are maximum intervals inside which fluents uninterruptedly hold. Furthermore, note that EC adopts a time structure with a minimal element. At the minimal time, the system is in its initial state, which can be characterized by describing the set of fluents holding at the beginning of the execution. This is done by means of $initially_holds$ predicates.

14.1.2 *Domain-Dependent vs Domain-Independent Axioms*

The EC formalization of a system is constituted by two parts:

- A domain-independent part, which *formalizes the meaning* of the EC ontology reported in Table 29. It contains a set of general axioms which capture the mutual relationships between the EC predicates, expressing how they impact on the status of fluents. These general axioms are valid for each domain-dependent formalization.

- A domain-dependent part, which *uses* the predicates of the EC ontology to declaratively formalize the specific system under study, describing the allowed actions and their effects, and characterizing the initial state.

There are many different formulations, in the LP setting, of the domain-independent axioms [52]. One possibility (P stands for *Property*, E for *Event*):

$$\begin{aligned}\mathtt{holds_at}(P,T) \leftarrow\ & \mathtt{initiates}(E,P,T_{Start}) \\ & \wedge T_{Start} < T \wedge \neg \mathtt{clipped}(T_{Start},P,T).\end{aligned} \quad (ec_1)$$

$$\begin{aligned}\mathtt{clipped}(T_1,P,T_3) \leftarrow\ & \mathtt{terminates}(E,P,T_2) \\ & \wedge T_1 < T_2 \wedge T_2 < T_3.\end{aligned} \quad (ec_2)$$

$$\begin{aligned}\mathtt{initiates}(E,P,T) \leftarrow\ & \mathtt{happens_at}(E,T) \wedge \mathtt{holds_at}(P_1,T) \\ & \wedge \ldots \wedge \mathtt{holds_at}(P_N,T).\end{aligned} \quad (ec_3)$$

$$\begin{aligned}\mathtt{terminates}(E,P,T) \leftarrow\ & \mathtt{happens_at}(E,T) \wedge \mathtt{holds_at}(P_1,T) \\ & \wedge \ldots \wedge \mathtt{holds_at}(P_N,T).\end{aligned} \quad (ec_4)$$

(ec_1) and (ec_2) are the general EC axioms, while (ec_3) and (ec_4) are the domain-specific axioms. Dual axioms and predicates, such as declipped, can be added to define when properties do *not* hold and to model actions with duration [177].

An example of a domain-dependent axiom is: "if someone *touches* the light, then the light becomes *broken* and it is no more *on*", where "touch" is an action and "broken" and "on" are fluents. It can be encoded in the EC ontology e.g. as:

$$\mathtt{initiates}(\mathtt{touch_light}, \mathtt{light}(\mathtt{broken})).$$
$$\mathtt{terminates}(\mathtt{touch_light}, \mathtt{light}(\mathtt{on})).$$

14.1.3 *Reasoning with Event Calculus*

By grounding the classification of Peirce (see Section 4.3.1) on the EC setting, different reasoning tasks can be carried out:

DEDUCTIVE REASONING Given the EC formalization of a system and an execution trace, in the form of a list of happens/2 predicates, it deduces the validity intervals of fluents; in this way, queries can be placed to know if a given fluent held at a certain time.

ABDUCTIVE REASONING Given the EC formalization of a system and a query, describing a desired state of affairs, it seeks an execution trace which can be exhibited by the system, leading to the desired state; the synthesized execution trace is often considered as a *plan*.

INDUCTIVE REASONING Given an execution trace and the corresponding evolution of fluents, it tries to generalize the connection between the trace and the fluents by looking for a general theory of the effects of actions, accounting for the observed data.

Cached Event Calculus

All these tasks take place after or prior to execution, but not during execution. The reason is that each time an event occurs, the EC enables a straightforward update of the theory (it suffices to add happens_at facts), but it incurs a substantial increase of the query time, since backward reasoning has to be restarted from scratch. However, runtime reasoning tasks, such as monitoring, would greatly benefit from the EC's expressive power. For this reason, some propose to cache the outcome of the inference process every time the knowledge base is updated by a new event. The Cached Event Calculus (CEC) [51] computes and stores fluents' MVIs, which are the maximum time intervals in which fluents hold, according to the known events. The set of cached validity intervals is then extended/revised as new events occur or get to be known.

The lack of logic-based reactive EC reasoners

10 years ago, following a different line of research, Kowalski and Sadri [112] proposed to use Abuctive Logic Programming (ALP) as a way to reconcile backward with forward reasoning inside an intelligent agent architecture. However, beside planning, ALP has not been used in combination with the EC. Nor are we aware of other logical frameworks that implement the EC in a reactive way. For that reason, we only find reactive EC implementations outside of logical frameworks, or else logic-based implementations of the EC that do not exhibit any reactive feature. As a consequence, large application domains such as run-time monitoring and event processing have been tackled so far by EC-inspired methods but only based on ad-hoc methods without a strong formal basis. In particular, it is very difficult to understand and prove the formal properties of current reactive EC implementations.

We now show how to overcome this limitation. Building on Kowalski et al.'s work, we equip the EC framework with the reactive features of sciff, which have been described in Chapter 13. We obtain a reactive version of the calculus, which we call Reactive Event Calculus (REC). In this reactive calculus, fluents are initiated and terminated by dynamically occurring events, thanks to the run-time capabilities offered by the sciff proof procedure.

14.2 THE REACTIVE EVENT CALCULUS

The EC can be elegantly formalized in LP, but as we said above, that would be suitable for top-down, "backward" computation, and not for run-time monitoring. For this reason, we resort to the SCIFF framework,

which reconciles backward with forward reasoning and is equipped with the sciff proof procedure, specifically thought for reactive, dynamic reasoning.

The SCIFF-lite[1] axiomatization of EC that follows draws inspiration from Chittaro and Montanari's CEC and on their idea of MVIs. The basic predicates of the calculus are presented below (Axioms (rec_1) through (rec_7)). Events and fluents are terms and times are integer CLP variables, 0 being the "initial" time.

MVIs as abducibles

REC uses the abduction mechanism to generate MVIs and define their persistence. It has a fully declarative axiomatization: no operational specifications are needed. It uses two special internal events (denoted by the reserved clip/declip words) to model that a fluent is initiated/can be terminated. The expressive power of REC is the same as the one of CEC, specifically it enables the definition of a context. A use case will be shown below.

Axiom 14.1 (Holding of fluent). *A fluent* F *holds at time* T *if a MVI containing* T *has been abduced for* F.

$$\mathrm{holds_at}(F, T) \leftarrow \mathbf{mvi}(F, [T_s, T_e]) \wedge T > T_s \wedge T \leqslant T_e. \qquad (\mathrm{rec}_1)$$

Axiom (rec_1) is a backward rule (clause), as well as Axiom (rec_7), whereas Axioms (rec_2) through (rec_6) are forward implications (SCIFF-lite ICs). Such a mixture of backward and forward inference rules is enabled by ALP [112] and it represents the backbone of REC's reactive behaviour.

Axiom 14.2 (MVI semantics). *If* $(T_s, T_e]$ *is a MVI for* F*, then* F *must be declipped at time* T_s *and clipped at time* T_e*, and no further declipping/clipping must occur in between.*

$$\begin{aligned} &\mathbf{mvi}(F, [T_s, T_e]) \\ &\quad \rightarrow \mathbf{E}(\mathrm{declip}(F), T_s) \wedge \mathbf{E}(\mathrm{clip}(F), T_e) \\ &\qquad \wedge \mathbf{EN}(\mathrm{declip}(F), T_d) \wedge T_d > T_s \wedge T_d \leqslant T_e \\ &\qquad \wedge \mathbf{EN}(\mathrm{clip}(F), T_c) \wedge T_c \geqslant T_s \wedge T_c < T_e. \end{aligned} \qquad (\mathrm{rec}_2)$$

Axiom 14.3 (Initial status of fluents). *If a fluent initially holds, a corresponding declipping event is generated at time 0.*

$$\mathrm{initially}(F) \rightarrow \mathbf{H}(\mathrm{declip}(F), 0). \qquad (\mathrm{rec}_3)$$

Operationally, Axiom (rec_3) enforces the generation of a set of **H** events that are needed for the correct extension of MVIs.

1 As we will see, the expressiveness of CLIMB is not sufficient to express REC, in that happened events and a further predicate must be defined as abducibles.

AXIOM 14.4 (Fluents initiation). *If an event* Ev *occurs at time* T *which initiates fluent* F, *either* F *already holds or it is declipped.*

$$\begin{aligned}&\mathbf{H}(\mathsf{event}(\mathsf{Ev}),\mathsf{T}) \land \mathsf{initiates}(\mathsf{Ev},\mathsf{F},\mathsf{T})\\ \rightarrow&\mathbf{H}(\mathsf{declip}(\mathsf{F}),\mathsf{T})\\ &\lor \mathbf{E}(\mathsf{declip}(\mathsf{F}),\mathsf{T_d}) \land \mathsf{T_d} < \mathsf{T}\\ &\quad \land \mathbf{EN}(\mathsf{clip}(\mathsf{F}),\mathsf{T_c}) \land \mathsf{T_c} > \mathsf{T_d} \land \mathsf{T_c} < \mathsf{T}.\end{aligned} \qquad (\mathsf{rec_4})$$

Axiom ($\mathsf{rec_4}$) does not use the $\mathsf{holds_at}$ predicate and it does not incur a new MVI.

AXIOM 14.5 (Impact of initiation). *The happening of a* declip(F) *event causes fluent* F *to start to hold.*

$$\mathbf{H}(\mathsf{declip}(\mathsf{F}),\mathsf{T_s}) \rightarrow \mathbf{mvi}(\mathsf{F},[\mathsf{T_s},\mathsf{T_e}]) \land \mathsf{T_e} > \mathsf{T_s}. \qquad (\mathsf{rec_5})$$

AXIOM 14.6 (Fluents termination). *If an event* Ev *occurs which terminates a fluent* F, *and* F *holds, then it is clipped.*

$$\begin{aligned}&\mathbf{H}(\mathsf{event}(\mathsf{Ev}),\mathsf{T}) \land \mathsf{holds_at}(\mathsf{F},\mathsf{T})\\ &\qquad \land \mathsf{terminates_at}(\mathsf{Ev},\mathsf{F},\mathsf{T}) \rightarrow \mathbf{H}(\mathsf{clip}(\mathsf{F}),\mathsf{T}).\end{aligned} \qquad (\mathsf{rec_6})$$

AXIOM 14.7 (Final clipping of fluents). *All fluents are terminated by the special* complete *event.*

$$\mathsf{terminates}(\mathsf{complete},\mathsf{F}). \qquad (\mathsf{rec_7})$$

The *complete* event is used to state that no more events will occur, and therefore it clips all the still holding fluents. It can be used also to alert `sciff` that the *closure* transition must be applied in a deterministic way.

14.3 REC ILLUSTRATED: A PERSONNEL MONITORING FACILITY

The following illustration shows the usage and potential impact of REC in a real-world case study. It was proposed to us by a local medium-sized enterprise based in Emilia-Romagna, but the same situation can apply to any organization of any size, in the private or public sector. It is about a personnel monitoring activity, which costs a considerable amount of human resources, but could be solved instantly and in a fully automated way using REC.

System events

A company wants to monitor its personnel's time-sheets. Each employee punches the clock when entering or leaving the office. The system recognizes two events:

- $\mathsf{check_in}(\mathsf{E})$: employee E has checked in;
- $\mathsf{check_out}(\mathsf{E})$: employee E has checked out.

Requirements on personnel

The following requirements on employee behaviour require monitoring:

(R0) after check in, an employee must check out within 8 hours;

(R1) as soon as a deadline expiration is detected, a dedicated alarm fires at an operator's desk. It reports the employee ID, and an indication of the time interval elapsed between deadline expiration and its detection. The alarm is turned off when the operator decides to handle it.

Note that the second requirement is activated when the first one is violated. Hence, such a system cannot be modeled directly by means of a ConDec model or a CLIMB specification, because the violation of the first requirement would cause sciff to terminate the verification.

Operator actions

We assume that the following actions are available to the operator:

- $handle(E)$ states that the operator wants to handle the situation concerning the employee identified by E;
- tic is used to take a snapshot of the current situation of the system, by updating the current time.

14.3.1 *Formalizing the Personnel Monitoring Facility in REC*

We capture requirements (R0) and (R1), using three fluents:

- $in(E)$: E is currently in;
- $should_leave(E, T_d)$: E is expected to leave her office by T_d;
- $alarm(delay(E, D))$: E has not left the office in time – D represents the difference between the time a deadline expiration is detected and the deadline expiration time itself.

It is possible to model such requirements declaratively using $initiates$ and $terminates$ predicate definitions. We assume hour time granularity.

Let us first focus on the $in(E)$ fluent. E is in as of the time she checks in. She ceases to be in as of the time she checks out:

$$initiates(check_in(E), in(E), _).$$
$$terminates(check_out(E), in(E), T) \leftarrow holds_at(in(E), T).$$

When E checks in at T_c, a $should_leave$ fluent is activated, expressing that E is expected to leave the office by $T_c + 8$[2]:

$$initiates(check_in(E), should_leave(E, T_d), T_c) \leftarrow T_d \text{ is } T_c + 8.$$

Such a fluent can be terminated in two ways: either E correctly checks out within the 8-hour deadline, or the deadline expires. In the latter case, termination is imposed at the next tic action.

2 Note that T_c is ground at body evaluation time, due to Axiom (rec_4).

$$\mathsf{terminates}(\mathsf{check_out}(E), \mathsf{should_leave}(E, T_d), T_c) \leftarrow \mathsf{holds_at}(\mathsf{should_leave}(E, T_d), T_c) \wedge T_c \leqslant T_d.$$

$$\mathsf{terminates}(\mathsf{tic}, \mathsf{should_leave}(E, T_d), T) \leftarrow \mathsf{holds_at}(\mathsf{should_leave}(E, T_d), T) \wedge T > T_d.$$

The same tic action also causes an alarm to go off:

$$\mathsf{initiates}(\mathsf{tic}, \mathsf{alarm}(\mathsf{delay}(E, D), T) \leftarrow \mathsf{holds_at}(\mathsf{should_leave}(E, T_d), T) \wedge D \text{ is } T - T_d.$$

Past vs future dependent properties

Note that in these equations the time of event termination/start (T_c and T) is the same time present in their respective body's holds_at atoms. This is perfectly normal, but it is not a requirement. In particular, times could be different, as long as the times of holds_at atoms do not follow event termination/start times. That again would be allowed, but it would amount to defining properties that depend on future events: properties that are thus not suitable for runtime monitoring. For that reason, we assume that *well-formed* theories do not contain such kind of clauses. More details on this matter will be given below when we discuss the *irrevocability* property in a formal way.

Finally, an alarm is turned off when the operator handles it:

$$\mathsf{terminates}(\mathsf{handle}(E), \mathsf{alarm}(\mathsf{delay}(E, D)), T) \leftarrow \mathsf{holds_at}(\mathsf{delay}(E, D), T).$$

14.3.2 *Monitoring a Concrete Instance*

Based on such a theory, REC becomes able to dynamically reason from the employees' flow inside the company. In particular, REC tracks the status of each employee, and generates an alarm as soon as a tic action detects a deadline expiration.

Let us consider an execution trace involving two employees e_1 and e_2, where e_2 does respect the required deadline while e_1 does not:

$$\mathbf{H}(\mathsf{event}(\mathsf{check_in}(e_1)), 9), \quad \mathbf{H}(\mathsf{event}(\mathsf{tic}), 10),$$
$$\mathbf{H}(\mathsf{event}(\mathsf{check_in}(e_2)), 11), \quad \mathbf{H}(\mathsf{event}(\mathsf{tic}), 14),$$
$$\mathbf{H}(\mathsf{event}(\mathsf{tic}), 16), \quad \mathbf{H}(\mathsf{event}(\mathsf{tic}), 18).$$

Figure 57 shows the global state of fluents at 18, when REC generates an alarm because e_1 was expected to leave the office no later than 17, but she has not left yet. The operator can check all pending alarms, and pick an employee to handle in case. The execution now proceeds as follows:

$$\mathbf{H}(\mathsf{event}(\mathsf{check_out}(e_2)), 19), \quad \mathbf{H}(\mathsf{event}(\mathsf{handle}(e_1)), 22),$$
$$\mathbf{H}(\mathsf{event}(\mathsf{check_out}(e_1)), 23).$$

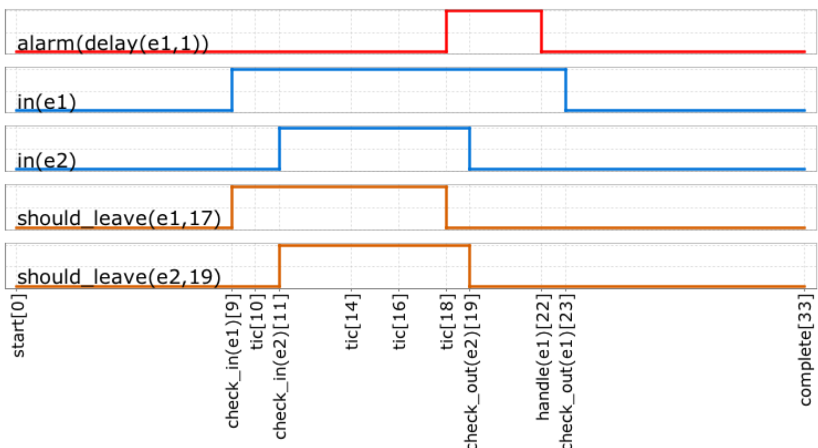


Figure 57: Fluents tracking with REC.

e_2 correctly leaves the office within the deadline, bringing her corresponding in and $\mathit{should_leave}$ fluents to termination. At 22 the operator handles an alarm involving e_1, who eventually leaves her office at 23.

14.3.3 *The Irrevocability Issue*

In general, a monitoring application is usable only if it provides stable, deterministic outputs, which do not flutter due to algorithmic issues but only change as a sensible response to the inputs. The reasons of an undesired fluttering behaviour could be of two types: a bad set of specifications, or an unsuitable underlying reasoning machinery.

As an example of the former, imagine to replace

$$\mathit{initiates}(\mathit{check_in}(E_1), \mathit{in}(E_2), _).$$

by

$$\mathit{initiates}(\mathit{check_in}(E_1), \mathit{in}(E_2), _).$$

This is an ambiguous specification since it does not clearly state which employee should change status as a consequence of E_1 checking in. Another different source of ambiguity could lay hidden in an alternative formulation of fhe firing alarm rule:

$$\begin{aligned} \mathit{initiates}(\mathit{tic}, \mathit{alarm}(\mathit{delay}(E, D), T) \leftarrow \\ \mathit{holds_at}(\mathit{should_leave}(E, T_d), T_1) \\ \wedge D \text{ is } T - T_d \wedge T_1 > T. \end{aligned}$$

The meaning would be that an action is a consequence of an alarm which has not fired yet. Only speculations are possible in that case, and no run-time monitoring algorithm could provide deterministic answers (save freezing until the alarm fires, but in that case the application would no longer be called “run-time”).

Thus we need to isolate "good" sets of specifications. Once we have them, we must ensure that the reasoning machinery does not make unjustified retractions. In other words, we must guarantee *irrevocability*.

14.4 formal properties of rec

REC is sound and complete

REC is implemented on top of the SCIFF-lite language, it thus inherits the soundness and completeness results of the declarative semantics with respect to the sciff operational semantics[3]. These are important results. The operational behaviour of REC is faithful to its axiomatization. We are unaware of other implementations of the EC that provide such a guarantee. This is a consequence of the axiom implementation by SCIFF-lite formulae. The next results concern uniqueness and irrevocability, and they are instead relative to the special kind of reasoning needed for the monitoring applications.

14.4.1 *Irrevocability of REC*

As in the case of run-time verification, we could often rely on the assumption that events occur in ascending order. As we have pointed out in Section 13.2.3, this assumption enables semi-open reasoning of sciff.

The concept of irrevocability

In this context, it is required that the generated MVIs are never retracted, but only extended or terminated as new events occur. If that is the case, the reasoning process is called *irrevocable*. Some target applications need irrevocable tracking procedures, which can give a continuously updated view of the status of all fluents. Fluttering behaviours must be by all means avoided. This is true, e.g., when the modeled fluents carry a normative meaning. It would be undesirable, for instance, to attach a certain obligation to an agent at runtime, and see it disappear later only because the calculus revises its computed status.

In the reminder of this section, we first define a class of REC theories, then we show that semi-open reasoning on the resulting REC specifications is irrevocable.

Definition 14.1 (Well-formed REC theory). A well-formed REC theory $\mathcal{T}$ is a set of clauses of the type[4]

$$\mathtt{initiates}(Ev, F, T) \leftarrow \mathtt{body}.$$
$$\mathtt{terminates}(Ev, F, T) \leftarrow \mathtt{body}.$$

which satisfies the following properties:

1. negation is not applied to holds_at predicates;

3 The soundness and completeness results of sciff have been recalled in Section 9.2 for the case of CLIMB; the theorems and proofs for the general full-SCIFF language can be instead found in [7].

4 The body is a conjunction of holds_at predicates and CLP constraints. It can be omitted when true.

2. for $\mathtt{initiates}/3$ clauses, fluent F must always be resolved with a ground substitution.

3. $\forall \mathtt{holds_at}(F_2, T_2)$ predicate used in $\mathtt{body}$, $T_2 \leqslant T$.

Why well-formedness

In the Section 14.3.1 our illustration was modeled by a well-formed REC theory. Using the same example, in Section 14.3.3 we have discussed the consequences of ill-formed specifications in the REC theory in a concrete case. Definition 14.1 identifies in the general case the three possible sources of non irrevocability. In particular, 1. ensures monotonicity, 2. prevents non-determinism due to the *case analysis* transition of `sciff` and 3. restricts us to reasoning on stable, past conditions. REC theories that violate 2. and 3. would introduce choice points that hinder irrevocability.

DEFINITION 14.2 (REC specification). Given a well-formed REC theory $\mathcal{T}$, the corresponding REC specification $\mathcal{R}^{\mathcal{T}}$ is defined as the SCIFF-lite specification[5]:

$$\mathcal{R}^{\mathcal{T}} \equiv \langle KB_{REC} \cup \mathcal{T}, \{\mathbf{E}, \mathbf{EN}, \mathbf{H}, \mathbf{mvi}\}, IC_{REC} \rangle$$

where $KB_{REC} = \{(\mathrm{rec}_1), (\mathrm{rec}_7)\}$ and $IC_{REC} = \{(\mathrm{rec}_2), (\mathrm{rec}_3), \ldots, (\mathrm{rec}_6)\}$.

The following three lemmas establish interesting properties of REC, defining the link between MVIs and the internal events used to clip and declip them.

LEMMA 14.1 (Groundedness of MVIs' starting times). *For each well-formed REC theory $\mathcal{T}$, for each intial execution trace $\mathcal{T}_i$ and for each final execution trace $\mathcal{T}_f$, the abduced MVIs always have a ground starting time, i.e.*

$$\forall\ \Delta, \mathcal{R}^{\mathcal{T}}_{\mathcal{T}_i} \models^{\mathcal{T}_i}_{\Delta} \mathtt{true} \Rightarrow \forall\ \mathbf{mvi}(F, [T_s, T_e]) \in \Delta, T_s \in \mathbb{N}$$

Proof. Axiom (rec_5) is the only IC in which the abducible representing a MVI appears in the head. The starting time T_s is bound to the time at which the $\mathtt{declip}$ event in the body happens. This event is not contained in the execution trace, but is instead generated by applying axiom (rec_3) or (rec_4). By axiom (rec_3), a $\mathtt{declip}$ event is generated at time 0, whereas by axiom (rec_4), a $\mathtt{declip}$ event is generated using the same time of the body's external **H** event. Since all happened events contained in an execution trace are ground, then also the $\mathtt{declip}$ event is abduced to happen at a ground time. Therefore, also the starting time T_s of the corresponding MVI is ground. □

LEMMA 14.2 (Relationship between clipping events and MVIs). *The expectation about the clipping of a MVI can be fullfilled by exactly one happened event, in particular the nearest which occurs after the declipping of the MVI.*

5 Throughout the discussion, we will simply use $\mathcal{R}$ to identify a generic REC specification. We will also state that a REC specification is well-formed as a shortcut to state that its theory is.

Proof. Let us consider by abdsurdum that there exists $\textbf{mvi}(f, [t_s, T_e])$ triggering Axiom (rec_2) and generating an expectation about $clip(f)$ which can potentially be fulfilled by two distinct event happening, say, at time $t_{e1} > t_s$ and $t_{e2} > t_{e1}$. If $\textbf{E}(clip(f), T_e)$ is fulfilled at time t_{e2}, then Axiom (rec_2) also imposes $\textbf{EN}(clip(f), T_c)$ between t_s and t_{e2}. However, this time interval also includes t_{e1}, leading to inconsistency. □

Lemma 14.3 (Interleaving between declipping and clipping events). *For each fluent, between two declipping events at least one clipping event must occur.*

Proof. Let us suppose that there exists a fluent f for which two $declip(f)$ events occur, say, at time t_1 and $t_2 > t_1$, without having a $clip(f)$ in-between. An MVI is generated for f starting at t_1 (thank to Axiom (rec_5)); this MVI will be clipped by the first consequent time at which a $clip(f)$ occurs (see Lemma 14.2). The hypothesis which states that this $clip(f)$ event must occur after t_2 is however inconsistent, because Axiom (rec_2) would state that between t_1 and this time (thus t_2 included) no further $declip(f)$ event can occur. □

We are now ready to state the following:

Theorem 14.1 (Uniqueness of derivation). *For each well-formed REC theory $\mathcal{T}$, for each initial execution trace $\mathcal{T}_i$ and for each final execution trace $\mathcal{T}_f$, there exists exactly one successful semi-open derivation computed by* sciff, *i.e.* $\exists 1 \Delta$ *s.t.* $\mathcal{R}^{\mathcal{T}}_{\mathcal{T}_i} \models^{\mathcal{T}_f}_{\Delta} true$.

Proof. First, we prove that at most one computed explanation exists. Different explanations are computed when inclusive disjunctions are contained in the head of some integrity constraint, or if there are different ways to fulfill a positive expectation[6].

Let us first consider the case of disjunctions in the head. The only integrity constraint containing a disjunctive head is (rec_4);[7] however, we prove that these two disjuncts are mutually exclusive. Let us consider the second disjunct. It states that fluent F has been already declipped at a certain past time (let us denote it with t_d), and that between t_d and T no clipping event has been generated: thus F still holds at time T. In fact, when $declip(F)$ happened at time t_d, a **mvi** abducible was generated by applying rule (rec_5); due to the negative expectation contained in the second disjunct of (rec_5)'s head, this maximal validity interval must be clipped at a time greater than T (say, t_c). The application of rule (rec_2), in turn, states that it is forbidden to declip F between t_d and t_c, i.e., also at time T. Therefore, the first disjunct of (rec_5)'s head cannot be true at time T. A further important observation concerns the

6 The other possibilities of having multiple explanations are ruled out by the fact that `initiates` predicate are resolved with a ground fluent, thus MVIs are always ground.

7 The presence of negated abducibles in the body of a rule would also produce inclusive disjunctive head [82], but Definition 14.1 forbids negated `holds_at` predicates.

semi-open nature of the derivation. Event if the first disjunct of Axiom (rec_5) did not lead to violation, an open derivation would open a choice point, waiting for a suitable past declipping event to fulfill the expectation in the second disjunct. As we have just proven, this second possibility is impossible, because the two disjuncts are mutually exclusive. A semi-open derivation is immediately able to detect this situation, because the second disjunct refers to the past, and the first one to the present. Therefore, no choice point is left open.

Let us now consider the reasons to fulfill positive expectations, which are present in Axioms (rec_2) and (rec_4). $\mathbf{E}(declip(F), T_s)$ in Axiom (rec_2) can be fulfilled in one way, because both F and T_s are ground, the former because the theory is well-defined by hypothesis, the latter as stated in Lemma 14.1. ByLemma 14.2, also the expectation about the clipping of the fluent can be fulfilled by exactly one event, in particular by the first clipping occurring after declip. Finally, the positive expectation in Axiom (rec_4) can be fulfilled by only one declip event; the proof is obtained by combining the negative expectation about the clipping of the fluent in Axiom (rec_4) and the result proven in Lemma 14.3.

Now we prove that there always exists a computed explanation, i.e. that all execution traces comply with the REC specifications. The axioms imposing expectations are (rec_2) and (rec_4). If the positive expectation in the second disjunct of (rec_4)'s head cannot be fulfilled, then the involved fluent is not holding, and therefore it can be declipped by choosing the first disjunct. When the goal is true, Axiom (rec_2) only fires if Axiom (rec_5) fires, and therefore the positive expectation about the declipping of the fluent has a corresponding matching event (exactly the one which has caused Axiom (rec_5) to fire). The positive expectation about the clipping of fluent is eventually fulfilled by the special complete event, which is able to terminate all fluents (see Axiom (rec_7)). Finally, the negative expectations contained in Axioms (rec_2) and (rec_4) are simply used to select the "nearest" declipping/-clipping, but are not used to rule out executions. □

Theorem 14.1 ensures that exactly one Δ is produced by a semi-open derivation of sciff. This, in turn, means that there exists exactly one "configuration" for the MVIs of each fluent. We give a precise definition of this notion of state, which is the one of interest when evaluating the irrevocability of the reasoning process, and define the notion of progressive extension between states, which gives a formal account for irrevocability. Note that the definitions rely on the definition of current time given in Definition 13.2[8].

8 For the sake of readability, we report the definition here. Given an execution trace $\mathcal{T}$, the *current time* associated to $\mathcal{T}$ is:

$$ct(\mathcal{T}) \triangleq \max\{t \mid \mathbf{H}(e, t) \in \mathcal{T}\}$$

DEFINITION 14.3 (MVI State). Given a REC specification $\mathcal{R}$ and an execution trace $\mathcal{T}$, the resulting *MVI state* at time $\mathtt{ct}(\mathcal{T})$ is the set of **mvi** abducibles contained in the computed explanation generated by `sciff`:

$$MVI(\mathcal{R}_\mathcal{T}) \equiv \{\mathbf{mvi}(F, [T_s, T_e]) \in \Delta\}, \text{ where } \mathcal{R}_\emptyset \vdash^{\mathcal{T}}_{\Delta} \mathtt{true}$$

DEFINITION 14.4 (State sub-sets). Given a REC specification $\mathcal{R}$ and a (partial) execution trace $\mathcal{T}$, the current state $MVI(\mathcal{R}_\mathcal{T})$ is split into two sub-sets:

- $MVI_{\sqcap\!\sqcup}(\mathcal{R}_\mathcal{T})$, is the set of (closed) MVIs, terminating at a ground time:

$$MVI_{\sqcap\!\sqcup}(\mathcal{R}_\mathcal{T}) = \{\mathbf{mvi}(F, [s, e]) \in MVI(\mathcal{R}_\mathcal{T}) \mid s, e \in \mathbb{N}\};$$

- $MVI_{\sqcap}(\mathcal{R}_\mathcal{T})$, is the set of (open) MVIs, terminating at a variable time:

$$MVI_{\sqcap}(\mathcal{R}_\mathcal{T}) = \{\mathbf{mvi}(F, [s, T]) \in MVI(\mathcal{R}_\mathcal{T}) \mid s \in \mathbb{N}, var(T)\}.$$

DEFINITION 14.5 (Trace extension). Given two execution traces $\mathcal{T}_1$ and $\mathcal{T}_2$, $\mathcal{T}_2$ is an *extension* of $\mathcal{T}_1$, written $\mathcal{T}_1 \prec \mathcal{T}_2$, iff

$$\forall\, \mathbf{H}(e, t) \in \mathcal{T}_2/\mathcal{T}_1,\ t \geqslant \mathtt{ct}(\mathcal{T}_1)$$

DEFINITION 14.6 (State progressive extension). Given a well-formed REC specification $\mathcal{R}$ and two execution traces $\mathcal{T}_1$ and $\mathcal{T}_2$, the state of $\mathcal{R}_{\mathcal{T}_2}$ is a *progessive extension* of the state of $\mathcal{R}_{\mathcal{T}_1}$, written $MVI(\mathcal{R}_{\mathcal{T}_1}) \trianglelefteq MVI(\mathcal{R}_{\mathcal{T}_2})$, iff

1. the set of closed MVIs is maintained in the new state:
   $MVI_{\sqcap\!\sqcup}(\mathcal{R}_{\mathcal{T}_1}) \subseteq MVI_{\sqcap\!\sqcup}(\mathcal{R}_{\mathcal{T}_2})$
2. if the set of MVIs is extended with new MVIs, these are declipped after the maximum time of $\mathcal{T}_1$:
   $\forall\, \mathbf{mvi}(f, [s, t]) \in MVI(\mathcal{R}_{\mathcal{T}_2})/MVI(\mathcal{R}_{\mathcal{T}_1}), s > \mathtt{ct}(\mathcal{T}_1)$
3. $\forall\, \mathbf{mvi}(f, [s, T_e]) \in MVI_{\sqcap}(\mathcal{R}_{\mathcal{T}_1})$, either
   a) it remains untouched in the new state:
      $\mathbf{mvi}(f, [s, T_e]) \in MVI_{\sqcap}(\mathcal{R}_{\mathcal{T}_2})$, or
   b) it is clipped after the maximum time of $\mathcal{T}_1$:
      $\mathbf{mvi}(f, [s, e]) \in MVI_{\sqcap\!\sqcup}(\mathcal{R}_{\mathcal{T}_2}), e > \mathtt{ct}(\mathcal{T}_1)$.

Progressive extensions capture the intuitive notion that a state extends another one if it keeps the already computed closed MVIs and affects the status of fluents only at later times w.r.t. the time the first state was recorded. The extension is determined by adding new MVIs and by clipping fluents which held at the previous state. We can state the main result leading to irrevocability, namely that extending a trace results in a progressive extension of the MVI state.

LEMMA 14.4 (Trace extension leads to a state progressive extension). *Given a well-formed* REC *specification* $\mathcal{R}$ *and two execution traces* $\mathcal{T}_1$ *and* $\mathcal{T}_2$*,*

$$\mathcal{T}_1 \prec \mathcal{T}_2 \Rightarrow \text{MVI}(\mathcal{R}_{\mathcal{T}_1}) \trianglelefteq \text{MVI}(\mathcal{R}_{\mathcal{T}_2})$$

Proof. Let us consider Definition 14.6, showing that $MVI(\mathcal{R}_{\mathcal{T}_1})$ and $MVI(\mathcal{R}_{\mathcal{T}_2})$ obey to its four requirements:

1. Let us consider an element of $MVI_{\sqcap\sqcup}(\mathcal{R}_{\mathcal{T}_1})$, say, $\mathbf{mvi}(f, [s, e])$. The presence of this element is evidence that an event $\in \mathcal{T}_1$ occurring at time e exists s.t. fluent f is declipped. Since this event belongs to $\mathcal{T}_1$, $e < \mathtt{ct}(\mathcal{T}_1)$. From the hypotheses that $\mathcal{T}_1 \prec \mathcal{T}_2$, all the events that belong to $\mathcal{T}_2/\mathcal{T}_1$ happen at a time greater than $\mathtt{ct}(\mathcal{T}_1)$. Lemma 14.2 thus ensures that none of these events can change $\mathbf{mvi}(f, [s, e])$, which is maintained untouched in $MVI_{\sqcap\sqcup}(\mathcal{R}_{\mathcal{T}_2})$.

2. As pointed out in the proof of Lemma 14.1, a happened event can start a new MVI at the time it happens. Since $\mathcal{T}_1 \prec \mathcal{T}_2$, each MVI generated by events belonging to $\mathcal{T}_2/\mathcal{T}_1$ will always have a starting time greater than $\mathtt{ct}(\mathcal{T}_1)$. The only unfortunate case would be that there exists an event associated to an `initiates` predicate that can be evaluated as true `after` the time at which the event occurs. This case arises when the `initiates` predicate is defined in terms of `holds_at` predicates which refer to the future[9] However, Definition 14.1 rules out theories of this kind.

3. Axiom ($\mathtt{rec}_6$) is the rule which regulates the way fluents are clipped; a happened event can cause the termination of an MVI exactly at the time at which it occurs. From the hypotheses that $\mathcal{T}_1 \prec \mathcal{T}_2$, if an MVI is open at time $\mathtt{ct}(\mathcal{T}_1)$ (i.e., it belongs to $MVI_{\sqcap\sqcup}(\mathcal{R}_{\mathcal{T}_1})$), it is impossible for an event belonging to $\mathcal{T}_2/\mathcal{T}_1$ to clip it before time $\mathtt{ct}(\mathcal{T}_1)$. Indeed, as in the case of `initiates` predicates, theories which lead to violate this property are not well-formed. Two possible cases for such an MVI may then arise:

   a) no event $\in \mathcal{T}_2/\mathcal{T}_1$ is able to terminate the fluent associated to the MVI, which is therefore maintained untouched in $MVI_{\sqcap\sqcup}(\mathcal{R}_{\mathcal{T}_1})$;

   b) there exists at least one event $\in \mathcal{T}_2/\mathcal{T}_1$ able to terminate the fluent associated to the MVI; the first one (see Lemma 14.2) will shift the MVI from the open to the closed set and respect the requirement that the MVI termination time must be greater than $\mathtt{ct}(\mathcal{T}_1)$.

□

THEOREM 14.2 (Irrevocability of $\mathcal{REC}$). *Given a well-formed* REC *specification and a temporally ordered narrative, each time* `sciff` *processes a new event, the new MVI state is a progressive extension of the previous one.*

9 For example, if the user states that "event e initiates fluent f at time T if fluent f_2 holds at time $T + 2$", then it could be the case that at time $T + 2$ f_2 indeed holds, causing f to be declipped in the past and violating Item 2 of Definition 14.6.

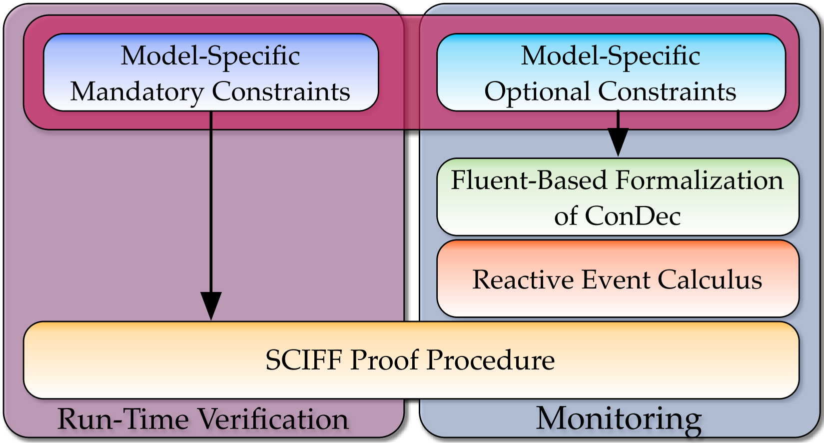


Figure 58: ConDec run-time verification and monitoring.

Proof. Let us suppose that the current execution trace is $\mathcal{T}_1$, and that a new happened event $\mathbf{H}(e,t)$ is acquired by SCIFF. Let us denote the new execution trace with $\mathcal{T}_2$, having $\mathcal{T}_2 = \mathcal{T}_1 \cup \{\mathbf{H}(e,t)\}$. If the execution of the system always grows by increasing times, then $t > \mathtt{ct}(\mathcal{T}_1)$. Therefore, from Definition 14.5 it holds that $\mathcal{T}_1 \prec \mathcal{T}_2$ and, in turn, Lemma 14.4 ensures that $MVI(\mathcal{R}_{\mathcal{T}_1}) \trianglelefteq MVI(\mathcal{R}_{\mathcal{T}_2})$, i.e. that the new state is a progressive extension of the previous one. □

14.5 MONITORING OPTIONAL CONSTRAINTS WITH REC

Having shown how a reactive irrevocable form of EC can be embedded into the SCIFF framework, we now exploit the advanced features provided by REC to monitor optional ConDec constraints. Being REC axiomatized on top of SCIFF, it can be seamlessly combined with a CLIMB specification formalizing the mandatory constraints, using `sciff` as a monitoring and run-time verification infrastructure at the same time (see Figure 58).

14.5.1 *Representing ConDec Optional Constraints in REC*

To enable monitoring facilities, a translation of ConDec (optional) constraints to REC must be provided. We propose a formalization separated in two parts:

- a general part, which describes how the different ConDec constraints can be formalized as fluents in the EC setting, and how such fluents are initiated and terminated, independently from the specific models;
- a specific part, whose purpose is to describe a specific ConDec diagram.

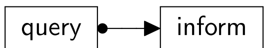
Both parts are inserted in the knowledge base of the specification, and are then used in conjunction with the general reactive axioms of REC for monitoring.

Model-specific part

The model-specific part is a set of $\mathtt{opt_con}/2$ facts. Each fact corresponds to a ConDec optional constraint in the diagram, associating a unique identifier to it. For example

$$\mathtt{opt_con}(c_1, \mathtt{response}(\mathtt{query}, \mathtt{inform}))$$

states that the ConDec model contains an optional constraint



and that such a constraint is identified by c_1. The translation of optional constraints is therefore straightforward.

General part

The general part is a set of predicate definitions aimed at formalizing optional constraints in terms of fluents and linking their initiation and termination to the occurrence of activities. Since fluents are used to characterize a partial state of the system, we provide a formalization capturing the *status* of optional constraints. In particular, constraints can be divided into two families:

- Constraints that must be eventually *satisfied*, and that could become temporarily unsatisfied when a certain activity is executed. An example is the response constraint. a •—▶ b is initially satisfied; if a is performed, the constraint switch to a temporary unsatisfied state, becoming satisfied again after an execution of activity b.
- Constraints that *forbid* the presence of a certain activity when they are active. An example is the negation response constraint. When activity a occurs, a •—‖—▶ b is activated and forbids the execution of b in the future.

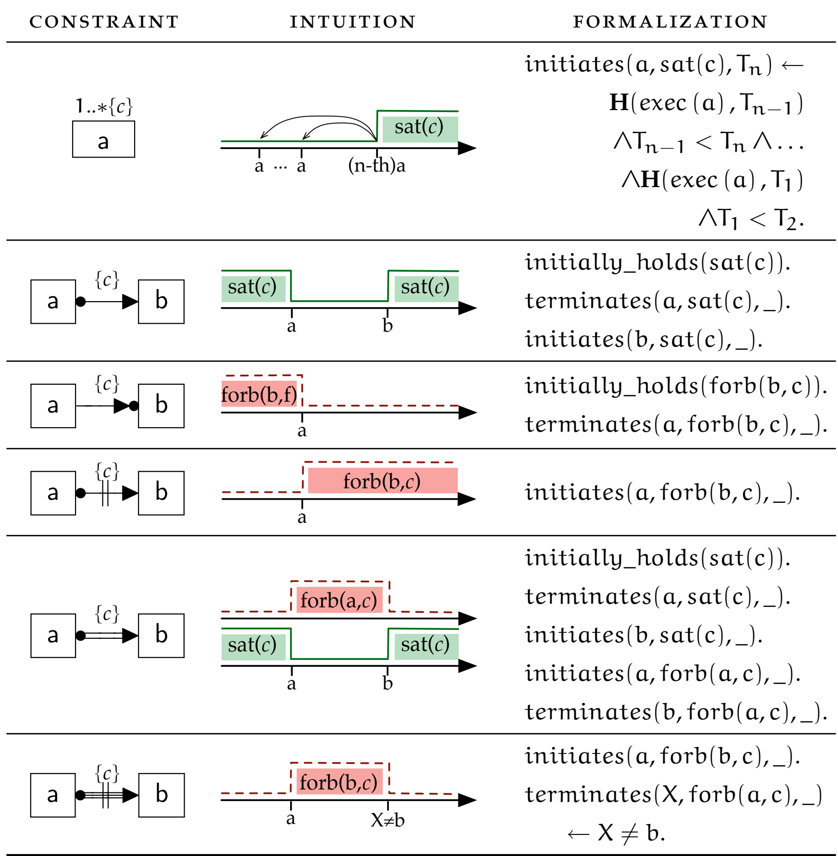
We therefore rely on two different fluents:

- $\mathtt{sat}(c)$ states that the optional constraint c is currently satisfied;
- $\mathtt{forb}(a, c)$ states that the optional constraint c is currently active, and it is forbidding the execution of activity a.

Table 30 briefly indicates our usage of fluents in the formalization of some ConDec constraints. The full theory can be found in [145]. It is worth noting that the proposed theory is well-formed, and thus guarantees irrevocability.

Some parts of the formalization are left implicit for ease of presentation. In particular, Table 30 omits the binding between each formalization and its corresponding $\mathtt{opt_con}/2$ fact. For example, the complete formalization of response would be:

$$\begin{aligned}
\mathtt{initially_holds}(\mathtt{sat}(C)) &\leftarrow \mathtt{opt_con}(C, \mathtt{response}(A, B)).\\
\mathtt{terminates}(A, \mathtt{sat}(C), _) &\leftarrow \mathtt{opt_con}(C, \mathtt{response}(A, B)).\\
\mathtt{initiates}(B, \mathtt{sat}(C), _) &\leftarrow \mathtt{opt_con}(C, \mathtt{response}(A, B)).
\end{aligned}$$



CONSTRAINT	INTUITION	FORMALIZATION
1..∗{c} a	sat(c); a ··· a (n-th)a	$\text{initiates}(a, \text{sat}(c), T_n) \leftarrow \mathbf{H}(\text{exec}(a), T_{n-1}) \land T_{n-1} < T_n \land \ldots \land \mathbf{H}(\text{exec}(a), T_1) \land T_1 < T_2.$
a {c} b	sat(c) sat(c); a b	initially_holds(sat(c)). terminates(a, sat(c), _). initiates(b, sat(c), _).
a {c} b	forb(b,f); a	initially_holds(forb(b, c)). terminates(a, forb(b, c), _).
a {c} b	forb(b,c); a	initiates(a, forb(b, c), _).
a {c} b	forb(a,c); sat(c) sat(c); a b	initially_holds(sat(c)). terminates(a, sat(c), _). initiates(b, sat(c), _). initiates(a, forb(a, c), _). terminates(b, forb(a, c), _).
a {c} b	forb(b,c); a X≠b	initiates(a, forb(b, c), _). terminates(X, forb(a, c), _) $\leftarrow X \neq b$.

Table 30: Representing some optional ConDec constraint in REC.

The first clause states that if the ConDec model under study contains an optional response constraint, then such a constraint is initially satisfied. The second clause points out that when the source of the response occurs, then the constraint is no more satisfied. The last clause brings the constraint back to a satisfied status when the target activity of the response is executed.

Note that, differently from the CLIMB formalization, all the ConDec constraints are represented in REC as forward constraint, in order to ensure irrevocability. For example, constraint a ——▶● b, which has been represented in CLIMB by stating that if b is executed, then a previous occurrence of a is expected, is now represented as "b is forbidden by the constraint unless a is executed". In the REC ontology, this means that b is initially forbidden by the precedence constraint, until an execution of a terminates such a prohibition.

Dealing with branching constraints

The proposed formalization can be easily adapted to deal also with branching constraints. To model branches, we extend the way constraints are represented by considering lists of activities instead of individual activities. We then adapt the formalization shown in Table 30, using membership constraints to specify that intitiating/terminating

activities range into a set. For example, the formalization of the branching response optional constraint is extended as follows:

$$\mathit{initially_holds}(\mathit{sat}(C)) \leftarrow \mathtt{opt_con}(C, \mathit{response}(_,_)).$$
$$\mathit{terminates}(A, \mathit{sat}(C), _) \leftarrow \mathtt{opt_con}(C, \mathit{response}(As, _) \land A :: As.$$
$$\mathit{initiates}(B, \mathit{sat}(C), _) \leftarrow \mathtt{opt_con}(C, \mathit{response}(_, Bs)) \land B :: Bs.$$

14.5.2 *Identification and Reification of Violations*

Having equipped REC with a fluent-based formalization of ConDec optional constraints, it is now possible to monitor an instance of the system, having a constantly updated and irrevocable view about the current status of each optional constraint. The last step is to identify violations of optional constraints, in order to report them to the system administrator.

Kind of violations

We focus on two kind of violations:

- violation of a prohibition, which occurs when an activity happens, but an holding fluent is forbidding its execution;
- deadline expiration, for optional constraints extended with metric temporal constraints (as specified in Section 6.1.3).

There exists a third kind of violation, related to unsatisfied constraints at the end of the execution. However, if the end of the execution has been reached, no identification of violation is needed anymore; if the user is anyway interested in controlling whether some optional constraints has finally been violated, she can simply observe the last status of constraints.

Modeling deadlines

Deadlines are expressed, in this context, as a maximum time span during which a certain constraint could persist in a "non satisfied" state[10]. For example, let us suppose that the model contains an optional constraint, called c_1, of the form

pay •—(–,24)—▶ deliver

c_1 specifies that when an order is paid, a receipt must be delivered within 24 time unit. It can be modeled in our framework by adding the information that c_1 cannot persist in a non satisfied state for more than 24 time units. We suppose that, to describe this condition, the user simply uses a $\mathit{deadline}(\mathit{sat}(c_1), 24)$ declaration. In general, $\mathit{deadline}(F, D)$ states that fluent F can persist in a "not-holding" state at most D time units.

Reification of violations

Since REC provides us a constantly updated view of the status associated to each constraint, we can easily combine the current status of a given constraint and the current occurrence of an activity, to explicitly identify if a violation is taking place. We add new axioms to

10 In the following, we will focus only on deadlines; delays can be handled in a complementary way.

capture the two kind of violations described above. Such axioms state that, if a violation takes place at time t , then a special event occurrence $\mathbf{H}(violation(f), t)$ is generated[11], where f is the fluent involved in the violation. In other words, violations are *reified* as happened events.

Violation of a Prohibition

Violation of a prohibition is related to the current presence of a holding forbidding fluent. In particular, we can identify and reify this kind of violation by stating that if an event occur at time T, and at that time there exists at least one holding fluent forbidding the event, then a violation takes place:

$$\mathbf{H}(exec\,(A)\,,T) \wedge holds_at(forb(A,C),T) \rightarrow \mathbf{H}(violation(forb(A,C)),T).$$

Deadline Expiration

The identification of a deadline expiration resembles the Personnel Monitoring Facility realized in Section 14.3.1. To capture and verify deadlines, we add four new axioms.

Let us suppose that fluent F is associated to a $deadline(F, D)$ condition. When F is terminated, a new fluent $d_check(F, T_e)$ is initiated. This fluent represents that F is currently monitored, to check if the associated deadline will be met by the execution; T_e denotes the time at which the deadline will expire. Such a situation can be formalized by means of the following axiom:

$$initiates(A, d_check(F, T_e), T) \leftarrow deadline(F, D), terminates(A, F, T), \\ T_e == T + D.$$

The fluent $d_check(F, T_e)$ can be terminated in two cases. In the first case, an event capable to terminate F happens within the deadline (i.e., within T_e):

$$terminates(A, d_check(F, T_e), T) \leftarrow deadline(F, _), initiates(A, F, T), T < T_e.$$

The second case deals with the expiration of the deadline. `sciff` has no notion of the flow of time: it becomes aware of the current time only when a new event occurs. Therefore, we can keep `sciff` up-to-date by generating special tic events. The deadline expiration is then detected and handled as soon as the first tic event after the deadline occurs:

$$terminates(tic, d_check(F, T_e), T) \leftarrow deadline(F, _), T \geqslant T_e.$$

11 Remember that SCIFF-lite allows for happened events in the head of ICs.

A further axiom recognizes this abnormal situation, by evaluating whether the deadline check fluent has been terminated after the expiration time (and generating a violation if it is the case):

$$\mathbf{H}(tic, T)$$
$$\land holds_at(deadline_check(F, T_e), T) \land T \geqslant T_e \rightarrow \mathbf{H}(violation(F), T).$$

14.5.3 *Compensating Violations*

Among the many possibilities offered by the reification of violations, an interesting option is to attach further constraints involving the special violation events. This could be a way to specify how the interacting parties must *compensate* for a violation, or to define a context for violations, i.e. to model constraints which become optional only in certain situations.

Modeling (critical) compensations

Compensation can be modeled by e.g. inserting a mandatory response constraint having a violation event as source, and the compensation activity as target; chain response could be then used to handle critical violations: it states that when the violation is detected, the next immediate activity to be executed is the compensating one.

Contextualization of violations

Contextualization of violations can be modeled using backward ConDec mandatory constraints (e.g., precedence). For example, modeling a precedence constraint involving an activity a and the event $violation(f)$ states that as soon as the violation is raised, REC verifies if an execution of activity a has been previously performed (the activity a representing some how the idea of context). In such a case, the violation can be managed, otherwise a definitive, negative answer attesting non-compliance is provided as a result.

14.5.4 *Monitoring Example*

We now briefly discuss a simple yet significative example of a choreography fragment, showing how the proposed approach can be fruitfully applied for run-time monitoring. Figure 59 shows the graphical ConDec representation of the example, augmented with a (contextualized) compensation. In order to show REC's ability to track the status of each constraint, we suppose that all the involved constraints are optional.

The choreography involves a customer, who creates an order by choosing one or more items, and a seller, who collects the ordered items and finally gives a receipt. The seller is committed to issue the final receipt within a pre-established deadline. Moreover, the seller offers the customer a fixed discount if he/she accepts some delays; in case of a delay, the seller also promises a further discount directly on the receipt.

In particular, the following rules of engagement must be fulfilled by the interacting services. It is worth noting that each constraint can be easily mapped by means of a (possibly extended) ConDec relation.

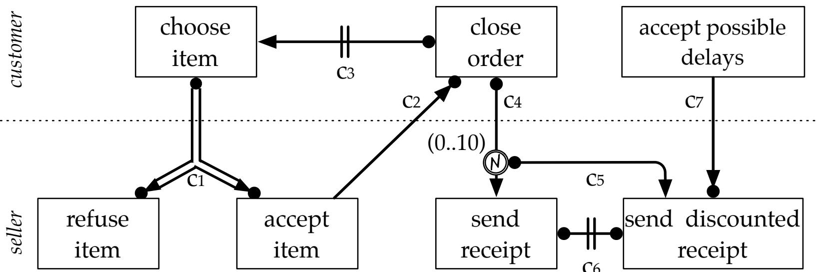


Figure 59: A ConDec choreography fragment, including a deadline and a compensation.

ID	REC REPRESENTATION
c_1	opt_con(c_1, alternate_succession([choose_item], [refuse_item, accept_item])).
c_2	opt_con(c_2, precedence([accept_item], [close_order])).
c_3	opt_con(c_3, negation_response([close_order], [choose_item])).
c_4	opt_con(c_4, response([close_order], [send_receipt])). deadline(satisfied(c_4), 10).
c_5	opt_con(c_5, response([violation(c_4)], [send_discounted_receipt])).
c_6	opt_con(c_6, precedence([accept_possible_delays], [send_discounted_receipt])).

Table 31: REC formalization of the choreography fragment shown in Figure 59.

- Every choose item activity must be followed by an answer from the seller, either positive or negative; no further upload can be executed until the response is sent. Conversely, each positive/negative response must be preceded by a choose item activity, and no further response can be sent until a new item is chosen (constraint c_1).
- If at least one uploaded item has been accepted by the seller, then it is possible for the customer to close the order (constraint c_2).
- When an order has been closed, no further item can be chosen (constraint c_3); moreover, the seller is committed to send a corresponding receipt by at most 10 time units (constraint c_4).
- If the seller does not meet the deadline, it must deliver a discounted receipt (constraint c_5, modeled as a response constraint triggered by the violation of constraint c_4; the graphical representation of the violation is inspired by the BPMN *intermediate error* event).

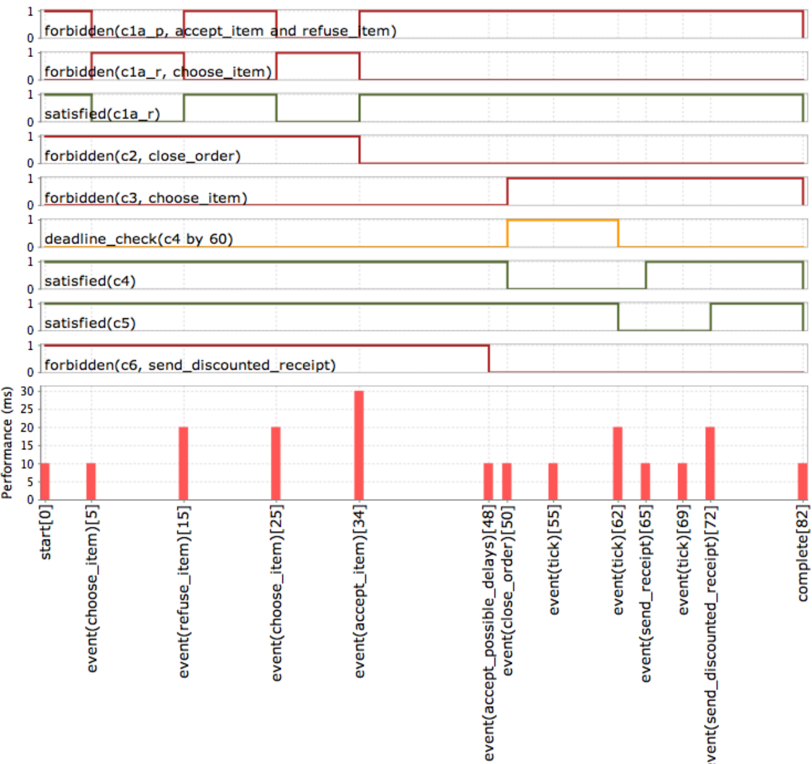


Figure 60: Fluents trend generated by REC when monitoring a specific interaction w.r.t. the diagram of Figure 59. The verification time spent for reacting to each happened event is also reported.

- Discounted receipt is enabled only if the customer has previously accepted the possibility of experiencing delays (constraint c_6).

Figure 60 illustrates how REC monitors a specific course of interaction w.r.t. the above described model. Clipping and declipping of fluents are handled at run-time, thus giving a constantly updated snapshot of the reached interaction status. In the bottom part of the figure, verification performance is reported, showing the amount of time spent by REC in order to dynamically react to and reason upon occurring events.

The central part of the execution shows how REC deals with a deadline expiration. Indeed, as soon as the activity close order is executed (at time 50), constraint c_4 becomes unsatisfied, and a corresponding deadline check is initiated, having 60 as expiration time. At time 62, a tic event makes the proof aware that the deadline related to the satisfaction of constraint c_4 is expired. As a consequence, `sciff` reacts by terminating the d_check fluent and by installing the corresponding compensation; this is attested by the fact that constraint c_5 becomes unsatisfied.

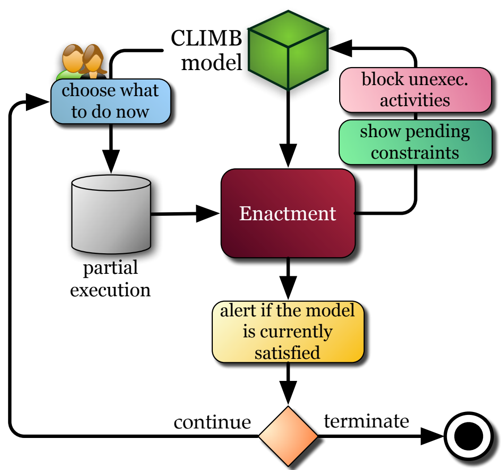


Figure 61: Enactment of ConDec models.

14.6 ENACTMENT OF CONDEC MODELS

The enactment of ConDec models consists in providing support to the interacting entities during the execution.

Enactment in a procedural setting

In the BPM setting, the enactment is supported by a *process engine*, which takes in input a BP model and shows to the users the so-called *worklist*, representing the list of activities that must be currently executed. When a new activity is performed, the process engine logs the operation (extending the execution trace of the instance) and then moves one step forward, updating the worklist.

Enactment in a declarative setting

For ConDec, enactment must be adapted to the declarative nature of the approach; instead of showing to the interacting entities what to do next, the enactment of ConDec, schematized in Figure 61, is about

A. highlighting constraints that are temporarily unsatisfied (i.e., constraints that are waiting for the execution of some activity);

B. showing the forbidden, non-executable activities (i.e., activities that cannot be currently executed);

C. showing the enabled activities (i.e., activities that can be currently executed) – an activity is currently enabled iff it is not forbidden;

D. informing whether the execution can be currently terminate, i.e., whether the interacting entities can quit without breaking some mandatory constraint.

In other words, the aim of enactment is to inform the user about the current status of each constraint, as well as to *block* the activities whose execution would surely lead to violate the model (i.e., to violate one of its mandatory constraints) [157, 159].

EXAMPLE 14.1 (Enactment of a ConDec model). *Table 32 shows the enactment of a (fragment of a) ConDec model (taken from Figure 12). At the*

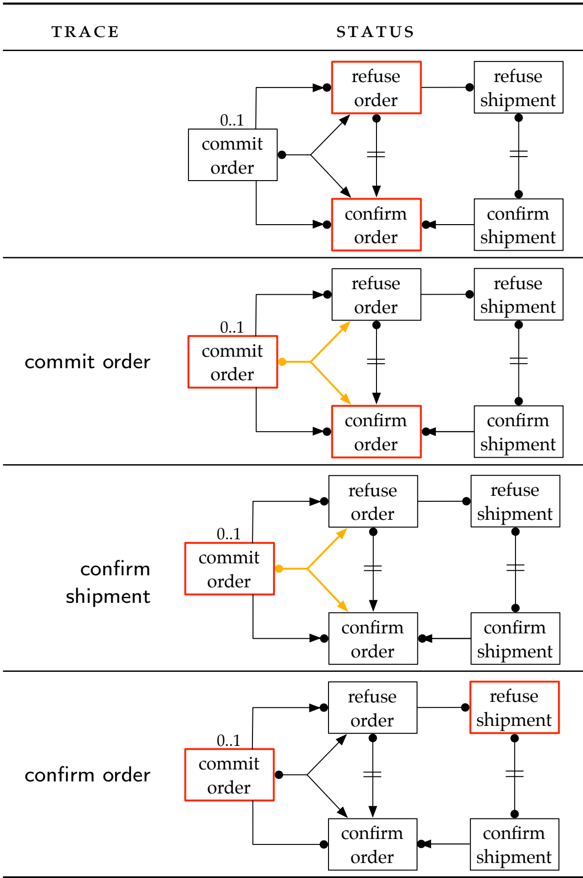


Table 32: Enactment of a ConDec model.

beginning, refuse order *and* confirm order *are blocked, because they need a previous* commit order*. Then, an order is committed; the effect is that*

- refuse order *is no more blocked;*
- confirm order *is blocked, because it needs a previous* confirm shipment*;*
- commit order *is blocked, due to the* `0..1` *cardinality constraint;*
- *the disjunctive response constraint is temporarily unsatisfied, waiting for an order confirmation or refusal.*

Now the shipment is confirmed, enabling the possibility of executing the order's confirmation and blocking shipment's refusal. Finally, the order is confirmed, and the response constraint binding commit order *with the two possible answers goes back to a satisfied state. The execution can then correctly terminate, because all constraints are satisfied, i.e., the execution trace* commit order $\longrightarrow$ confirm shipment $\longrightarrow$ confirm order *is compliant with the model.*

14.6.1 *Showing Temporarily Unsatisfied Constraints*

At each time, the partial execution trace collected so far can be used by REC to highlight temporarily unsatisfied constraints. In fact, the translation discussed in Section 14.5.1 can be seamlessly applied to mandatory constraints as well. The ConDec model under enactment can then be represented as a REC theory; using REC, the execution of the system can be monitored, extracting the current status of each constraint. Given a partial execution trace $\mathcal{T}_p$, the current time is represented by $ct(\mathcal{T}_p)$; if at time $ct(\mathcal{T}_p)$ a certain $sat(c)$ fluent is not holding, then c is currently temporarily unsatisfied.

14.6.2 *Blocking Unexecutable Activities*

REC can be exploited also to highlight the set of currently unexecutable activities, i.e., activities forbidden by some constraints. Given a partial execution trace $\mathcal{T}_p$, if at time $ct(\mathcal{T}_p)$ a certain $forb(a, c)$ fluent is holding, then activity a is currently unexecutable. Obviously, this does not mean that a is a dead activity: in the future it could be possible that it will become executable.

Speculative reasoning for enactment

However, not all the unexecutable activities are highlighted by exploiting REC. In fact, exactly as in the case of run-time verification, there could exist activities which are not explicitly forbidden, but are unexecutable as well. An unexecutable activity is an activity that, if executed at the current time, will surely bring the system, sooner or later, in a violated state (a state in which at least one mandatory constraint is violated). Therefore, to complete the set of unexecutable activities, speculative reasoning is needed.

In Section 13.4, speculative reasoning has been exploited to identify violations as soon as possible. This task has been accomplished by asking `g-sciff` about the possibility of completing the current partial execution trace so as to comply with the ConDec model. Now we exploit `g-sciff` to ask a slightly different question: whether, given the current partial execution trace, it is possible to extend it with a new occurrence of an activity a, s.t. there will be at least one way to continue the interaction without violating the model. If it is the case, then a is executable, otherwise it is not.

Discovery of non-executable activities

In the general case, given

- A ConDec model $\mathcal{CM}$, which has been suitably pre-processed before the execution[12]
- A partial trace $\mathcal{T}_p$ representing the evolution of the instance reached so far

the discovery of non-executable activities proceeds as follows:

1. By means of REC, obtain the set $\mathcal{U}^{expl}$ of activities made explicitly non executable by some constraint;

12 Using the pre-processing procedure described in Section 10.5, which aims at guaranteeing termination of `g-sciff` when reasoning upon the model.

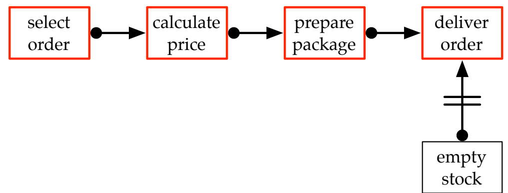


Figure 62: Enactment of the ConDec model shown in Figure 55, after the execution of activity empty stock.

2. For each activity a in $\mathcal{A}^{\mathcal{CM}}/\mathcal{U}^{expl}$
   a) Suppose that a is the next executed activity, i.e., that the new execution trace of the system will be
   $$\mathcal{T}^{a}_{next} = \mathcal{T}_p \cup \{\mathbf{H}(exec(a), t) \mid t = ct(\mathcal{T}_p) + 1\}$$
   b) Check if the hypothetical execution of activity a will bring the system to a state which will surely lead to violation – if it is the case, then a is added to the set of implicitly unexecutable activities $\mathcal{U}^{impl}$.
3. All the activities which do not belong to $\mathcal{U}^{expl} \cup \mathcal{U}^{impl}$ are enabled at time $ct(\mathcal{T}_p)$.

Point 2-b is accomplished by exploiting `g-sciff` in a semi-open setting[13] (i.e., by enabling the *times update* transition). This is needed because `g-sciff` must speculate only on the future: the evolution of the instance reached so far is completely determined by the happened events contained in the partial execution trace $\mathcal{T}_p$.

In particular, we check if `g-sciff` has a successful derivation starting from the CLIMB translation of $\mathcal{CM}$ and the execution trace $\mathcal{T}^{a}_{next}$. A positive answer means that a can be executed as the next activity; instead, a negative answer means that no possible extension of $\mathcal{T}^{a}_{next}$ exists s.t. all the mandatory constraints of $\mathcal{CM}$ are respected.

EXAMPLE 14.2 (Discovery of unexecutable activities). *Let us consider the ConDec model shown in Figure 55 – Page 242, after the execution of activity* empty stock, *say, at time* 1. *The resulting state is shown in Figure 62: all the other activities become unexecutable. The non-executability of* deliver order *is directly established as an effect of the negation response constraint which bind the two activities. For the other activities, non-executability is discovered by* `g-sciff`*'s speculative reasoning.*

For example, to verify whether the select order *activity is executable,* `g-sciff` *verifies the model against the hypothetical evolution in which the order is selected at time* 2, *i.e., against the execution trace*

$$\mathcal{T}^{select_order}_{next} = \{\mathbf{H}(exec(empty_stock), 1), \mathbf{H}(exec(select_order), 2)\}$$

Due to the presence of empty stock, `g-sciff` *generates a negative expectation concerning a future execution of* deliver order*:*

$$\mathbf{EN}(exec(deliver_order), T_d) \wedge T_d > 1$$

13 Semi-open reasoning has been described in Section 13.2.3.

The hypothesized execution of select order*, instead, triggers the sequence of response constraints, generating two intensional event occurrences:*

$$\mathbf{H}(\mathtt{exec}(\mathtt{calculate_price}), T_c) \wedge T_c > 2,$$
$$\mathbf{H}(\mathtt{exec}(\mathtt{prepare_package}), T_p) \wedge T_p > T_c.$$

The second generated occurrence triggers the last response constraint, expecting a future execution of the deliver order *activity:*

$$\mathbf{E}(\mathtt{exec}(\mathtt{deliver_order}), T_{d2}) \wedge T_{d2} > T_p$$

However, this positive expectation matches with the previously generated negative expectation concerning the same activity, thus leading to **E***-inconsistency. Activity* select order *cannot be therefore executed as the next activity. Non-executability of* calculate price *and* prepare package *is identified in a similar way.*

14.6.3 *Termination of the Execution*

The last task that must be accomplished to support enactment is checking whether, in a given state of affairs, the instance of the system can be terminated without violating some constraint. If the answer is yes, then the interacting entities have the possibility to decide whether to quit or continue the interaction; in the latter case, a new enactment cycle is triggered.

Given the current execution trace $\mathcal{T}_p$, related to an instance of the ConDec model $\mathcal{CM}$, the interacting entities can quit by respecting all the model's mandatory constraints iff $\mathcal{T}_p$ is a complete trace supported by $\mathcal{CM}$. This can be verified by checking if $\mathcal{T}_p$ complies with the constraints of $\mathcal{CM}$, which corresponds to evaluate whether sciff has at least one successful closed derivation for $\mathcal{CM}$ and $\mathcal{T}_p$.

Let us for example consider the enactment shown in Table 32, identifying the enacted ConDec model as $\mathcal{CM}$. At the beginning of the execution, the interacting entities can quit, because no temporarily unsatisfied constraint is contained in the model. If we focus on the underlying CLIMB formalization $\mathcal{S} = t_{CLIMB}(\mathcal{CM})$, the absence of a temporarily unsatisfied constraint means that there does not exists a pending positive expectation. The possibility to quit the execution is then attested by the fact that $\mathcal{S}_\emptyset \vdash^{\emptyset}_{\emptyset} \mathtt{true}$.

In the second and third state, the execution cannot be instead terminated without violating $\mathcal{CM}$, because the response constraint binding the commit order activity with refuse order or confirm order is temporarily unsatisfied, i.e., there is a pending expectation concerning refuse/confirm order which must be satisfied before quitting.

In the last state, the execution trace collected so far is the sequence commit order $\rightarrow$ confirm shipment $\rightarrow$ confirm order. Let us call such a trace $\mathcal{T}$. It holds that $\exists\ \Delta$ s.t. $\mathcal{S}_\emptyset \vdash^{\mathcal{T}}_{\Delta} \mathtt{true}$, and therefore the execution can correctly terminate at this stage.

15

DECLARATIVE PROCESS MINING

Contents

15.1 Grounding the Process Mining Framework: SCIFF Checker, DecMiner, ProM **277**
15.2 The SCIFF Checker ProM Plug-in **278**
15.2.1 CLIMB Textual Business Rules 279
15.2.2 A Methodology for Building Rules 280
15.2.3 Specification of Conditions 281
15.2.4 Compliance Verification with Logic Programming 282
15.2.5 Embedding SCIFF Checker in ProM 283
15.3 Case Studies **284**
15.3.1 The Think3 Case Study 285
15.3.2 Screening Guideline of the Emilia Romagna Region 288
15.3.3 Quality Assessment in Large Wastewater Treatment Plans 289
15.4 The DecMiner ProM Plug-in **291**
15.4.1 Inductive Logic Programming For Declarative Process Discovery 292
15.4.2 Embedding DecMiner Into the ProM Framework 293
15.5 The Checking-Discovery Cycle **294**

When an interaction system is executed, the relevant event occurrences involved in its instances are tracked and stored in an information system. The necessity of logging all the performed operations is twofold. On the one hand, external regulations, such as the Sarbanes-Oxley Act, require the presence of complete logs in order to audit the behaviour of the organization and assess its compliance. On the other hand, only by analyzing its own behaviour, an organization becomes aware of its own fallacies and bottlenecks and can improve itself.

All the modern Business Process Management (BPM) systems used in the industry provide logging facilities to track the evolution of its running instances. For example, Care-Flow Management Systems store all the made examinations, treatments, drug administrations, providing to health-care professionals a complete picture about the current status of each patient. Web Service-Oriented Systems provide SOAP interceptors to capture message exchanges and log them.

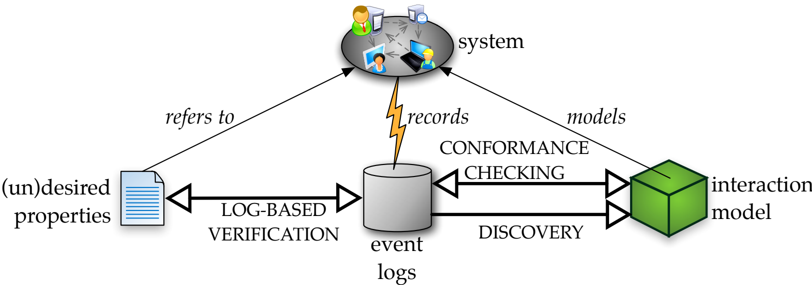


Figure 63: Some process mining techniques (from [197]).

The extraction of information from the stored logs is called *process mining*[1]. Differently from run-time verification and monitoring, process mining techniques are applied *a-posteriori*, i.e., on traces representing already completed instances of the system. Process mining can be considered as a special case of data mining, in which data are focused on the information regarding the dynamics of a system.

Process mining techniques

Figure 63 depicts three process mining techniques[197]:

LOG-BASED VERIFICATION analyzes the execution traces verifying whether the instances of the system meet certain desired properties or not.

DISCOVERY aims at extracting a new interaction model starting from the execution traces; the obtained model provides a feedback about the real behaviour exhibited by the interacting entities.

CONFORMANCE CHECKING compares the execution traces with a pre-established model, in order to assess the discrepancy between the prescribed and the actual behaviour.

Process mining and industries

In the last years, process mining is drawing the attention of industry. Companies are interested in applying process mining techniques on their own event logs mainly because:

- The extraction of information from past execution traces is of key importance, to help business analysts in decision making and to support the revision and improvement of Business Processes.
- Process mining centers around the event logs, which reflect the real, concrete behaviour exhibited by the company when running its Business Processes.
- Process mining techniques rely only on the presence of event logs, and are therefore independent from the specific BPM system adopted by the company; companies can therefore benefit from the outcome of process mining without needing to touch their software infrastructure.

1 http://www.processmining.org

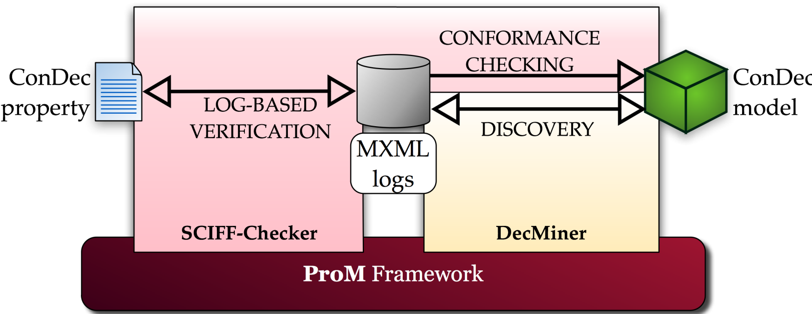


Figure 64: Grounding of the process mining general schema on declarative technologies.

In this chapter, we focus on *declarative process mining*, i.e., on techniques which exploit declarative languages (ConDec^{++} and CLIMB in particular – see Chapters 3, 4 and 6) and corresponding reasoning frameworks in order to extract information from event logs. In particular, we present two software tools that we have developed in this setting:

- *SCIFF Checker* is an analysis ProM [190] plug-in whose purpose is to classify events as compliant or non-compliant w.r.t. declarative business rules, expressed in a textual pseudo-natural language which resembles the CLIMB language and is therefore able to express all the ConDec^{++} constraints. The tool provides different template rules whose activity types, originators and execution times can be constrained and specialized by the user through a GUI.

SCIFF Checker plug-in

- *DecMiner* is a ProM plug-in which implements the mining algorithm described in [118, 117]. Its purpose is to discover a declarative constraint-based specification starting from a set of MXML execution traces, previously labeled as compliant or not.

The DecMiner plug-in

We report the experience made on some case studies, in which declarative process mining techniques have been effectively applied on real execution traces.

15.1 GROUNDING THE PROCESS MINING FRAMEWORK: SCIFF CHECKER, DECMINER, PROM

Figure 64 illustrates how we have grounded the general schema shown in Figure 63. We adopt ConDec^{++}/CLIMB as modeling languages. As we have discussed in Chapters 8 and 10, ConDec models and CLIMB specifications can be used to seamlessly express interaction models, regulatory models and properties. Therefore, log-based verification and conformance checking collapse in a single task.

As an underlying infrastructure, our techniques rely on the ProM framework. ProM is an open source framework (under the Common

The ProM framework

Public License, CPL) for process mining[2]; it is plug-able, i.e., people can plug-in new pieces of functionality. Beside a plethora of mining techniques, ProM offers a wide range of plug-ins related to model transformations and model analysis (e.g., verification of soundness, analysis of deadlocks, invariants, reductions, etc.). ProM currently contains about 200 plug-ins and is in continuous evolution.

A prominent advantage of ProM is that it defines an universal format for representing event logs: the MXML[199] format. MXML is an extensible XML-based meta model thought for standardizing the way event logs are represented. It is able to accommodate a wide-range of execution traces, produced in different settings[3]. A third party must only provide a conversion mechanism which produces an MXML representation from its own event logs, and it becomes able to exploit all the functionalities supported by ProM.

ProMimport

The ProMimport framework[4] can support third parties in such a conversion process; it provides more than 20 pre-defined import filters, and it can be easily extended. MXML has been described in Section 6.2.1 – Page 114 – of this dissertation.

15.2 THE SCIFF CHECKER PROM PLUG-IN

SCIFF Checker [48] is a ProM plug-in aimed at dealing with compliance checking of execution traces w.r.t. reactive business rules. Such business rules could express external regulations, internal policies or business trends, and it is important for business analysts to be able to verify if the company is behaving as expected. Such a demand has grown in the last years, which have seen the outbreak of financial scandals and the corresponding increase of strict regulations, but also the evolution of BPM towards flexibility.

Among the different kind of flexibility (see Section 2.2.2), flexibility by change and by deviation [160, 165] enable the possibility of changing the process instance or deviating from the prescribed model during execution, making therefore impossible to assess compliance *before* the execution, as we have done in Chapter 8. As claimed in [195] "deviations from the 'normal process' may be desiderable but may also point to inefficiencies or even fraud", and therefore flexibility could lead the organization to miss its strategic goals or even to violate regulations and governance directives. Furthermore, as the complexity of BP models increases, it becomes important to provide support for a business analyst in the task of analyzing past executions. This analysis can help the business manager in assessing business trends and consequently making strategic decisions.

2 ProM can be downloaded from `http://www.processmining.org`

3 To have an idea about the number of case studies in which ProM has been applied, just take a look at `http://prom.win.tue.nl/research/wiki/publications/overview`

4 `http://promimport.sourceforge.net`

```
Rule ::= [IF Body THEN] Head
Body ::= Activity_Exec
         [AND Activity_Exec]* [AND Constraints]
Activity_Exec ::= S_Activity | R_Activity
S_Activity ::= activity A_ID is performed
               [by O_ID] [at time O_T]
R_Activity ::= activity A_IDis performed N times
               [by O_ID][between time O_T and time O_T]
Head ::= Head_Disjunct [OR Head_Disjunct]*
Head_Disjunct ::= Activity_Exp
                  [AND Activity_Exp]* [AND Constraints]
Activity_Exp ::= S_Activity_Exp|R_Activity_Exp
S_Activity_Exp ::= activity A_ID should [not] be performed
                   [by O_ID] [at time O_T]
R_Activity_Exp ::= activity A_IDshould be performed N times
                   [by O_ID] [between time O_T and time O_T]
```

Figure 65: An excerpt of the CLIMB textual rules grammar.

15.2.1 *CLIMB Textual Business Rules*

To specify business rules, we introduce a textual pseudo-natural notation, which has the same expressiveness of CLIMB (and can be straightforwardly mapped to it), and can therefore accommodate all the ConDec^{++} constraints. Such a notation is readable and can be easily customized by a non-IT savvy. The structure of rules therefore resembles ECA (Event-Condition-Action) rules [161]; the main difference w.r.t. ECA rules is that, since CLIMB rules are used for checking, they envisage expectations about executions rather than actions to be executed.

An excerpt of the grammar describing the syntax of such textual rules is reported in Figure 65. As in CLIMB, rules follow an `IF` ***`Body`*** `having` ***`BodyConditions`*** `THEN` Head structure, where Body is a conjunction of occurred events, with zero or more associated conditions BodyConditions, and Head is a disjunction of positive and negative expectations (or *false*). Each head element can be subject to conditions as well.

Events and the MXML meta model

The concept of event is tailored to the one of entry in the MXML meta model [199]. Events are atomic and mainly characterized by:

- the name of the *activity* it is associated to;
- an *event type*, according to the MXML transactional model [199], which captures the life cycle of each activity with event types like “start", “re-assignment", “completion";
- an *originator*, identifying the worker who generated the event;

- an *execution time*, representing the time at which the event has been generated.

As far as now, other data are not taken into account, but they could be seamlessly introduced.

Variable parameters

The main distinctive feature of our rules is that all these parameters are treated, by default, as variables. To specify that a generic activity *A* has been subject to a whatsoever event, the rule body will simply contain a string like: `activity A is performed by` O_A `at time` T_A, where A stands for the activity's name, O_A and T_A represent the involved originator and execution time respectively, and *performed* is a keyword denoting any event type. To facilitate readibility, the part concerning originator and execution time can be omitted if the corresponding variables are not involved in any condition.

Such a generic sentence will match with any kind of event, because all the involved variables (A, O_A and T_A) are completely free, and the event type is not specified. The sentence can then be configured in many different ways. In particular, the involved variables can be grounded to specific values or constrained by means of explicit conditions. The event type can be instead fixed by simply substituting the generic *performed* keyword with one of the specific types envisaged in the MXML transactional model.

Positive (negative) expectations are represented similarly to occurred events, by only changing the `is` part with `should (not) be`.

15.2.2 *A Methodology for Building Rules*

To clarify our methodology, let us consider a completely configured rule, expressing that if a person executes a check activity, then the same person cannot execute the publish activity:

```
  IF activity A is performed by O_A
     having A equal to check
THEN activity B should NOT be performed by O_B
     having B equal to publish and O_B equal to O_A
```
(R3-Think3)

By analyzing this rule, we can easily recognize two different aspects: on the one hand, the rule contains generic elements, free variables and constraints, whereas on the other hand it specifically refers to concrete activities. The former aspect captures re-usable patterns – in this case, the fact that the same person cannot perform two different activities A and B, which is known as the *four-eyes principle*. The latter aspect instantiates the rules in a specific domain, grounding A and B to two activity names. To reflect such a separation, we foresee a three-step methodology to build, configure and apply business rules (see Figure 66):

1. A set of re-usable rules, called *rule templates*, are developed and organized into groups by a technical expert (i.e., someone having

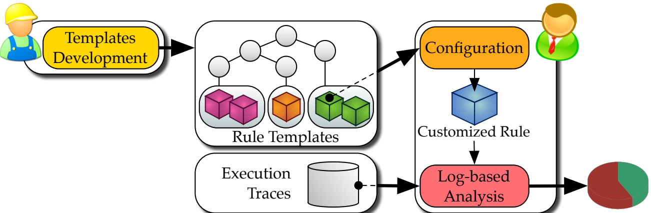


Figure 66: A methodology for building, configuring and applying business rules.

a deep knowledge of rules syntax and semantics). It is worth noting that, being the textual notation of rules based on the CLIMB one, it is more expressive than ConDec^{++}. For For example, the following re-usable template represents a *synchronized response*, i.e. a response triggered by the occurrence of two events:

IF activity A is performed at time T_A
and activity B is performed at time T_B
THEN activity C should be performed at time T_C
having T_C after T_A and T_C after T_B.

2. Rule templates are further configured, constrained and customized by a business manager to deal with her specific requirements and needs. Here, conditions are exploited to ground the variables involved in the templates (see the next section).

3. Configured rules are exploited to perform compliance checking of company's execution traces.

15.2.3 *Specification of Conditions*

Conditions are exploited to constrain variables associated to event occurences and expectations inside business rules (namely activity names, originators and execution times). As shown in Figure 67 two main families of conditions are currently envisaged: string and time conditions. String conditions are used to constrain an activity/originator by specifying that it is equal to or different from another activity/originator, either variable or constant. An example of a string condition constraining two originator variables is the "O_B equal to O_A" part in rule (R3-Think3).

Time conditions are used instead to relate execution times, in particular for specifying ordering among events or imposing quantitative constraints, such as deadlines and delays. The semantics of constraints is determined by time operators, which intuitively capture basic time relationships (such as before or at). Absolute time conditions constrain a time variable w.r.t. a certain time/date, whereas relative time conditions define orderings and constraints between two variables. Relative conditions can optionally attach a displacement to the target

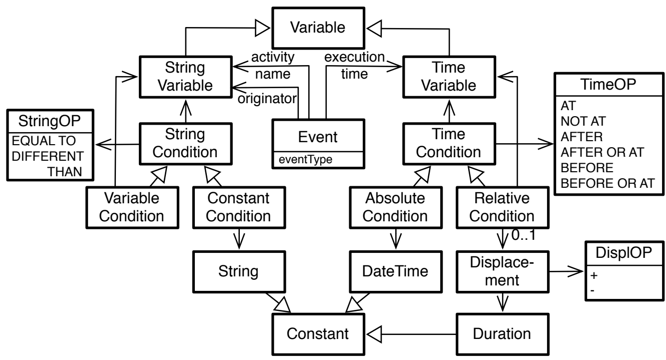


Figure 67: Basic hierarchy of string and time constraints.

time variable as well. For example, to specify that the time variable T_B must be within a *displacement* of 2 days *after* T_A, we simply write T_B BEFORE $T_A+2days$.

15.2.4 *Compliance Verification with Logic Programming*

Verification by sciff

The reasoning phase, which aims at verifying whether a set of execution traces comply with a given rule, can be tackled by sciff. In this case, the verification procedure is as follows. For each MXML execution trace T belonging to the log under study:

A. translate T to a set of CLIMB happened events, forming a CLIMB execution trace $\mathcal{T}$;

B. express the textual rule in the CLIMB syntax, obtaining an IC BR;

C. if $\exists\ \Delta$ s.t. $\langle\emptyset, \{BR\}\rangle_{\emptyset} \vdash^{\mathcal{T}}_{\Delta}$ $true$, then the execution trace is compliant with the rule.

Verification by Prolog

However, it is worth noting that, being the analysis carried out a-posteriori, the reactive nature of sciff is not exploited. We therefore asked ourselves if, in this specific case, sciff could be substituted with a classical backward resolution algorithm, such as the one of Prolog. The answer is yes, as we briefly show below.

An execution trace is treated by Prolog as a knowledge base storing each audit trail entry as a fact of the type

$$h(exec\,(EventType, ActivityName, Originator)\,, ExecutionTime))).$$

The rule used for checking is instead transformed into a Prolog query by computing the negation of the implication represented by the CLIMB textual rule. So, if the CLIMB rule is represented by the implication $B \rightarrow H$, then the query would be $B \wedge \neg H$. Such a query tries to find a set of occurred events in the execution trace that satisfy the rule's body but violate its head. For example, rule (R3-Think3) is translated to the following query:

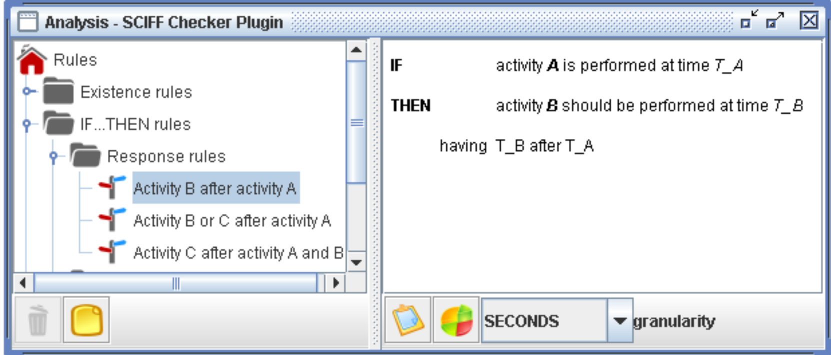


Figure 68: A screenshot of the main SCIFF Checker window.

$$?\!-\!A = \texttt{'Check'}, h(exec(_, A, O_A), T_A),$$
$$not(B = \texttt{'Publish'}, O_B \neq O_A, not(h(exec(_, B, O_B), T_B))).$$

Since the analysis is performed a-posteriori, positive expectations are flattened to occurred events, and negative expectations to the absence of events. If the query succeeds, then a counter-example which violates the rule has been found in the execution trace. The trace is then evaluated as non-compliant.

15.2.5 *Embedding SCIFF Checker in ProM*

Drawing inspiration from the *LTL Checker* ProM plug-in [195], we have implemented SCIFF Checker as a ProM analysis plug-in. The implementation supports the three-steps methodology described in Section 15.2.2, providing a user-friendly GUI for the customization of rule templates.

At start-up, templates are loaded from an XML-based template file. As far as now, all the templates covering the ConDec language have been specified using the textual notation. In order to extend or modify the template hierarchy, the technical expert has simply to change this file.

As shown in Figure 68, templates are displayed exploiting a tree-like component; clicking on a template description causes its corresponding textual representation to appear in the center panel. By clicking on a "configuration" button, the different variables and customizable elements of the rule become highlighted. When selecting an highlighted element, a specific customization panel appears, supporting the user in setting the parameters (such as event types and repetitions) and in the specification of conditions.

When the chosen rule has been customized, it can be either saved to a special group containing all the user-defined rules, or directly used for compliance checking. In the latter case, the user has first to choose a *granularity*, which ranges from milliseconds to months and defines the time unit for converting time quantities into integer values. For each

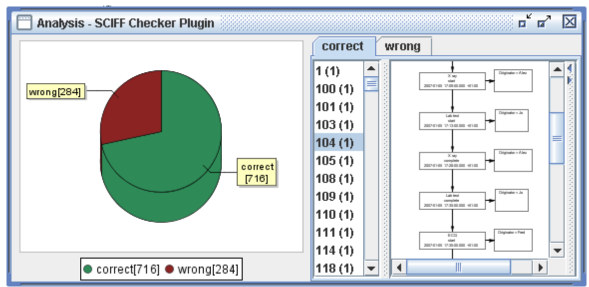


Figure 69: Compliance chart produced by SCIFF Checker at the end of verification.

execution trace contained in the considered MXML log, three steps are then performed transparently to the user:

A. the execution trace is translated into a Prolog knowledge base, converting involved execution times; *t*he textual business rule is mapped to a Prolog query, following the proof-of-concept shown in Section 15.2.4;

B. a Prolog engine based on SWI[5] is exploited to verify whether the execution trace complies with the specified rule.

All verification outcomes are finally collected and a summarizing pie chart is shown, together with the explicit list of compliant/non compliant traces. The user can then start a new classification by considering the whole log or only the (non) compliant execution traces. In this way, a conjunction of business rules can be verified by performing a sequence of tests, each one dealing with a single rule, and selecting at each step only the compliant subset for the next verification.

15.3 CASE STUDIES

In this section, we describe three real case studies in which we have applied SCIFF Checker. The first case study involves the verification of business rules on traces generated by the customers of Think3®[6], a company working in the Computer Aided Design (CAD) and Product Life-cycle Management (PLM) market. The second case study aims at verifying compliance of health-care professionals with a screening process applied by the Emilia Romagna region of Italy. The last case study deals with an intelligent framework which aims at automatically monitoring and assessing the quality of wastewater treatment plants; SCIFF Checker is used in this context to identify the occurrence of high-level events in the plant, starting from signal-level events.

5 http://www.swi-prolog.com/
6 http://www.think3.com

15.3.1 *The Think3 Case Study*

An important current challenge in the manufactoring industry is to handle, verify and distribute the technical information produced by the design, development and production processes of the company. The adoption of a system supporting the management of technical data and the coordination of the people involved is of key importance, to improve productivity and competitiveness. The main issue is to provide solutions for managing all the technical information and documentation (such as CAD projects, test results, photos, revisions), which is mainly produced by workers during the design phase. Since an important part of the design process is spent by testing, modifying and improving previously released versions, the *traceability* of relevant information concerning an item is necessary.

Think3 is one of the leading global players in the field of CAD and PLM solutions: it provides an integrated software which bridges the gap between CAD modeling environments and other tools involved in the process of designing (and then manufacturing) products. All these tools are transparently combined with a non-intrusive information system which handles the underlying product workflow, recording all the relevant information and making it easily accessible to the workers involved, enabling its consultation, use and modification. Such an information system supplies a detailed, shared and constantly updated vision of the life cycle of each product, providing a complete log of the executed activities.

The underlying Think3 workflow centers around the design of a manufacturing product. Different activities can be executed to affect the progress-status of an item, involving the modification and even the evolution of multiple co-existing versions of its corresponding project. Such a workflow can be adapted on each single Think3 client company in order to meet different specific requirements.

Think3 Workflow

Figure 70 shows the basic Think3 workflow used to handle the design of a manufacturing product. It describes how different activities affects the progress-status of an item, supporting the modification and even the evolution of multiple co-existing versions of its corresponding project. Such a workflow can be adapted on each single Think3 client company in order to meet different specific requirements. The process starts with the creation of a new project, which is associated to a unique identifier. The project becomes now *in progress*: different designers work cooperatively on it by developing CAD drawings, manuals and other documentation. In this phase, a vaulting system is exploited to handle accesses and changes to resources [183]: read-accesses can be performed concurrently, whereas modifications are mutually-exclusive. When a complete draft of the project has been designed, it becomes *published* and ready to be *checked* by a technical reviewer (in the meanwhile, if someone realizes that further details are needed, the project can be carried back to the previous state by

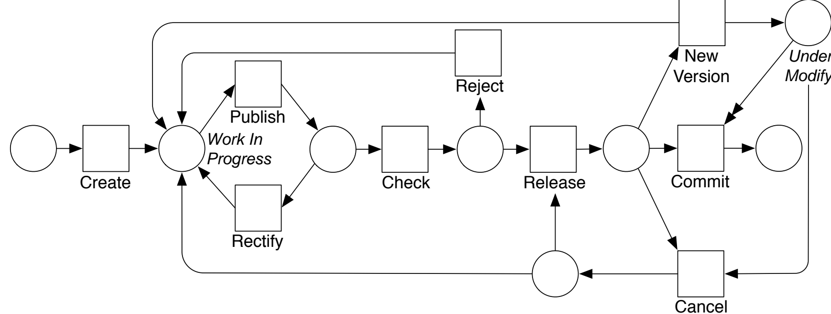


Figure 70: Basic Think3 workflow for the managament of manufacturing products.

executing the *rectify* activity). After having been checked, a decision is made: if all the performed tests have been successufully passed, the project can be *released*, otherwise it will be *rejected* asking for changes, carrying it back to the *work in progress* state. The *release* activity attests that a completed and checked version of the project actually exists. At this point, there are three possibility:

- Technical experts decide[7] to save the current project and to generate a *new version* of it, triggering another design-and-test iteration; if this is the case, two different versions co-exists: the older one is put in a *under modify* state, to attest that a new version is currently being processed, whereas the newer one is put *in progress*.
- If at least one previous version of the project exists, it is possible to *cancel* the current one; as a consequence, one of the previous versions is retrieved from the *under modify* state.
- The project can be finally *committed*, i.e. passed to the manufacturing phase; this choice leads to removing all the previous versions of the project (the information system will anyway store them in order to satisfy the traceability requirement).

Think3 Requirements

To support a business analyst in decision making, and in particular in the tasks of analyzing the life cycle of different projects and pinpointing problems and bottlenecks, Think3 is investigating the development of a Business Intelligence dashboard. Within the TOCAI.IT FIRB Project[8], Think3 and the University of Bologna are collaborating to realize one of the main dashboard components: a tool supporting compliance verification of design processes w.r.t. configurable business rules. This will facilitate the analysts in the identification of behavioural trends and non-compliances to regulations or internal

7 This decision could have different motivations, one can be the desire to optimize some specific aspects of the project.
8 http://www.dis.uniroma1.it/~tocai/

policies. In this particular case study, we elicitated the following non-exaustive list of interesting properties:

(Br1) Evaluating the time relationship between the execution of two given activities (e.g. *Was a project committed by 18 days after its creation?*).

(Br2) Identifying which projects passed too many times through a certain activity (e.g., *Which projects have been modified at least twice?*).

(Br3) Analysing originators, i.e., workers involved in the process (e.g., *Was a project checked by a person different than the one who published it?*).

Such properties can be easily specified by using the CLIMB textual notation. For example, Rule Br3 can be represented as shown in (R3-Think3) – Page 280.

Rule Br1 can be instead modeled by grounding the general response pattern:

```
  IF activity A is performed at time T_A
THEN activity B should be performed at time T_B
     and T_B after T_A.
```

The pattern is adapted on Think3's Rule Br1 by grounding the involved activities to the ones cited in Rule Br1 and by adding a *before* relative time constraint having a 18 days-displacement.

```
    IF activity A is performed at time T_A
having A equal to Creation
  THEN activity B should be performed at time T_B
having B equal to Commit
       and T_B after T_A and T_B before T_A + 18days.
```

Applying SCIFF Checker to the Think3 Case Study

SCIFF Checker has been concretely applied to analyze the execution traces of a Think3 client. We have first exploited the ProMimport tool in order to convert the relevant information from the client database into an MXML format, by considering the project name as case identifier.

In particular, we extracted a portion of 9000 execution traces, ranging from 4 to 15 events. Then we used, together with a Think3 business manager, the plug-in to express and test the business rules of interest. The average time for performing compliance checking has been assessed to be around 10-12 seconds. The verification outcomes have been finally analyzed with the business manager. For example, considering Rules Br2 and Br3, we discovered that, fortunately, only 2% of the execution traces involved more than two project revisions, and

that in 3,5% of the cases only the same person was responsible for both publishing and checking the project.

The verification of rules like Rule Br1 was found interesting especially by varying the deadline involved. Indeed, the business analyst wanted to detect projects taking too much time as well as projects released too soon, to point out both possible bottlenecks and potential inaccuracies.

15.3.2 *Screening Guideline of the Emilia Romagna Region*

The Emilia Romagna Region proposes several screening programs for the early detection and treatment of cancer. In particular, we have applied SCIFF Checker on the Cervical Cancer Screening Guideline proposed by the sanitary organization of the Emilia Romagna region of Italy[9].

Cervical cancer screening guideline

Cervical cancer is a disease in which malignant (cancer) cells grow in the tissues of the cervix. The screening program proposes several tests in order to early detect and treat cervical cancer, and it is usually organized in six phases: Screening Planning, Invitation Management, First Level Test (PAP-test), Second Level Test (Colposcopy), Third Level Test (Biopsy), and Therapeutic Treatment. Every participant is asked to repeat the screening process periodically (a round being thus three years long).

The need for compliance verification in the clinical setting

The careflow modeled by the Cervical Cancer Screening Guideline is quite complex and involves more than 50 activities performed by 15 different health care professionals and structures. Furthermore, it is applied on a huge number of persons at the same time, and involves a plethora of different activities ranging from administrative procedures (such as the emission of a letter containing the result of a PAP-test) to clinical and therapeutic actions (such as the analysis of a biological sample by anatomo-pathology laboratories). Providing suitable techniques for analyzing how the screening process is being applied on the territory is therefore of key importance, to assess the quality from the patient's perspective, as well as to check the compliance of health-care professionals with the prescribed recommendations and to evaluate the amount of undertaken resources. The detection of non-compliant situations helps to better understand how the ideal Clinical Guideline is being effectively applied in the concrete setting, providing the basic information needed for taking proper countermeasures or for revising the Clinical Guideline, so as to better fit with the specific peculiarities of the Emilia Romagna region.

Formalization of the cancer screening process

To this aim, we have formalized a fragment of the screening process, starting from the recommendations provided by the sanitary organization of the Emilia Romagna region [143, 46]. The obtained model consists of a set of rules, constraining the behaviour of administrative personnel and of health-care professionals. Rules covered many different aspects, ranging from normative aspects (such as the emission of a letter after the PAP-test has been completed) to quality aspects (it

9 http://www.regione.emilia-romagna.it/screening/

is desirable that a PAP-test takes place no later than 15 days after the scheduled date).

Then, we have extracted a set of 1950 careflow executions from the sanitary information system. In order to fully test our tools, some wrong behaviours have been introduced in this database. Each screening round has been checked as a single interaction (hence we did not check the conformance for the repetition of the screening rounds). Each screening contains several events, up to the maximum of 18 (the whole careflow).

Results

We first tested the event logs abstracting away from quality aspects, i.e., taking into account only the strict rules that should be followed by the health-care professionals so as to comply with the recommendations of the regional Clinical Guideline. At a first run of the trace analysis process, 1091 executions resulted to be non-compliant. These results were analyzed by a screening expert which confirmed all the compliant classifications and proposed some changes to the careflow model: in fact, some traces classified as compliant by the domain expert were instead considered as non-compliant w.r.t. the initial model. The major discrepancy between the formalized model and the real world concerned the initial phase of the screening process, in which an invitation is sent to a patient, who is then committed to accept or refuse the invitation. In many concrete cases, however, the patient simply decided to not answer at all: such a behaviour is not an indicator of non-compliance from the viewpoint of the sanitary organization.

We have therefore revised the model to reflect also this possibility. In this way, we avoided false non-compliant classifications, reducing the number of "wrong" executions to 44: this result agreed with the "wrong behaviour" executions we artificially introduced.

Verification of quality aspects

Finally, we have incorporated also quality aspects, such as the desired requirement stating that if a patient has been invited to attend the PAP-test, the exam should take place no more than 15 days after the scheduled exam. We repeated the analysis of the logs, and we discovered that 200 times the PAP-test has been attended more than 15 days later w.r.t. the initial schedule. The delay could be explained by the fact that the screening center allocates in advance a certain number of slots: as a consequence, free slots for new booking are not immediately available.

15.3.3 *Quality Assessment in Large Wastewater Treatment Plans*

Large wastewater treatment plants are usually equipped with Software Control And Data Acquisition (SCADA) and / or Programmable Logic Controller (PLC) systems, but the vast amount of data acquired from the sensors is seldom actually used in real-time plant management. An intelligent control software, instead, could act as a "virtual operator", monitoring the processes continuously. First of all, it could try to optimize the yield and detect faults at an early stage, possibly even correcting them. Then, the collected time series, properly validated and classified, could be used to build a knowledge base describing the var-

Figure 71: Architecture of an intelligent monitoring and quality assessment framework for wastewater treatment plants (from [123]).

ious operating conditions of a given plant: this knowledge could be used to further improve the overall performance of the plant.

An intelligent monitoring and quality assessment architecture

Figure 71 shows the overall architecture of an intelligent monitoring and quality assessment framework for wastewater treatment plants, that is currently being realized by the University of Bologna and ENEA, as part of a collaboration with Hera, one of the leading italian groups in Waste, Gas and Electricity Business [180].

Signal-level events

In the first stage, a multi-parameter converter continuously measures physical signals (e.g., pH values) during the whole treatment process: the samples are acquired and stored in a MySQL database for both immediate and later access. Then, a neural network is exploited to analyze each obtained signal, to the aim of identifying "low-level" events of interest (called signal-level events), such as the presence of an apex, step or knee in the signal.

Mapping signal-level events to MXML

Starting from these signal-level events, SCIFF Checker is then employed to identify process-level events, such as the start or the completion of a reaction in the plant. The first step needed to properly integrate SCIFF Checker within the architecture concerned the mapping of low-level event sequences describing the signals to MXML. In particular, in [123] the following correspondence has been established:

- the low-level event associated to the signal, i.e., $\mathsf{step}^{\{+/-\}}$, $\mathsf{apex}^{\{+/-\}}$, $\mathsf{knee}^{\{+/-\}}$;
- the physical signal, to which the sequence of low-level events refer to (e.g., pH), represents the originator of the event;
- the activity execution time represents the time-stamp at which the low-level event has been detected by the neural network.

Identification of process-level events with SCIFF Checker

Starting from this correspondence, the conditions which support the recognition of an higher level event can be defined using a CLIMB textual business rule. An higher level event occurs when one or more of the monitored signals present certain characteristics at the same time. In particular, by combining three signals, namely pH, ORP (redox potential) and DO (dissolved oxygen concentration), three rules have been specified:

- completion of a denitrification (pH apex^{+} and ORP knee^{-});
- completion of a nitrification (pH apex^{-} and ORP knee^{+} and DO step^{+});
- aerobic switch (pH knee^{-} and ORP apex^{-} and DO step^{+}).

For example, the CLIMB textual representation of the completion of a denitrification is:

```
  IF activity A is started by O_a
     having A equal to cycle
THEN activity B should be performed by O_b at time T_b
     and activity C should be performed by O_c at time T_c
     having B equal to apex+ and O_b equal to pH
     and C equal to knee- and O_c equal to ORP
     and T_c after T_b - 15min and T_c before T_b + 15min
```

The rule states that a maximum in pH and a knee in ORP must be detected almost contemporarily. The 30 minutes window has been introduced to deal with noise due to the measures themselves and the interpolation error.

Process validation

In order to assess the performance of the plant, a process instance, a cycle modelled by a sequence of signal-level events, is submitted to a rule, which either accepts or rejects it: a set of instances, then, is partitioned into two subsets, the "correct" ones and the "wrong" ones. Each one of these subsets can be further processed by other rules, effectively applying them in cascade. Therefore, starting from the denitrification, nitrification and aerobic switch rules, different combinations of the rules can be applied to define an acceptance policy for the cycle. Even if a cycle is discarded, it is not necessarily a failed one: the occurrence of a false negative is a likely event since, in addition to errors of the algorithm itself, probes tend to yield very noisy measurements (especially if not cleaned daily) which may not satisfy the required conditions. In order to overcome this problem, it is possible to define less constraining rules by removing or changing some of the conditions in the basic ones. This can be simply done through the SCIFF Checker GUI.

15.4 THE DECMINER PROM PLUG-IN

Process discovery deals with the extraction of a model starting from the real execution traces of a system. The extracted model is of key importance, because it explains the real behaviour exhibited by the interacting entities.

Traditional process discovery approaches extract procedural BP models from the execution traces, such as Petri Nets or Event-driven Process Chains [188, 198]. However, following the motivations discussed in the first part of this dissertation (see Section 2.2.1, in particular Figure 3 at Page 16), it would be desirable to discover declarative specifications as well. In fact, declarative languages such as ConDec are easily understandable by business analysts and are not affected by procedural details.

ILP for declarative process discovery

DecMiner is one of the few effective tools which aims at *declarative process discovery*. It relies on Inductive Logic Programming (ILP) techniques and, given as input a set of process execution traces, previously

labeled as compliant or not w.r.t. a certain criterion, is able to produce a SCIFF specification which correctly "replays" the classification. By exploiting the inverse ConDec^{++} - CLIMB translation, the produced result can be finally rendered as a graphical ConDec^{++} model, for the sake of readability.

15.4.1 *Inductive Logic Programming For Declarative Process Discovery*

The idea of exploiting ILP for declarative process discovery comes form the similarities between learning a SCIFF theory, composed by a set of ICs, and learning a clausal theory as described in the learning from interpretation setting of ILP [148]. Besides the fact that both SCIFF and clausal theories can be used to classify a set of atoms (i.e., an interpretation) as positive or negative, they have strong similarities in the structure of the logical formula composing the theory[118].

The DPML algorithm

In particular, Lamma et al. have adapted the Inductive Constraint Logic (ICL) algorithm [148], in order to learn SCIFF theories starting from a set of execution traces, labeled as compliant or not [118]. The new algorithm is called Declarative Process Model Learner (DPML). Inductive algorithms rely on a clausal language $\mathcal{L}$, specifying which kind of logical theories can be produced by applying the algorithm. The clausal language is usually described in an intensional way using a specific representation language. The description of $\mathcal{L}$ in this language is called *language bias* ($\mathcal{LB}$). In the DecMiner setting, the $\mathcal{LB}$ represents the structure of SCIFF ICs that can be discovered.

The target of DPML is to find a SCIFF specification $\mathcal{S}$ which "replays" the classification given as input, i.e.:

- for each execution trace $\mathcal{T}^+$ initially classified as compliant, it holds that COMPLIANT $(\mathcal{S}_{\mathcal{T}^+})$;
- for each execution trace $\mathcal{T}^-$ initially classified as non-compliant, it holds that $\neg$COMPLIANT $(\mathcal{S}_{\mathcal{T}^-})$.

To accomplish this task, DPML performs a covering loop in which non-compliant execution traces are progressively ruled out. At each step, DPML looks for the most general IC that correctly classifies as many as possible compliant execution traces and, at the same time, rules out as many as possible non-compliant execution traces.

Tuning DPML to learn ConDec^{++} models

DPML is able to learn ICs specified using the SCIFF-full language. In the context of DecMiner, we are interested in obtaining, at the end of the computation, a SCIFF specification which can be graphically rendered as a ConDec^{++}model. In this respect, in [47, 117] Chesani et al. have fixed the $\mathcal{LB}$ to reflect exactly the structure of ICs which are obtained by translating ConDec^{++} (i.e., the ICs syntax is restricted to a sub-set of the CLIMB language). The translation has been presented in the Chapters 5 and 6 of this dissertation. Since DPML is now tuned to learn only such a restricted set of ICs, it is possible to perform an inverse translation and obtain the corresponding graphical representation.

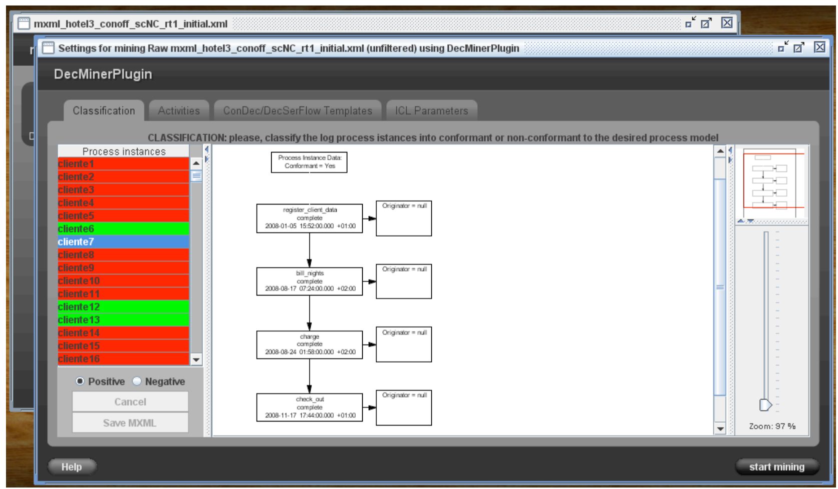


Figure 72: DecMiner plug-in: trace classification.

15.4.2 *Embedding DecMiner Into the ProM Framework*

The DPML algorithm has been wrapped into a ProM plug-in which supports the user in all the process discovery phases, from data preparation to visualization of the learned ConDec model[10].

Support for the manual tuning of the language bias

In particular, the tuned $\mathcal{LB}$ is automatically generated from a set of general templates, one for each ConDec constraint, that can be instantiated to generate specific assertions for the language bias. The number of all possible assertions can be huge, while the user could be interested to models defined only by a small, yet meaningful set of ConDec constraints. For this reason, we let the user free to select a subset of the activities contained in the execution traces, and a subset of the ConDec constraints. Then, our approach uses only the instantiation of these constraints with the selected activities for learning the model. Besides providing as output a model that fits the user requirements, smaller constraint sets allow also better performances of the learning algorithm.

Complexity of the approach w.r.t. the language bias

The accuracy and learning time depends on the choice of these subsets. They influence the accuracy of the learned model because an activity relation discriminating between compliant and non-compliant execution traces cannot be learned if the appropriate template and/or activities were not chosen. The time complexity is linear in the number of traces and in the number of constraints. With respect to the number of activities, it is quadratic if there are binary constraints, and linear if there are only unary constraints.

Discovery phases

DecMiner implements all the data preparation and learning phases of the discovery process described above and guides the user by means of its GUI. The phases of such a process are:

10 Visualization is currently limited to the basic ConDec language, but the algorithm is seamlessly able to deal with ConDec^{++}.

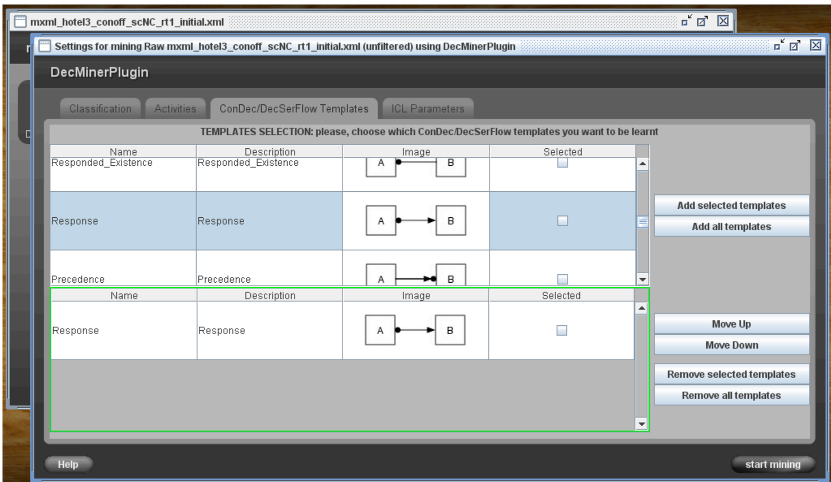


Figure 73: DecMiner plug-in: ConDec template selection.

CLASSIFICATION The user exploits the GUI shown in Figure 72 to browse the execution traces and label some of them as compliant (positive) or not compliant (negative). Such a phase could be automatically accomplished by using SCIFF Checker as a pre-processing step (see Section 15.5).

ACTIVITIES The user can choose among all the activities and their associated parameters the information that she considers important for learning the declarative model.

TEMPLATES The user exploits the GUI shown in Figure 73 to choose the set of existence, relation and negation ConDec templates to be used in the discovery phase.

MINING When the *start mining* button is pressed, DPML is used to discover the declarative model. Before starting DPML, the language bias is generated, by instantiating the chosen templates with the chosen activities.

RESULTS The learned SCIFF/CLIMB rules, together with the corresponding ConDec model, are presented to the user[11].

15.5 THE CHECKING-DISCOVERY CYCLE

An interesting integration between SCIFF Checker and DecMiner, which we plan to better investigate in the near future, concerns the possibility of realizing the *discovery-checking* cycle shown in Figure 74.

SCIFF Checker to DecMiner

The use of SCIFF Checker as a pre-processing step for DecMiner comes into help for dealing with one of the most critical phases of the declarative discovery process, namely the labeling of each given execution trace as compliant or not. Quite often, the execution traces

11 The ConDec model is shown by using the DECLARE tool: `http://declare.sf.net`

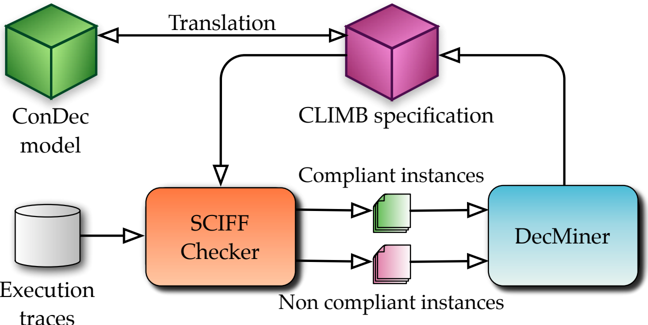


Figure 74: Integration of SCIFF Checker and DecMiner, realizing a checking-discovery cycle.

taken as input do not contain any hint about being positive or negative. Therefore, the splitting of traces into compliant and non-compliant subsets must be manually carried out. SCIFF Checker facilitates this task: instead of explicitly labeling each single execution trace, the user can state a classification criterion and automatically obtain the desired splitting. It is worth noting that integration is already implemented: SCIFF Checker provides as output an annotated log which can be directly imported into DecMiner. If the language bias is opportunely tuned, the model learned by DecMiner could be noticeably different from the adopted criterion, and hence interpreted as an *explanation* for the classification.

DecMiner to SCIFF Checker

The integration between the two plug-ins could be exploited also in the opposite way. In this respect, a discovered ConDec model can be interpreted as a regulatory model expressing the "real world" behaviour, and used to check whether new instances of the system maintain or break such a behaviour.

16

RELATED WORK AND SUMMARY

Contents

16.1 Related Work **297**
16.1.1 Run-Time Verification and Monitoring 297
16.1.2 Enactment 299
16.1.3 Log-Based Verification 300
16.1.4 Discovery 301
16.2 Summary of the Part **302**

In this Chapter, related work concerning the enactment, run-time verification&monitoring and mining of interaction models is presented. Then, the major contributions of this part of the dissertation are briefly summarized.

16.1 RELATED WORK

We organize related work separating the run-time and the a-posteriori aspects:

- run-time verification and monitoring;
- enactment;
- log-based verification;
- discovery.

16.1.1 *Run-Time Verification and Monitoring*

The importance of run-time verification and monitoring has been raised by many researchers, especially in the field of Service-Oriented Computing. Here, monitoring is addressed with several approaches: business rules [119], WS-BPEL [18], WS-Agreement [125], to cite some. Advanced conformance checking techniques described in [170] are used in [193] and implemented in the ProM framework [190]; this approach has been applied to SOAP messages generated from Oracle BPEL.

Many authors propose LTL as a suitable language for specifying requirements and properties to be checked at run-time[20, 86]. However, the construction of suitable monitoring facilities has a twofold impact:

- The semantics of LTL must be modified to reflect that, during the execution, reasoning cannot be definitive, but must be "open" to the acquisition of new event occurrences (see the discussion of Section 13.1). Let us for example consider a partial, ongoing instance of the system, which does not contain the execution of activity pay. A liveness property such as $\Diamond pay$ should not evaluate such an instance as wrong: there is still the possibility to meet the property, by executing the pay activity in the future. To deal with this issue, Bauer et al. propose to switch to a three-valued semantics for LTL [20]; in the example, such a semantics would evaluate $\Diamond pay$ as unknown. In [86], instead, Giannakopoulou and Havelund introduce a finite-trace semantics for LTL, reflecting the fact that partial ongoing executions are finite; their semantics would state that $\Diamond pay$ is violated, but in their setting the concept of violation corresponds to a temporary violation limited to the observed portion of execution.

- The corresponding automata generation algorithms must be modified ad-hoc, to reflect the new semantics and accomplish the monitoring task accordingly; ad-hoc methods makes it very difficult to prove their formal properties, such as soundness and completeness.

The run-time and monitoring infrastructure presented in this paper differs from all the cited approaches because:

- It relies on one declarative language (with one declarative semantics), that is employed as it is in all the life cycle of the targeted models, spanning from design and static verification to run-time verification, monitoring, enactment and mining.

- Reasoning is addressed by combining the `sciff` and `g-sciff` proof procedures, for which the formal properties of soundness, completeness and termination have been studied.

- By accommodating a reactive form of Event Calculus (EC), enhanced features such as monitoring optional constraints and handling compensation mechanisms can be seamlessly introduced inside the framework, at a pure declarative level (i.e., without revising the underlying reasoning techniques).

The first two points differentiate our proposal also from the monitoring frameworks which rely on the EC, because they use ad-hoc methods, outside of logical frameworks. For example, Mahbub and Spanoudakis present a framework [128] for monitoring the compliance of a WS-BPEL service composition w.r.t. behavioral properties automatically extracted from the composition process, or assumptions/requirements expressed by the user. EC is exploited to monitor the actual behavior of interacting services and to report different kind of violations. The approach is pursued in [129], where an extension of WS-Agreement is used to specify requirements. The monitoring framework relies on an ad-hoc event processing algorithm, which fetches the occurring events and updates the status of the affected fluents.

While only a few proposals have been focused on the adoption of EC as a monitoring framework, there is a vast literature on the use of EC before or after the execution. Rouached et al. propose a framework for engineering and verifying WS-BPEL processes is [168]. EC is used to provide an underlying semantics to WS-BPEL, enabling verification before and after execution. In particular, EC is exploited to verify consistency and safety of a service composition (i.e. to statically check if the specification always guarantees the desired requirements), and to check whether an already completed execution has deviated from the prescribed requirements. The authors rely on an inductive theorem prover for the verification task. Thanks to the Reactive Event Calculus (REC) provided in Chapter 14, the mapping of WS-BPEL presented in [168] could be exploited to monitor WS-BEL processes with the `sciff` proof procedure.

In [15], Aydin and colleagues use the Abductive EC to synthesize a web service composition starting from a goal. The composition process is described as a planning problem, where the functionalities provided by the individual services are (atomic) actions, requiring some inputs and producing certain outputs. Being `sciff` an abductive proof-procedure, it would be interesting to investigate the possibility of adopting the REC to deal also with this issue.

16.1.2 *Enactment*

The enactment of ConDec models has been addressed by Pesic and van der Aalst in [158, 157]. They rely on the formalization of ConDec as an LTL conjunction formula (see Section 3.7 – Page 51). Such a formula is translated to an automaton using the approach proposed in [86]. Enactment is then directly supported by starting from the initial state of the obtained automaton, triggering a transition and moving to a corresponding successor state every time a new activity is executed. Differently from the approach presented in this dissertation, the automaton carries directly, by construction, the complete information about which activities are enabled or forbidden in a certain state. Therefore, speculative reasoning is not needed to properly support the enactment, eliminating a computational step which, in our case, must be executed each time a new activity is executed. On the other side, as we have shown in Chapter 11, the construction of the automation is a highly expensive process, exponential in the size of the formula. While timing aspects are not critical in this context (the automaton is built before the execution, and therefore the interacting entities do not experience delays during the execution), it is important to remember that also the space needed to store the automaton is exponential in the size of the formula, making it difficult to handle medium-sized models.

Enactment of declarative service specifications, specified by means of Concurrent Transaction Logic (CTR), has been addressed by Roman and Kifer in [167]. However, their notion of enactment is different than ours: while we focus on supporting the interacting entities during the

execution, showing the status of constraints and identifying the enabled and forbidden activities, they focus on the static synthesis of *one possible* way to enact the service under study, guaranteeing conformance with a choreography. This is done by combining the CTR specifications of the service and the choreography and finding a constructive proof for the global specification.

16.1.3 *Log-Based Verification*

A huge amount of work has been (and is being) carried out in order to deal with flexibility and adaptivity in Process Aware Information Systems [201]. DECLARE [160] is a constraint-based WfMS which adopt ConDec as a declarative graphical specification language. Flexibility is tackled at design-time, supporting the user in the specification of a minimal set of constraints that should be met during execution rather than focusing on a specific procedural solution, and at run-time, supporting the dynamic removal and insertion of constraints. In [165] ADEPT2 is proposed as a Workflow Management System capable to support the change of running instances (*flexibility by change*). The authors also suggest that Adaptive Process Management System should store, besides the enactment log, also the log of sequences of changes applied to a process model during execution.

SCIFF Checker could be considered as complementary to these systems: it can be used to assess whether executed instances met the desired requirements/regulations. An interesting future work concerns the verification of process logs containing also the sequence of changes; in this setting, it would be possible to express requirements involving also such changes, and to investigate the relationship between non-compliances and changes.

The closest work to SCIFF Checker is the ProM LTL-Checker [195], that shares with our approach motivation and purposes. While LTL-Checker exploits LTL for the formalization of properties, our approach belongs to the Logic Programming family. The LTL-Checker uses a variant of LTL which supports some kind of data (such as for example absolute time conditions), and exploits a verification technique which does not rely on the translation of the LTL formula to an automaton, thus avoiding the state-explosion problem. Anyway, the textual rules employed in the SCIFF Checker are more expressive than the LTL-Checker formulas, being they e.g. able to express relative metric time constraints (delays and deadlines). Furthermore, the configuration of templates inside the *LTL-Checker* mainly consists of associating ground values to the available parameters, while SCIFF Checker supports the specification of many different conditions on the variables, enabling the possibility of modeling a variety of business rules starting from a single template.

As discussed in Section 12.1.2, there are many proposals which aims at assessing compliance with regulations and rules before the execution. For example, in [172] the compliance of processes to regulations and standards is enforced by design rather than being checked a pos-

teriori. Obviously, a static approach is not suited to deal with flexibility by change or deviation. Furthermore, it cannot deal with situations where the outcome of the compliance evaluation inherently depends on the actual configuration (resources and data) of the instance, e.g. like in the case of the *four-eyes principle*, where the focus is about the actual originators.

16.1.4 *Discovery*

Process discovery is an active research area. Among the many proposals present in the literature, we cite [1, 188, 198, 91, 68, 75]. In [1], Agrawal et al. introduce the idea of applying process mining to workflow management. The authors propose an approach for inducing a process representation in the form of a directed graph encoding the precedence relationships. In [188], van der Aalst et al. present the α-algorithm for inducing Petri nets from data, identifying a class of models for which the approach is guaranteed to work. The α-algorithm is based on the discovery of binary relations in the log, such as the "follows" relation. In [198] van Dongen and van der Aalst describe an algorithm which derives causal dependencies between activities and use them for constructing instance graphs, presented in terms of Event-driven Process Chains. [91] is a recent work where a process model is induced in the form of a disjunction of special graphs called workflow schemas.

DecMiner differs from these works because it adopts a representation that is declarative rather than procedural, without sacrificing expressiveness. Moreover, the learning process starts from compliant and non-compliant traces, rather than from compliant traces only.

The works presented in [68, 75] are closer to DecMiner because they deal with discovering (partially) declarative specifications. In [68], the learning process starts from *runs*, which are Petri Nets representing high level specifications of a set of process traces. Mining is performed by merging the different runs for the same process. The model that is obtained is hybrid, in the sense that it may contain sets of activities that must be executed but for which no specific order is required. We differ from this work because we start from traces rather than runs: while runs specify already a partial order among activities, carrying out significant information about the system, traces are simply event sequences denoting the execution of activities.

A critical requirement of DecMiner is that each input trace must be previously classified as compliant or non-compliant. SCIFF Checker comes into help: the user must only provide a classification criterion, and then the two sub-sets are automatically computed. However, finding a classification criterion is not always a reasonable assumption, because it requires that the user already has a deep knowledge about the domain under study. In order to avoid asking the user to classify activities, in [87] Goedertier proposes an approach for automatically generating negative events, i.e., events that are used as negative exam-

ples. The integration of this approach with DecMiner is therefore an interesting line of research.

16.2 SUMMARY OF THE PART

In this part of the thesis, we have applied the CLIMB framework for reasoning upon the execution traces of a system both at run-time and a-posteriori. We have described how the reactive nature of `sciff` can be exploited to accomplish the run-time verification task: `sciff` is able to dynamically acquire the occuring events of an instance, checking on-the-fly whether they comply with a prescriptive ConDec model or not. We have pointed out that `sciff` is not always able to detect a violation as soon as possible, and we have proposed a solution to overcoming this issue. The solution interleaves a `sciff` reasoning phase with a speculative reasoning step in which `g-sciff` is employed to verify whether at least one way to continue the interaction by respecting the ConDec model actually exists.

We have then discussed the rigid nature of run-time verification, which stops itself as soon as the first violation is encountered. We have then introduced a more flexible monitoring framework, able to explicitly capture and reify violations. Such an extended framework relies on a reactive axiomatization of the Event Calculus (REC) on top of `sciff`; beside discussing the axiomatization, we have also proven that, under the assumption that events occur in ascending order, it is irrevocable, i.e., it monitors the execution by never retracting the provided information. REC has been exploited to monitor optional ConDec constraints, supporting the possibility of reifying violations and handling corresponding compensation mechanisms, again expressed with the ConDec language. The combination of `sciff`, `g-sciff` and REC has then be used to handle the enactment of ConDec models, providing support to the interacting entities by showing, instant by instant, the enabled as well as the forbidden activities, preventing the possibility of performing activities that would surely lead to violate the model.

The application of CLIMB in the context of (declarative) process mining has given birth to two concrete tools, which have been implemented as part of the well-known ProM framework. The first tool, called SCIFF Checker, aims at performing log-based verification, checking if the execution traces of a system comply with a given business rule, specified using a pseudo-natural language which resembles the CLIMB one. We have explained the architecture of the tool and discussed its application on three real case studies, belonging to different settings (Business Process Management, Clinical Guidelines, Wastewater Treatment Plants). The second tool, called DecMiner, is dedicated to discover a new ConDec model starting from a set of execution traces, previously labeled as compliant or not. The obtained model provides an high-level, declarative description of the real behaviour of a system, abstracting away from procedural details. The two tools can be combined to realize a checking-discovery cycle, where SCIFF Checker could be employed as a pre-processing step for preparing data to Con-

Dec, and the discovered model could be adopted to assess compliance of new execution traces.

Part IV

CONCLUSIONS AND FUTURE WORK

17

CONCLUSIONS AND FUTURE WORK

This dissertation contains at least one error.
— Marco Montali

Contents

17.1 Conclusions 307
17.2 Future Work 309

17.1 CONCLUSIONS

We have put forwards significant evidence to support our thesis:

> *Declarativeness* and *openess* are needed to deal with different emerging settings, where systems are composed by several autonomous entities which collaborate and *interact* to the aim of achieving complex strategic goals, impossible to be accomplished on their own. Computational Logic is a suitable framework to support the entire life cycle of such systems, ranging from their specification and static verification to their execution, monitoring and analysis.

In particular, we have proposed the CLIMB integrated framework to deal with all this issues. The framework is composed by:

- ConDec, a graphical language proposed by Pesic and van der Aalst to specify interaction models with an open, declarative flavor [157, 158].
- The CLIMB language, a subset of the SCIFF language [7] which exhibits some interesting properties (such as compositionality); the language targets the specification of interaction by means of Integrity Constraints that must be respected by the interacting entities, and is equipped with a clear declarative semantics, providing a meaning to the notion of compliance.
- `g-sciff`, an abductive proof procedure able to statically verify CLIMB specifications by adopting a generative approach.
- `sciff`, an abductive proof procedure able to verify, at run-time or a-posteriori, whether an execution trace describing the (partial) evolution of the system complies with a CLIMB specification.

- REC, a reactive encoding of the Event Calculus [114] on top of `sciff`.

In the first part of the dissertation, we have pointed out that many different challenging settings, such as Business Process Management, Clinical Guidelines, Service Oriented and Multi-Agent Systems, require the adoption of open and declarative approaches. We have proposed the conjunct use of the ConDec notation and the CLIMB language to overcome the limits of classical, procedural closed approaches. We have synthesized a translation procedure able to take as input an arbitrary ConDec model, automatically producing a corresponding CLIMB formalization. We have carried out an extensive theoretical comparison with the framework of propositional Linear Temporal Logic (LTL), which has been already used to formalize the ConDec constructs [157]. The comparison showed that the two proposed formalization are equivalent w.r.t. the notion of compliance of an execution trace with a ConDec model, and that CLIMB is strictly more expressive than LTL. Differently from LTL, CLIMB is a first-order language, which supports variables and metric temporal constraints; these added capabilities have been exploited to extend to ConDec with new features, maintaining a valid CLIMB mapping.

In the second part of the dissertation, we have discussed the issue of static verification of ConDec models, illustrating different verification tasks, such as existential and universal entailment of properties, a-priori compliance verification and issues related to the composition of many local specification to realize a global choreographic model. We have demonstrated that each kind of verification can be reduced to existential entailment, which in turn can be successfully treated by the `g-sciff` proof procedure. We have criticized `g-sciff`, pointing out that there exists ConDec model for which its termination cannot be guaranteed. We have then proposed a method to overcome this problem in the specific case of ConDec. The possibility of using model checking techniques to address the static verification of ConDec models led us to carry out an extensive quantitative evaluation of `g-sciff`, comparing it with state of the art explicit and symbolic model checkers. The comparison support our claims and motivate us to pursue this line of research.

In the third part of the thesis, we have applied the CLIMB framework to provide support at run-time and a-posteriori. We have exploited the reactive nature of `sciff` to accomplish the run-time verification task. We have criticized `sciff`, pointing out that there exists cases in which it is not able to detect a violation as soon as possible. To overcome this problem, we have proposed an integration between `sciff` and `g-sciff`, where the first is used to reason about the present and about the past, and the second is used to speculatively reason about the future. We have then pointed out that REC can be fruitfully introduced to monitor the status of optional ConDec constraints during the execution, making it possible to reify the detected violations as special event occurrences. The application of ConDec constraints to such special events has been the basis to augment the monitoring process with the con-

cept of compensation to a violation. The combination of (speculative) run-time verification and monitoring has then been exploited to enact ConDec models, preventing the possibility of performing activities that would surely lead to violate the model.

Finally, the CLIMB framework has been applied to build declarative process mining tools, which have been concretely implemented on top of ProM [190], one of the most popular frameworks in the field of process mining. The first tool, called SCIFF Checker, classifies a set of execution traces as compliant or non-compliant with a business rule, specified in a pseudo-natural language which resembles the CLIMB language. Beside the description of the main functionalities of the tool, we have reported its application on three real industrial case studies, demonstrating its versatility. The second tool, called DecMiner, is one of the first attempts to discover a declarative specification starting from a set of execution traces. DecMiner starts from execution traces previously labeled as compliant or non-compliant w.r.t. a certain user criterion, and relies on Inductive Logic Programming techniques to induce a SCIFF specification which correctly reproduces the classification. By properly restricting the language bias of the learning algorithm, the structure of the induced SCIFF specifications can be set so as to cover exactly the CLIMB rules that we have previously used to formalize ConDec. In this way, an inverse translation can be applied, obtaining a graphical ConDec model as a result.

In conclusion, we have shown how the integration of languages and techniques coming from different mainstreams (Business Process Management, Multi-Agent Systems, Knowledge Representation and Reasoning) can be fruitfully exploited to specify and reason about declarative open interaction models, covering both theoretical as well as practical aspects.

We hope that we have convinced the reader that Computational Logic, and in particular extensions of Logic programming, are a significant approach, in that they provide both effective technologies to carry out reasoning, but at the same time provides the possibility of defining and proving that such technologies meet certain fundamental formal properties, such as soundness and completeness.

We hope also that this work will stimulate new researcher to combine proposals and approaches coming from different research areas, contributing to their mutual cross-fertilization.

17.2 FUTURE WORK

Beside the need of further experimentation and application of the proposed techniques to other industrial case studies, this dissertation leaves many open challenges and issues for future research. We briefly discuss some of them in the following.

Termination of Static Verification and ConDec Extensibility

In [159], Pesic et al. state that ConDec is an extensible language: a new constraint can be added in the DECLARE tool, by deciding its graphical representation and providing its corresponding LTL formalization. In our setting, extending the set of constraints could affect the termination of `g-sciff`, making the pre-processing algorithm presented in this dissertation incomplete: new kind of loops could be built by composing the new constraints in subtle ways. To be sure that the pre-processing algorithm still guarantees that all looping situations can be identified, we should rewrite the CLIMB representation of the new constraint so as to make it "forward" or "backward" (see Section 10.4.2 – Page 185). However, this is a non-trivial task, nor applicable in the general case.

The only way to guarantee the termination of `g-sciff` for an arbitrary specification would be the incorporation of a general method for dealing with infinite computations. A promising approach to handle infinite computations during verification seems to be Coinductive Logic Programming [95], because Coinductive LP extends the usual operational semantics of logic programming to allow reasoning over infinite and cyclic structures and properties. We will therefore investigate the possibility of augmenting `g-sciff` with Coinductive LP techniques in the future.

Development of an Editor and Enactment Prototype

An ongoing work concerns the development of an editor for the extended version of ConDec. The main purpose of this editor will be to support the enactment of extended ConDec models by exploiting the CLIMB framework. Indeed, the combination of the reasoning techniques proposed to provide enactment support, has not yet been implemented in a concrete prototype.

It is worth noting that such a prototype should include two different translation procedures:

- one from ConDec to the CLIMB language, for performing compliance verification about the past and speculative reasoning about the future;
- one from ConDec to REC, for tracking the status of each constraint.

An interesting possibility to realize these two translations, would be to adopt Model Transformation techniques. Given:

- a source meta-model (in our case, the ConDec meta-model)
- a target meta-model (in our case, the structure of the CLIMB language and of the REC ontology)
- a set of rules translating each entity of the source meta-model to the target meta-model (in our case, the translation procedure which maps ConDec constraints to CLIMB rules, and the REC fluent-based representation of ConDec's constraints)

Model Transformation techniques automatically apply the general rule on a specific source model, producing the corresponding target model. For example, ATL [104] is a Model Transformation Language which supports the specification of translation rules, supposing that both the source and the target meta-models are expressed in EMF[1], the core modeling framework used inside the well-known Eclipse framework[2].

The ongoing implementation of the editor is being carried out inside Eclipse, and therefore we will investigate the use of ATL for reducing the translation procedures presented in this dissertation to a Model Transformation problem.

Contracting, Discovery and Interoperability in the Semantic Web

The SCIFF framework and the `sciff` and `g-sciff` proof procedures, have been recently exploited in the context of Semantic Web (Services), to deal with discovery/contracting [6, 49] and interoperability issues [2]. An open ongoing research activity concerns the comparison between the verification techniques described in such papers, and the ones addressed in this dissertation.

Integration Between ConDec and Social Commitments

The Event Calculus has been exploited by Yolum and Singh to formalize social commitments [206][3]. Social commitments are a powerful framework for capturing the mutual obligations between interacting entities during the execution; they are even more flexible than ConDec because they provide the possibility of describing a desired state of affairs, leaving the interacting entities free to exploit their opportunities and dynamically find a sequence of activities leading to such a state; they provide support for compensation mechanisms and exception handling [130]. Thanks to the translation presented in [206], we can seamlessly exploit our REC to monitor commitment-based interaction systems, relying on the CLIMB framework for the integration between ConDec and commitments. Our preliminary studies pursuing this line of research [45, 50] show the potentialities of such an integrated framework: commitments are used to capture mutual obligations and state of affairs, while ConDec is exploited to impose constraints on the activities that can be executed to reach these state of affairs. The semantics of the integrated model, as well as the impact on the underlying verification techniques, needs further investigation and experimentation.

Integration Between Declarative/Open and Procedural/Closed Approaches

In Section 2.5, we have discussed the non-trivial interplay between the declarative, open background knowledge of the heath-care professionals, and the procedural, closed prescriptions of the Clinical Guidelines that describe how assist patients in specific clinical circumstances. The

1 http://www.eclipse.org/modeling/emf/
2 http://www.eclipse.org/
3 Related work concerning social commitments has been discussed in Section 7.2.2 - Page 127.

work of Bottrighi et al. [32] is one of the first attempt to integrate these two kind of knowledge in order to obtain a hybrid, semi-open system. We plan to pursue this challenging line of research in the next future.

BIBLIOGRAPHY

[1] R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *6th International Conference on Extending Database Technology (EDBT1998)*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–483. Springer Verlag, 1998. (Cited on page 301.)

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and M. Montali. An Abductive Framework for A-Priori Verification of Web Services. In A. Bossi and M. J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 39–50. ACM Press, 2006. (Cited on pages 228, 229, and 311.)

[3] M. Alberti, F. Chesani, E. Lamma, M. Gavanelli, P. Mello, M. Montali, S. Storari, and P. Torroni. Computational Logic for the Run-time Verification of Web Service Choreographies: Exploiting the SOCS-SI Tool. In M. Bravetti, M. Nùñez, and G. Zavattaro, editors, *Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM'06)*, volume 4184 of *Lecture Notes in Computer Science*, pages 58–72. Springer Verlag, 2006. (Cited on page 235.)

[4] M. Alberti, M. Gavanelli, E. Lamma, F. Chesani, P. Mello, and P. Torroni. Compliance Verification of Agent interaction: a Logic-Based Software Tool. *Applied Artificial Intelligence*, 20(2-4):133–157, 2006. (Cited on page 240.)

[5] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, G. Sartor, and P. Torroni. Mapping Deontic Operators to Abductive Expectations. *Computational and Mathematical Organization Theory*, 12(2-3):205–225, 2006. (Cited on pages 127 and 226.)

[6] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Web Service contracting: Specification and Reasoning with SCIFF. In E. Franconi, M. Kifer, and W. May, editors, *Proceedings of the 4th European Semantic Web Conference (ESWC'07)*, volume 4519 of *Lecture Notes in Artificial Intelligence*, pages 68–83. Springer Verlag, 2007. (Cited on pages 21 and 311.)

[7] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable Agent Interaction in Abductive Logic Programming: the SCIFF framework. *ACM Transactions on Computational Logic*, 9(4), 2008. (Cited on pages 2, 3, 32, 53, 54, 66, 67, 154, 157, 163, 165, 235, 255, and 307.)

[8] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, Paola Mello, Marco Montali, and Paolo Torroni. Expressing and Verifying Contracts with Abductive Logic Programming. *Electronic Commerce, Special Issue on Contract Architectures and Languages*, 12(4): 9–38, 2008. (Cited on pages 226 and 235.)

[9] R. Alur and T. A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 104:35–77, 1993. (Cited on pages 219 and 222.)

[10] R. Alur and T. A. Henzinger. A Really Temporal Logic. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 164–169. IEEE Computer Society, 1989. (Cited on page 222.)

[11] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, and W. Yi. UPPAAL: Now, Next, and Future. In *Modeling and Verification of Parallel Processes*, pages 99–124. Springer Verlag, 2001. (Cited on page 222.)

[12] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003. (Cited on pages 15, 21, and 226.)

[13] A. Artikis, J. Pitt, and M. J. Sergot. Animated Specifications of Computational Societies. In *Proceedings of the The First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS2002)*, pages 1053–1061, 2002. (Cited on page 127.)

[14] A. Awad, G. Decker, and M. Weske. Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *6th International Conference on Business Process Management (BPM 2008)*, volume 5240 of *Lecture Notes in Computer Science*, pages 326–341. Springer Verlag, 2008. (Cited on page 226.)

[15] O. Aydin, N. K. Cicekli, and I. Cicekli. Automated Web Services Composition with Event Calculus. In *Proceedings of the 8th International Workshop in "Engineering Societies in the Agents World" (ESAW07)*, 2007. (Cited on page 299.)

[16] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *International Workshop on Web Services and Formal Methods (WS-FM 2005)*, volume 3670 of *Lecture Notes in Computer Science*, pages 257–271. Springer Verlag, 2005. (Cited on page 24.)

[17] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Verification of Protocol Conformance and Agent Interoperability. In Toni F and P. Torroni, editors, *Proceedings of the 6th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA2005)*, volume 3900 of *Lecture Notes in Computer Science*, pages 265–283. Springer Verlag, 2006. (Cited on pages 228 and 229.)

[18] L. Baresi, C. Ghezzi, and S. Guinea. Smart Monitors for Composed Services. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC '04)*, pages 193–202. ACM Press, 2004. (Cited on page 297.)

[19] A. Barros, M. Dumas, and P. Oaks. A Critical Overview of the Web Services Choreography Description Language (WS-CDL). *BPTrends*, 2005. (Cited on pages 2 and 21.)

[20] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of Real-Time Properties. In *Proceedings of the 26th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2006)*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006. (Cited on pages 297 and 298.)

[21] B. Bauer, M. Cossentino, S. Cranefield, M. P. Huget, K. Kearney, R. Levy, M. Nodine, J. Odell, R. Cervenka, P. Turci, and H. Zhu. The FIPA Agent UML Web Site, 2007 (last update). URL `http://www.auml.org/`. (Cited on page 24.)

[22] Tom Belwood, Luc Clément, David Ehnebuske, Andrew Hately, Maryann Hondo, Yin Leng Husband, Karsten Januszewski, Sam Lee, Barbara McKee, Joel Munter, and Claus von Riegen. UDDI Version 3.0. http://uddi.org/pubs/uddi_v3.htm, 2000. (Cited on page 19.)

[23] B. Benatallah, F. Casati, and F. Toumani. Analysis and Management of Web Service Protocols. In *Proceedings of the 23rd International Conference on Conceptual Modeling (ER 2004)* , volume 3288 of *Lecture Notes in Computer Science*, pages 524–541. Springer Verlag, 2004. (Cited on page 126.)

[24] B. Benatallah, F. Casati, and F. Toumani. Representing, Analysing and Managing Web Service Protocols. *Data and Knowledge Engineering*, 58(3):327–357, 2006. (Cited on page 228.)

[25] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Verlag, 2003. (Cited on page 222.)

[26] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Service Composition Based on Behavioral Descriptions. *Cooperative Information Systems*, 14(4):333–376, 2005. (Cited on page 227.)

[27] D. Beyer, A. Chakrabarti, and T.A. Henzinger. Web Service Interfaces. In *Proceedings of the 14th international World Wide Web Conference (WWW2005)*, pages 148–159, 2005. (Cited on page 228.)

[28] D. Bianculli, A. Morzenti, M. Pradela, P. San Pietro, and P. Spoletini. Trio2Promela: a Model Checker for Temporal Metric Specifications. In *Proceedings of the 20th International Conference on Software Engineering (ICSE2007)*, pages 61–62. IEEE Computer Society, 2007. (Cited on page 223.)

[29] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:118–149, 2003. (Cited on page 194.)

[30] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In M.C. Shan, U. Dayal, and M. Hsu, editors, *Proceedings of the 5th International Workshop on Technologies for E-Services (TES 2004)*, pages 15–28, 2004. (Cited on page 228.)

[31] F. Bosi and M. Milano. Enhancing CLP branch and bound techniques for scheduling problems. *Software Practice & Experience*, 31(1):17–42, 2001. (Cited on page 58.)

[32] A. Bottrighi, F. Chesani, P. Mello, G. Molino, Marco Montali, Stefania Montani, Sergio Storari, Paolo Terenziani, and Mauro Torchio. An Hybrid Approach to Clinical Guideline Conformance. In *12th Conference on Artificial Intelligence in Medicine (AIME'09)*, 2009 (sibmitted). (Cited on pages xxiv, 25, 28, and 312.)

[33] A. Bottrighi, F. Chesani, P. Mello, M. Montali, S. Montani, S. Storari, and P. Terenziani. Analysis of the GLARE and GPROVE Approaches to Clinical Guidelines. In *12th Conference on Artificial Intelligence in Medicine (AIME'09)*, 2009 (submitted). (Cited on pages xxi and 27.)

[34] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/soap, 2000. (Cited on page 19.)

[35] I. Bratko. *Prolog Programming for Artificial Intelligence*. Pearson Education. Addison–Wesley, 3rd edition, 2001. (Cited on page 53.)

[36] R. Bringhurst. *The Elements of Typographic Style*. Version 2.5. Hartley & Marks, Publishers, 2002. (Cited on page 335.)

[37] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. (Cited on page 211.)

[38] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*, pages 403–410. ACM Press, 2003. (Cited on page 228.)

[39] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 428–439, Washington, DC, USA, 1990. IEEE Computer Society. (Cited on page 211.)

[40] H. J. Bürckert. A Resolution Principle for Constrained Logics. *Artificial Intelligence*, 66:235–271, 1994. (Cited on page 154.)

[41] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and Orchestration Conformance for System Design. In P. Ciancarini and H. Wiklicky, editors, *Proceedings of the 8th International Conference on Coordination Models and Languages (COORDINATION 2006)*, volume 4038 of *Lecture Notes in Computer Science*, pages 63–81. Springer Verlag, 2006. (Cited on page 228.)

[42] G. Casella and V. Mascardi. West2East: exploiting WEb Service Technologies to Engineer Agent-based SofTware. *Agent-Oriented Software Engineering*, 1:396–434, 2007. (Cited on page 224.)

[43] C. Castelfranchi. Commitments: From individual intentions to groups and organizations. In V. R. Lesser Les. Gasser, editor, *Proceedings of the First International Conference on Multiagent Systems (ICMAS1995)*, pages 41–48. The MIT Press, 1995. (Cited on pages 2 and 127.)

[44] F. Chesani. *Specification, Execution and Verification of Interaction Protocols: an Approach based on Computational Logic*. PhD thesis, University of Bologna, 2007. (Cited on pages 53, 66, and 67.)

[45] F. Chesani, P. Mello, M. Montali, and S. Storari. Agent Societies and Service Choreographies: a Declarative Approach to Specification and Verification. In *International Workshop on Agents, Web-Services and Ontologies: Integrated Methodologies (AWESOME'007)*, 2007. (Cited on page 311.)

[46] F. Chesani, E. Lamma, P. Mello, M. Montali, S. Storari, P. Baldazzi, and M. Manfredi. Compliance Checking of Cancer-Screening Careflows: an Approach Based on Computational Logic. In A. ten Teije, S. Miksch, and P. Lucas, editors, *Book Chapter of Computer-Based Medical Guidelines and Protocols: a Primer and Current Trends*. IOS Press, 2008. (Cited on page 288.)

[47] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Exploiting Inductive Logic Programming Techniques for Declarative Process Mining. *LNCS Transactions on Petri Nets*

and Other Models of Concurrency (ToPNoC), Special Issue on Concurrency in Process-Aware Information Systems, 5460:278–295, 2009. (Cited on page 292.)

[48] F. Chesani, P. Mello, M. Montali, F. Riguzzi, M. Sebastianis, and S. Storari. Checking compliance of execution traces to business rules. In *Proceedings of BPM 2008 Workshops*, volume 17 of *Lecture Notes in Business Information Processing*. Springer Verlag, 2009. (Cited on page 278.)

[49] F. Chesani, P. Mello, M. Montali, and P. Torroni. Ontological Reasoning and Abductive Logic Programming for Service Discovery and Contracting. In A. Gangemi, J. Keizer, V. Presutti, and H. Stoermer, editors, *Proceedings of the 5th Workshop on Semantic Web Applications and Perspectives (SWAP2008)*, volume 429 of *CEUR Workshop Proceedings*, 2009. (Cited on page 311.)

[50] F. Chesani, P. Mello, M. Montali, S. Storari, and P. Torroni. On the Integration of Declarative Choreographies and Commitment-based Agent Societies into the SCIFF Logic Programming Framework. *Multiagent and Grid Systems, Special Issue on Agents, Web Services and Ontologies: Integrated Methodologies*, 6(2), 2010. (Cited on page 311.)

[51] L. Chittaro and A. Montanari. Efficient Temporal Reasoning in the Cached Event Calculus. *Computational Intelligence*, 12:359–382, 1996. (Cited on pages 247 and 249.)

[52] L. Chittaro and A. Montanari. Temporal Representation and Reasoning in Artificial Intelligence: Issues and Approaches. *Annals of Mathematics and Artificial Intelligence*, 28(1-4):47–106, 2000. (Cited on page 248.)

[53] A. K. Chopra and C. P. Singh. Producing Compliant Interactions: Conformance, Coverage, and Interoperability. In *4th International Workshop on Declarative Agent Languages and Technologies IV (DALT 2006), Selected, Revised and Invited Papers*, volume 4327 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 2006. (Cited on page 229.)

[54] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001. (Cited on page 19.)

[55] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):410–425, 2000. (Cited on page 211.)

[56] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978. (Cited on pages 60 and 73.)

[57] E. M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999. (Cited on pages xxiii, 178, 209, 211, and 218.)

[58] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In R. Wilhelm, editor, *Informatics - 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer Verlag, 2001. (Cited on pages 211 and 219.)

[59] S. Colin and L. Mariani. Run-Time Verification. In M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer Verlag, 2005. (Cited on page 234.)

[60] M. Colombetti, N. Fornara, and M. Verdicchio. A Social Approach to Communication in Multiagent Systems. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *First International Workshop on Declarative Agent Languages and Technologies (DALT2004)*, volume 2990 of *Lecture Notes in Artificial Intelligence*, pages 191–220. Springer Verlag, 2004. (Cited on page 127.)

[61] D. B. Fridsma (guest editor). Special Issue on Workflow Management and Clinical Guidelines. *Journal of the American Medical Informatics Association*, 22(1):1–80, 2001. (Cited on page 26.)

[62] G. Decker, J.M. Zaha, and M. Dumas. Execution Semantics for Service Choreographies. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *Proceedings of the 3rd Workshop on Web Services and Formal Method (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 163–177. Springer Verlag, 2006. (Cited on page 126.)

[63] G. Delzanno and A. Podelski. Model Checking in CLP. In *Proocedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999)*, volume 1579 of *Lecture Notes in Computer Science*, pages 223–239. Springer Verlag, 1999. (Cited on page 224.)

[64] S. Demri, F. Laroussinie, and P. Schnoebelen. A Parametric Analysis of the State-Explosion Problem in Model Checking. *Journal of Computer and System Sciences*, 72(4):547–575, 2006. (Cited on pages 211 and 218.)

[65] M. Denecker and D. De Schreye. SLDNFA: An Abductive Procedure for Abductive Logic Programs. *Logic Programming*, 34(2): 111–167, 1998. (Cited on page 73.)

[66] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121. ACM Press, 1975. (Cited on page 1.)

[67] N. Desai, A. K. Chopra, and M. P. Singh. Business Process Adaptations via Protocols. In *2006 IEEE International Conference on*

Services Computing (SCC 2006), pages 103–110. IEEE Computer Society, 2006. (Cited on page 25.)

[68] J. Desel and T. Erwin. Hybrid Specifications: Looking at Workflows From a Run-Time Perspective. *Computer System Science & Engineering*, 15(5):291–302, 2000. (Cited on page 301.)

[69] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of Communicating Data-Driven Web Services. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems (PODS '06)*, pages 90–99. ACM Press, 2006. (Cited on page 126.)

[70] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, 1988. (Cited on pages 58 and 172.)

[71] M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005. (Cited on page 15.)

[72] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In M. A. Ardis and J. M. Atlee, editors, *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP1998)*, pages 7–15. ACM Press, 1998. (Cited on page 51.)

[73] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*. Elsevier and MIT Press, 1990. ISBN 0-444-88074-7, 0-262-22039-3. (Cited on page 48.)

[74] F. A. Schreiber. Is Time a Real Time? An Overview of Time Ontology in Informatics. *Real Time Computing*, F 127:283–307, 1994. (Cited on page 13.)

[75] H. M. Ferreira and D. R. Ferreira. An Integrated Life Cycle for Workflow Management Based on Learning and Planning. *Cooperative Information Systems*, 15(4):485–505, 2006. (Cited on page 301.)

[76] M. Fisher, C. Dixon, and M. Peim. Clausal Remporal Resolution. *ACM Transactions on Computational Logic*, 2(1):12–56, 2001. (Cited on pages 100, 104, 105, 106, and 224.)

[77] N. Fornara and M. Colombetti. Operational Specification of a Commitment-Based Agent Communication Language. In *Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2002)*, pages 535–542. ACM Press, 2002. (Cited on page 127.)

[78] A. Förster, G. Engels, T. Schattkowsky, and R. van der Straeten. Verification of Business Process Quality Constraints Based on Visual Process Patterns. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 197–208. IEEE Computer Society, 2007. (Cited on page 226.)

[79] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Composition. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 152–161, 2003. (Cited on pages 126 and 227.)

[80] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Logic Programming*, 37(1-3):95–138, 1998. (Cited on page 172.)

[81] X. Fu, T. Bultan, and J. Su. Synchronizability of Conversations among Web Services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005. (Cited on page 126.)

[82] T. H. Fung and R. A. Kowalski. The Iff Proof Procedure for Abductive Logic Programming. *Logic Programming*, 33(2):151–165, 1997. (Cited on pages 73, 154, 157, and 257.)

[83] P. Gastin and D.Oddoux. Fast LTL to Büchi Automata Translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001. (Cited on pages xxiii and 210.)

[84] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A Logic Language for Executable Specifications of Real-Time Systems. *Systems and Software*, 12(2):107–123, 1990. (Cited on page 223.)

[85] A. Ghose and G. Koliadis. Auditing Business Process Compliance. In B. J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Fifth International Conference on Service-Oriented Computing (ICSOC 2007)*, volume 4749 of *Lecture Notes in Computer Science*, pages 169–180. Springer Verlag, 2007. (Cited on page 226.)

[86] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, volume 412-416. IEEE Computer Society, 2001. (Cited on pages 222, 297, 298, and 299.)

[87] S. Goedertier. *Declarative Techniques for Modeling and Mining Business Processes*. PhD thesis, Katholieke Universiteit Leuven, 2008. (Cited on pages 2, 28, 124, and 301.)

[88] S. Goedertier and J. Vanthienen. Designing Compliant Business Processes with Obligations and Permissions. In J. Eder and S. Dustdar, editors, *Proceedings of the Business Process Management Workshops (BPMN2006)*, volume 4103 of *Lecture Notes in Computer Science*, pages 5–14. Springer Verlag, 2006. (Cited on page 225.)

[89] G. Gößler and J. Sifakis. Composition for Component-Based Modeling. *Science of Computer Programming*, 55(1-3):161–183, 2005. (Cited on page 227.)

[90] G. Governatori, Z. Milosevic, and S. W. Sadiq. Compliance Checking Between Business Processes and Business Contracts. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006)*, pages 221–232. IEEE Computer Society, 2006. (Cited on pages 142, 225, and 226.)

[91] G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. Discovering Expressive Process Models by Clustering Log Traces. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1010–1027, 2006. (Cited on page 301.)

[92] T. R. G. Green. Cognitive Dimensions of Notations. *People and Computers*, V:443–460, 1989. (Cited on pages 35 and 46.)

[93] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: a 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996. (Cited on page 46.)

[94] G. Gupta and E. Pontelli. A Constraint-Based Approach for Specification and Verification of Real-Time Systems. In *Proceedings of the 18th IEEE Real-time Systems Symposium (RTSS 1997)*, pages 230–239. IEEE Computer Society, 1997. (Cited on page 225.)

[95] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive Logic Programming and Its Applications. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP2007)*, pages 27–44, 2007. (Cited on page 310.)

[96] S. Hallé and R. Villemaire. Runtime Monitoring of Web Service Choreographies Using Streaming XML. In *Proceedings of to the 24th Annual ACM Symposium on Applied Computing (ACM SAC 2009)*, 2009. (Cited on page 126.)

[97] J. Y. Halpern and M. Y. Vardi. Model Checking vs. Theorem Proving: A Manifesto. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pages 151–176, 1991. (Cited on page 221.)

[98] C. Hartshorn and P. Weiss. *Collected Papers of Charles Sanders Peirce*, volume 2. Harvard University Press, 1932. (Cited on pages 67 and 68.)

[99] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003. (Cited on pages 126, 210, and 211.)

[100] M. N. Huhns and M. P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, 2005. (Cited on pages 2 and 24.)

[101] J. Jaffar and M. J. Maher. Constraint Logic Programming: a Survey. *Logic Programming*, 19-20:503–582, 1994. (Cited on pages 57, 73, 157, and 159.)

[102] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992. (Cited on pages 57 and 172.)

[103] J. Jaffar, M. J. Maher, K. Marriott, and P. J.Stuckey. The Semantics of Constraint Logic Programs. *Logic Programming*, 37(1-3):1–46, 1998. (Cited on page 157.)

[104] F. Joualt and I. Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of the ACM Symposium on Applied Computing (SAC06)*, 2006. (Cited on page 311.)

[105] K. R. Apt and M. Bezem. Acyclic Programs. In D. H. Warren, editor, *Logic Programming*, pages 617–633. MIT Press, 1990. (Cited on page 164.)

[106] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Logic and Computation*, 2(6):719–770, 1992. (Cited on pages 68 and 69.)

[107] H. Kautz and P. Ladin. Integrating Metric and Qualitative Temporal Reasoning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI 1991)*, volume 1, pages 241–246. AAAI Press/The MIT Press, 1991. (Cited on page 13.)

[108] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web Services Choreography Description Language Version 1.0, 2004. http://www.w3.org/TR/ws-cdl-10/. (Cited on pages 20 and 21.)

[109] M. Kifer, R. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen, and D. Fensel. A Logical Framework for Web Service Discovery. In *Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, 2004. (Cited on page 21.)

[110] E. Kindler. Safety and Liveness Properties: a Survey. *EATCS-Bulletin*, 53, 1994. (Cited on page 138.)

[111] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979. (Cited on page 14.)

[112] R. A. Kowalski and F. Sadri. Towards a Unified Agent Architecture that Combines Rationality with Reactivity. In *Proceedings of the International Workshop on Logic in Databases (LID'96)*, volume 1154 of *Lecture Notes in Computer Science*, pages 137–149. Springer Verlag, 1996. (Cited on pages 249 and 250.)

[113] R. A. Kowalski and M. Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986. (Cited on page 246.)

[114] R. A. Kowalski and M. Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986. (Cited on pages 5 and 308.)

[115] K. Kunen. Negation in Logic Programming. In *Logic Programming*, volume 4, pages 289–308, 1987. (Cited on page 73.)

[116] O. Kupferman, N. Piterman, and M. Y. Vardi. From Liveness to Promptness. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 406–419. Springer Verlag, 2007. (Cited on page 139.)

[117] E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Inducing Declarative Logic-Based Models from Labeled Traces. In M. Rosemann and M. Dumas, editors, *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 344–359. Springer Verlag, 2007. (Cited on pages 277 and 292.)

[118] E. Lamma, P. Mello, F. Riguzzi, and S. Storari. Applying Inductive Logic Programming to Process Mining. In *Proceedings of the 17th International Conference on Inductive Logic Programming (ILP2008)*, volume 4894 of *Lecture Notes in Artificial Intelligence*, pages 132–146. Springer Verlag, 2008. (Cited on pages 277 and 292.)

[119] A. Lazovik, M. Aiello, and M. Papazoglou. Associating Assertions with Business Processes and Monitoring their Execution. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC '04)*, pages 94–104. ACM Press, 2004. (Cited on page 297.)

[120] F. Leymann and D. Roller. *Production Workflow - Concepts and Techniques*. Prentice Hall, 2000. (Cited on page 16.)

[121] X. Liu, S. Müller, and K. Xu. A Static Compliance-Checking Framework for Business Process Models - References. In *IBM Systems Journal*, volume 46, pages 335–362, 2007. (Cited on page 226.)

[122] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 2nd edition, 1987. (Cited on page 53.)

[123] L. Luccarini, G. L. Bragadin, M. Mancini, P. Mello, M. Montali, and D. Sottara. Process Quality Assessment in Automatic Management of Wastewater Treatment Plants Using Formal Verification. In *Proceedings of Simposio Internazionale di Ingegneria Sanitaria Ambientale (SIDISA 2008)*, 2008. (Cited on pages xxiii and 290.)

[124] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001. (Cited on page 66.)

[125] H. Ludwig, A. Dan, and R. Kearney. Crona: An Architecture and Library for Creation and Monitoring of WS-agreements. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC '04)*, pages 65–74. ACM Press, 2004. (Cited on page 297.)

[126] L. T. Ly, S. Rinderle, and P. Dadam. Integration and Verification of Semantic Constraints in Adaptive Process Management Systems. *Data and Knowledge Engineering*, 64(1):3–23, 2008. (Cited on page 125.)

[127] M. Baldoni and C. Baroglio and A. Martelli and V. Patti. A Priori Conformance Verification for Guaranteeing Interoperability in Open Environments. In A. Dan and W. Lamersdorf, editors, *Proocedings of the 4th International Conference on Service-Oriented Computing (ICSOC 2006)*, volume 4294 of *Lecture Notes in Computer Science*. Springer Verlag, 2006. (Cited on page 228.)

[128] K. Mahbub and G. Spanoudakis. Run-Time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. In *Proeedings of the 3rd IEEE International Conference on Web Services (ICWS 2005)*, pages 257–265. IEEE Computer Society, 2005. (Cited on page 298.)

[129] K. Mahbub and G. Spanoudakis. Monitoring WS-Agreements: An Event Calculus-Based Approach. In L. Baresi and E. Di Nitto, editors, *Test and Analysis of Web Services*, pages 265–306. Springer Verlag, 2007. (Cited on page 298.)

[130] A. U. Mallya and M. P. Singh. Modeling Exceptions Via Commitment Protocols. In *Proceedings of the 4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 122–129. ACM, 2005. (Cited on page 311.)

[131] A. U. Mallya, N. Desai, A. K. Chopra, and M. P. Singh. OWL-P: OWL for protocol and processes. In *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 139–140. ACM Press, 2005. (Cited on page 25.)

[132] J. Marques-Silva. Model Checking with Boolean Satisfiability. *Algorithms*, 63(1-3):3–16, 2008. (Cited on page 223.)

[133] A. Martens. Analyzing Web Service Based Business Processes. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer Verlag, 2005. (Cited on page 227.)

[134] A. Martens. Consistency between Executable and Abstract Processes. In *Proceedings of International IEEE Conference on e-Technology, e-Commerce, and e-Services (EEE'05)*, pages 60–67. IEEE Computer Society, 2005. (Cited on page 227.)

[135] D. L. Martin, M. H. Burstein, D. V. McDermott, S. A. McIlraith, M. Paolucci, K. P. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan. Bringing Semantics to Web Services with OWL-S. In *Proceedings of the 16th International World Wide Web Conference (WWW2007)*, pages 243–277, 2007. (Cited on page 227.)

[136] P. Massuthe, W. Reisig, and K. Schmidt. An Operating Guideline Approach to the SOA. In *Proceedings of the 2nd South-East European Workshop on Formal Methods 2005 (SEEFM05)*, 2005. (Cited on page 227.)

[137] J. McCarthy and P. J. Hayes. Some Philosophical Problems From the StandPoint of Artificial Intelligence. *Machine Intelligence*, 4: 463–502, 1969. (Cited on page 247.)

[138] K. L. McMillan. Interpolation and SAT-Based Model Checking. In W. A. Hunt Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13, 2003. (Cited on page 224.)

[139] M. Mecella, F. Parisi Presicce, and B. Pernici. Modeling E-service Orchestration through Petri Nets. In *Proceedings of the Third International Workshop on Technologies for E-Services*, volume 2644 of *Lecture Notes in Computer Science*, pages 38–47. Springer Verlag, 2002. (Cited on page 126.)

[140] I. Meiri. Combining Qualitative and Quantitative Constraints in Temporal Reasoning. *Artificial Intelligence*, 87(1-2):343–385, 1996. (Cited on page 13.)

[141] S. Miksch, Y. Shahar, and P. Johnson. Asbru: a Task-Specific, Intention-Based, and Time-Oriented Language for Representing Skeletal Plans. In *Proceedings of the 7th Workshop on Knowledge Engeneering Methods and Languages*, pages 9–20, 1997. (Cited on page 125.)

[142] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Journal of Information and Computation*, 100(1):1–40, 1992. (Cited on page 126.)

[143] M. Montali, F. Chesani, P. Mello, and S. Storari. Testing Careflow Process Execution Conformance by Translating a Graphical Language to Computational Logic. In R. Bellazzi, A. Abu-Hanna, and J. Hunter, editors, *Proceedings of the 11th International Conference on Artificial Intelligence in Medicine (AIME'07)*, volume 4594 of *Lecture Notes in Computer Science*, pages 479–488. Springer Verlag, 2007. (Cited on pages 227, 235, and 288.)

[144] M. Montali, M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verification from Declarative Specifications Using Logic Programming. In M. Garcia De La Banda and

E. Pontelli, editors, *24th International Conference on Logic Programming (ICLP)*, number 5366 in Lecture Notes in Computer Science, pages 440–454. Springer Verlag, 2008. (Cited on pages xxv, 216, and 217.)

[145] M. Montali, F. Chesani, P. Mello, and P. Torroni. Verification of Choreographies During Execution Using the Reactive Event Calculus. In R. Bruni and K. Wolf, editors, *Proceedings of the 5th International Workshop on Web Service and Formal Methods (WS-FM2008)*, volume 5387 of *Lecture Notes in Computer Science*, pages 55–72. Springer Verlag, 2009. (Cited on page 262.)

[146] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative Specification and Verification of Service Choreographies. *ACM Transactions on the Web - submitted to the second round of reviews*, 2009. (Cited on pages xxi, 2, 24, 32, 35, 97, 137, and 228.)

[147] S. Moschoyiannis and M. W. Shields. A Set-Theoretic Framework for Component Composition. *Fundamenta Informaticae*, 59(4):373–396, 2004. (Cited on page 227.)

[148] S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Methods. *Logic Programming*, 19/20:629–679, 1994. (Cited on page 292.)

[149] N. Mulyar, M. Pesic, W. M. P. van der Aalst, and M. Peleg. Declarative and Procedural Approaches for Modelling Clinical Guidelines: Addressing Flexibility Issues. In *Proceedings of BPM 2007 Workshops*, pages 335–346, 2007. (Cited on pages 32, 35, and 125.)

[150] N. Mulyar, W. M. P. van der Aalst, and M. Peleg. A Pattern-based Analysis of Clinical Computer-Interpretable Guideline Modelling Languages. *Journal of the American Medical Informatics Association*, 14:781–787, 2007. (Cited on page 125.)

[151] G. Naumovich and L. A. Clarke. Classifying Properties: an Alternative to the Safety-Liveness Classification. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: 21st Century Applications*, pages 159–168. ACM Press, 2000. (Cited on page 138.)

[152] A. Dal Palú, A. Dovier, and E. Pontelli. Heuristics, Optimizations, and Parallelism for Protein Structure Prediction in CLP(FD). In P. Barahona and A. P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–241, 2005. (Cited on page 58.)

[153] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: a Research Roadmap. *Cooperative Information Systems*, 17(2):223–255, 2008. (Cited on page 18.)

[154] J. Pearl. Embracing Causality in Formal Reasoning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI1987)*, pages 369–373, 1987. (Cited on page 68.)

[155] M. Peleg, S. Tu, J. Bury, P. Ciccarese, N. Jones, J. Fox, R. A. Greenes, R. Hall, P.D. Johnson, N. Jones, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, E. H. Shortliffe, and M. Stefanelli. Comparing Computer-Interpretable Guideline Models: A Case-Study Approach. *Journal of the American Medical Informatics Association*, 10(1):52–68, 2003. (Cited on pages 26 and 125.)

[156] C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, 2003. (Cited on page 19.)

[157] M. Pesic. *Constraint-Based Workflow Management Systems: Shifting Controls to Users*. PhD thesis, Beta Research School for Operations Management and Logistics, Eindhoven, 2008. (Cited on pages xxii, 2, 6, 16, 28, 32, 35, 36, 46, 48, 51, 115, 124, 125, 137, 143, 144, 209, 245, 246, 269, 299, 307, and 308.)

[158] M. Pesic and W. M. P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In *Proceedings of the BPM 2006 Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer Verlag, 2006. (Cited on pages 2, 6, 16, 17, 32, 35, 48, 51, 124, 209, 299, and 307.)

[159] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 287–300. IEEE Computer Society, 2007. (Cited on pages 137, 222, 269, and 310.)

[160] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst. Constraint-Based Workflow Models: Change Made Easy. In R. Meersman and Z. Tari, editors, *Proceedings of the OTM 2007 Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Artificial Intelligence*, pages 77–94. Springer Verlag, 2007. (Cited on pages 278 and 300.)

[161] N. Pissinou, R. T. Snodgrass, R. Elmasri, I. S. Mumick, T. Özsu, B. Pernici, A. Segev, B. Theodoulidis, and U. Dayal. Towards an Infrastructure for Temporal Databases: report of an invitational ARPA/NSF workshop. *ACM SIGMOD Record*, 23(1):35–51, 1994. (Cited on page 279.)

[162] S. R. Ponnekanti and A. Fox. Interoperability Among Independently Evolving Web Services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware (Middleware '04)*, pages 331–351. Springer Verlag, 2004. (Cited on page 228.)

[163] M. Pradella, A. Morzenti, and P. San Pietro. Refining Real-Time System Specifications through Bounded Model- and

Satisfiability-Checking. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 119–127. IEEE Computer Society, 2008. (Cited on page 223.)

[164] R. Barruffi and M. Milano and R. Montanari. Planning for Security Management. *IEEE Intelligent Systems*, 16(1):74–80, 2001. (Cited on page 58.)

[165] M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive Process Management with ADEPT2. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, pages 1113–1114. IEEE Computer Society, 2005. (Cited on pages 17, 278, and 300.)

[166] R. Reiter. On Closed-Word Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978. (Cited on page 159.)

[167] D. Roman and M. Kifer. Semantic Web Service Choreography: Contracting and Enactment. In A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan, editors, *Proceedings of the 7th International Semantic Web Conference (ISWC 2008)*, volume 5318 of *Lecture Notes in Computer Science*, pages 550–566. Springer Verlag, 2008. (Cited on pages 229 and 299.)

[168] M. Rouached, W. Fdhila, and C. Godart. A Semantical Framework to Engineering WSBPEL Processes. *Information Systems and E-Business Management*, 7(2):223–250, 2008. (Cited on page 299.)

[169] K. Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. In *Model Checking Software. Proceedings of the 14th International SPIN Workshop*, volume 4595 of *Lecture Notes in Computer Science*, pages 149–167. Springer Verlag, 2007. (Cited on pages 213, 214, 215, and 223.)

[170] A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In *Proceedings of the BPM 2005 Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer Verlag, 2006. (Cited on page 297.)

[171] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An Abductive Approach for Analysing Event-Based Requirements Specifications. In P. J. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming (ICLP 2002)*, volume 2401 of *Lecture Notes in Computer Science*, pages 22–37. Springer Verlag, 2002. (Cited on page 224.)

[172] S. W. Sadiq, G. Governatori, and K. Namiri. Modeling Control Objectives for Business Process Compliance. In *Proceedings of the 5th International Conference on Business Process Management (BPM*

2007), volume 4714 of *Lecture Notes in Computer Science*, pages 149–164. Springer Verlag, 2007. (Cited on page 300.)

[173] S. Wasim Sadiq, M. E. Orlowska, and W. Sadiq. Specification and Validation of Process Constraints for Flexible Workflows. *Information Systems*, 30(5):349–378, 2005. (Cited on page 124.)

[174] G. Sartor. *Legal Reasoning*, volume 5 of *Treatise*. Kluwer, 2004. (Cited on page 127.)

[175] B. H. Schlingloff, A. Martens, and K. Schmidt. Modeling and Model Checking Web Services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems*, 126:3–26, 2005. (Cited on page 227.)

[176] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. M. P. van der Aalst. Towards a Taxonomy of Process Flexibility. In Z. Bellahsene, C. Woo, E. Hunt, X. Franch, and R. Coletta, editors, *Proceedings of the Forum at the CAiSE'08 Conference*, volume 344 of *CEUR Workshop Proceedings*, pages 81–84, 2008. (Cited on page 17.)

[177] M. Shanahan. The Event Calculus Explained. In M. Wooldridge and M. M. Veloso, editors, *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *Lecture Notes in Computer Science*, pages 409–430. Springer Verlag, 1999. ISBN 3-540-66428-9. (Cited on pages 128, 246, and 248.)

[178] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In W. A. Hunt Jr. and S. D. Johnson, editors, *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125, 2000. (Cited on page 223.)

[179] M. P. Singh. Agent Communication Language: Rethinking the Principles. *IEEE Computer*, pages 40–47, 1998. (Cited on page 127.)

[180] D. Sottara, L. Luccarini, and P. Mello. AI Techniques for Waste Water Treatment Plant Control Case Study: Denitrification in a Pilot-Scale SBR. In B. Apolloni, R. J. Howlett, and L. C. Jain, editors, *Proceedings of the 17th Italian Workshop on Neural Networks, part of the 11th International Conference on Knowledge-Based Intelligent Information and Engineering Systems(KES2007)*, volume 4692 of *Lecture Notes in Computer Science*, pages 639–646. Springer Verlag, 2007. (Cited on page 290.)

[181] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 2nd edition, 1994. (Cited on page 53.)

[182] A. Ten Teije, S. Miksch, and P. Lucas, editors. *Computer-based Medical Guidelines and Protocols: A Primer and Current Trends*, volume 139 of *Studies in Health Technology and Informatics*. IOS Press, Amsterdam, July 2008. (Cited on pages 26 and 125.)

[183] M. H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis. A Case Study on the Automated Verification of Groupware Protocols. In G. C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 596–603. ACM, 2005. (Cited on page 285.)

[184] P. Terenziani, G. Molino, and M. Torchio. A Modular Approach for Representing and Executing Clinical Guidelines. *Artificial Intelligence in Medicine*, 23(3):249–276, 2001. (Cited on page 26.)

[185] W. M. P. van der Aalst. How to handle dynamic change and capture management information? An approach based on generic workflow models. *Computer Systems, Science and Engineering*, 16 (5):295–318, 2001. (Cited on page 123.)

[186] W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *Proceedings of the 3rd Workshop on Web Services and Formal Methods (WS-FM2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23. Springer Verlag, 2006. (Cited on pages xxiv, 17, 22, 32, 35, 42, 95, 96, 97, 111, 139, and 228.)

[187] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005. (Cited on pages 15 and 125.)

[188] W. M. P. van der Aalst, T. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004. (Cited on pages 291 and 301.)

[189] W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, N. Russell, H. M. W. Verbeek, and P. Wohed. Life After BPEL? In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *Proceedings of the 2nd International Workshop on Web Services and Formal Methods (WS-FM 2005)*, volume 3670 of *Lecture Notes in Computer Science*, pages 35–50. Springer Verlag, 2005. (Cited on pages 2 and 21.)

[190] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, R. S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H. M. W. Verbeek, and A. J. M. M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *Proceedings of the 28th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer*

Science, pages 484–494. Springer Verlag, 2007. (Cited on pages 5, 277, 297, and 309.)

[191] W. M.P. van der Aalst, A. H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. (Cited on page 36.)

[192] W. M.P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003. (Cited on page 31.)

[193] Wil M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance Checking of Service Behavior. *ACM Transactions on Internet Technologies*, 8(3), 2008. (Cited on pages 126 and 297.)

[194] W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Web Service Composition Languages: Old Wine in New Bottles? In G. Chroust and C. Hofer, editors, *Proceeding of the 29th EUROMICRO Conference: New Waves in System Architecture*, pages 298–305. IEEE Computer Society, Los Alamitos, CA, 2003. (Cited on pages 2 and 18.)

[195] W.M.P. van der Aalst, H.T. de Beer, and B.F. van Dongen. Process Mining and Verification of Properties: An Approach based on Temporal Logic. In R. Meersman and Z. Tari, editors, *Proceedings of the OTM 2005 Confederated International Conferences CoopIS, DOA, and ODBASE*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer Verlag, 2005. (Cited on pages 278, 283, and 300.)

[196] L. van der Torre. Contextual Deontic Logic: Normative Agents, Violations and Independence. *Annals of Mathematics and Artificial Intelligence*, 37(1):33–63, 2003. (Cited on page 127.)

[197] B. F. van Dongen. *Process Mining and Verification*. PhD thesis, Eindhoven University of Technology, 2007. (Cited on pages xxiii and 276.)

[198] B. F. van Dongen and W. M. P. van der Aalst. Multi-phase Process Mining: Building Instance Graphs. In *23rd International Conference on Conceptual Modeling (ER2004)*, volume 3288 of *Lecture Notes in Computer Science*, pages 362–376. Springer Verlag, 2004. (Cited on pages 291 and 301.)

[199] B. F. van Dongen and W. M. P. van der Aalst. A Meta Model for Process Mining Data. In J. Casto and E. Teniente, editors, *Proceedings of the CAiSE'05 Workshops (EMOI-INTEROP Workshop)*, volume 2, pages 309–320. FEUP, Porto, Portugal, 2005. (Cited on pages xxii, 13, 109, 114, 278, and 279.)

[200] W. J. van Hoeve. The AllDifferent Constraint: a Survey. In *Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001. (Cited on page 90.)

[201] B. Weber, S. Rinderle, and M. Reichert. Change Patterns and Change Support Features in Process-Aware Information Systems. In *Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAiSE 2007)*, volume 4495 of *Lecture Notes in Computer Science*, pages 574–588. Springer Verlag, 2007. (Cited on pages 17 and 300.)

[202] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Verlag, 2007. (Cited on page 14.)

[203] S. A. White. Business Process Modeling Notation Specification 1.0. Technical report, OMG, 2006. (Cited on pages 15, 22, and 225.)

[204] G. H. Wright. Deontic logic. *Mind*, 60:1–15, 1951. (Cited on page 127.)

[205] I. Xanthakos. *Semantic Integration of Information by Abduction*. PhD thesis, Imperial College London, 2003. (Cited on page 164.)

[206] P. Yolum and M. P. Singh. Flexible Protocol Specification and Execution: Applying Event Calculus Planning Using Commitments. In *Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2002)*, pages 527–534. ACM Press, 2002. (Cited on pages 2, 25, 127, 128, and 311.)

[207] J. M. Zaha, A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In R. Meersman and Z. Tari, editors, *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, volume 4275 of *Lecture Notes in Computer Science*, pages 145–162. Springer Verlag, 2006. (Cited on page 126.)

colophon

This thesis was typeset with LaTeX 2_ε using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

The typographic style was inspired by Bringhurst's genius as presented in *The Elements of Typographic Style* [36]. It is available for LaTeX via CTAN as "`classicthesis`".

Final Version as of March 17, 2009 at 8:58.