Dottorato di Ricerca in Informatica
Università di Bologna, Padova
INF/01 INFORMATICA

Automatic Code Generation: From Process Algebraic Architectural Descriptions to Multithreaded Java Programs

Edoardo Bontà

Marzo 2008

Coordinatore:
Prof. Özalp Babaoğlu

Relatore:
Prof. Marco Bernardo

To the memory of my father, Domenico Bontà.

Abstract

Process algebraic architectural description languages provide a formal means for modeling software systems and assessing their properties. In order to bridge the gap between system modeling and system implementation, in this thesis an approach is proposed for automatically generating multithreaded object-oriented code from process algebraic architectural descriptions, in a way that preserves – under certain assumptions – the properties proved at the architectural level.

The approach is divided into three phases, which are illustrated by means of a running example based on an audio processing system. First, we develop an architecture-driven technique for thread coordination management, which is completely automated through a suitable package. Second, we address the translation of the algebraically-specified behavior of the individual software units into thread templates, which will have to be filled in by the software developer according to certain guidelines. Third, we discuss performance issues related to the suitability of synthesizing monitors rather than threads from software unit descriptions that satisfy specific constraints.

In addition to the running example, we present two case studies about a video animation repainting system and the implementation of a leader election algorithm, in order to summarize the whole approach.

The outcome of this thesis is the implementation of the proposed approach in a translator called PADL2Java and its integration in the architecture-centric verification tool TwoTowers.

Acknowledgements

I would like to acknowledge the following people who contributed in various ways to this thesis.

A very special thanks goes to my supervisor Marco Bernardo who has guided me through all stages of the thesis and taught me to carry out research properly.

Then, I would like to thank Jeff Magee and Jeff Kramer. Their experience, their willingness, and their support have been fundamental to the conception of a significant part of this work.

I would also like to thank my tutors Lorenzo Donatiello and Simone Martini for their precious suggestions, as well as the two reviewers Paola Inverardi and Frédéric Lang for their constructive criticism that has helped me to improve the thesis.

Finally, I would like to thank all the colleagues of the Information Science and Technology Institute of the University of Urbino for their willingness to help me in different situations, as well as my mother Maria Chiara and my brother Giuseppe for the moral support and encouragement that they gave me during these years.

Contents

Abstract iii

Acknowledgements v

List of Tables xi

List of Figures xiii

1 Introduction 1

1.1 Process Algebraic Architectural Description Languages . . . 2
1.2 Code Generation Approaches . . . 3
1.3 Multithreaded Programs . . . 4
1.4 Proposed Approach . . . 5
1.5 Structure of the Thesis . . . 6

2 Automatic Code Generation 9

2.1 CASE Tools . . . 10
2.2 Model-Driven Development and MDA . . . 11
2.3 Czarnecki-Helsen Classification of Model Transformation Approaches . . . 12
2.4 Adopted Model Transformation Approach . . . 15

3 The Architectural Description Language PADL 17

3.1 Process Algebra . . . 17
3.2 PADL Descriptions and Notations . . . 20
3.2.1 PADL Textual Notation . . . 20
3.2.2 PADL Graphical Notation . . . 23
3.3 The semantics for PADL . . . 23
3.4 Handling Non-Synchronous Interactions . . . 26
3.5 Running Example: An Audio Processing System . . . 29
3.5.1 Informal Specification of the Audio Processing System . . . 30
3.5.2 PADL Description of the Audio Processing System . . . 33

4 Thread Coordination Management **41**
4.1 Thread Communication Model . . . 41
4.2 The Java Package `Sync` . . . 42
4.3 The Layer `Connector` . . . 43
4.4 The Layer `Port` . . . 46
4.5 The Layer `RunnableElem` . . . 48
4.6 The Layer `RunnableArchi` . . . 50
4.7 Audio Processing System: Phase 1 . . . 53

5 Thread Behavior Generation **57**
5.1 Thread Generation Model . . . 57
5.2 Synthesizing Thread Method `run()` . . . 59
5.3 Translating `stop` . . . 59
5.4 Translating Behavioral Invocations . . . 60
5.5 Translating Action Prefixes . . . 60
5.6 Translating Choices . . . 60
5.7 Preservation of Architectural Properties . . . 62
5.8 Audio Processing System: Phase 2 . . . 65

6 Monitor Synthesis **89**
6.1 Monitor Constraints . . . 90
6.1.1 Terminology . . . 90
6.1.2 Thread-Monitor Interaction Model . . . 91
6.1.3 Constraint 1: No Cycles of Internal Actions . . . 92
6.1.4 Constraint 2: No Attached Monitor Type Instances . . . 93
6.1.5 Constraint 3: No Non-Synchronous Interactions . . . 93
6.1.6 Constraint 4: No Non-Disjoint Hybrid Choices . . . 93
6.2 Syntactic Transformation into Monitor Normal Form . . . 94
6.2.1 Step 1: Rewriting Complex Choices . . . 96
6.2.2 Step 2: Splitting Defining Equations . . . 97
6.2.3 Step 3: Building the Interacting Choice Equation . . . 99
6.2.4 Step 4: Building Setting Equations . . . 101
6.2.5 Step 5: Rearranging the Interacting Choice Equation . . . 104
6.2.6 Correctness of the Transformation . . . 107
6.3 Generating the Core Monitor Class . . . 107
6.3.1 Translating Internal Actions into Stub Class Methods . . . 109
6.3.2 Declaring IAS Stub and Synthesizing the Monitor Class Constructor . . . 110
6.3.3 Translating Setting and Internal Equations . . . 110
6.3.4 Translating the Interacting Choice Equation . . . 111
6.3.5 Synthesizing the Starting Method . . . 113
6.4 Generating the Monitor Wrapper Class . . . 113

6.5 Audio Processing System: Phase 3 115

7 The Translator PADL2Java 125

7.1 Implementation of PADL2Java 125
7.2 Structure of the Generated Code 126
7.3 Translating Data Types 128
7.4 Translation Options 129
7.5 Audio Processing System: Completing Code Generation 130
7.6 Æmilia: A Performance-Oriented Variant of PADL 132
7.7 Integration of PADL2Java in TwoTowers 132

8 Case Studies 137

8.1 A Video Animation Repainting System 137
8.1.1 Informal Specification of the Video Animation Repainting System 137
8.1.2 PADL Description of the Video Animation Repainting System 139
8.1.3 Verifying Properties of the Video Animation Repainting System 146
8.1.4 Synthesizing the Video Animation Repainting System 146
8.2 A Leader Election System 166
8.2.1 The Fokkink-Pang Leader Election Algorithm 166
8.2.2 Modeling a Leader Election System 168
8.2.3 Æmilia Description of the Leader Election System 169
8.2.4 Analyzing the Leader Election System 175
8.2.5 Synthesizing the Leader Election System 178
8.2.6 Results on the Implementation Side 193

9 Conclusion 197

9.1 Summary of Results 197
9.2 Related Work 199
9.3 Future Research 201

References 203

List of Tables

3.1 Process term syntax for PA 18
3.2 Structure of a PADL textual description 21

4.1 Structure of a `RunnableElem`-implementing class 48
4.2 Structure of the `RunnableArchi`-implementing class 51

5.1 Structure of thread method `run()` 59

6.1 Structure of a core monitor class 108
6.2 Structure of a monitor wrapper class 113

List of Figures

3.1 Synchronous, semi-synchronous and asynchronous communications 27
3.2 Extended flow graph of the audio processing system 31

5.1 Internal action refinement and related statement abstraction 63

6.1 Component control switch from native-thread T to monitor M 92

7.1 Generated Java classes/packages and dependencies from package `Sync` 127
7.2 Architecture of TwoTowers 6.0 133

8.1 Extended flow graph of the video animation repainting system 140
8.2 Extended flow graph of the leader election system 170
8.3 Probability of electing a leader 178

Chapter 1

Introduction

The growing complexity and the increasing size of modern software systems demand the adoption of notations for formal or semi-formal system modeling (model-driven approach [60, 30, 53]). The main objective of such notations is to enable the production of incremental design documents, to be shared by all the people contributing to the various system development phases.

Another task that the aforementioned notations should carry out is to allow for the rigorous and hopefully automated analysis of functional and non-functional system properties, in order to avoid delays and cost increases due to the late discovery of errors in the system development process. In fact it is widely recognized that the verification of system properties finds its own rightful place in the architectural design phase [61, 15]. The reason is that this phase precedes system implementation, hence it opens the way to early property assessment. Moreover, this phase provides declarative/behavioral/topological system models that are complete at a high level of abstraction.

As observed e.g. in [35], one of the big issues in the software engineering field is guaranteeing that the implementation of a software system conforms to its architectural description. This amounts to say that a way has to be found to check whether the properties verified at the architectural level are preserved at the code level. In this respect, it may be helpful to generate code directly from architectural descriptions, as these represent abstract models of the final systems. Indeed, the purpose of automatic code generation should be not only to speed up system implementation, but also to ensure conformance by construction.

This is the issue that we wish to address in this thesis in the specific case of concurrent software systems represented through process algebraic architectural descriptions.

1.1 Process Algebraic Architectural Description Languages

Several architectural description languages (ADLs) have been proposed in the literature, see e.g. [55]. Many of them – like Wright [4], Darwin/FSP [51, 52], and PADL/Æmilia [13, 1, 14, 6] – are based on process algebra (PA) due to its support to compositional modeling [56, 40, 7]. It is worth noting that PA is compositional, but not component-oriented. Thus, from the point of view of PA, its architectural versions are a significant step forward in terms of usability. In fact, they give special prominence to the main architectural concepts – components, connectors, and styles – while hiding PA technicalities to the designer.

On the modeling side, this architectural upgrade has three important consequences. First, it permits to describe the behavior of the components separately from the system topology, thus overcoming the modeling difficulties deriving from the direct use of certain PA operators like e.g. parallel composition. Second, it highlights the interactions among components and the classification of their communications, thus allowing for static checks to establish the system model well-formedness. Third, it fosters the reuse of the specification of single components as well as of entire systems, thus supporting the compositional and hierarchical modeling of system families.

On the analysis side, process algebraic ADLs inherit all the techniques applicable to PA, in particular equivalence checking and compositional state space minimization [23]. In addition, such languages are equipped with ad-hoc analysis techniques (see, e.g., [4, 43, 20, 1, 6]). These ad-hoc techniques are useful for (*i*) detecting architectural mismatches – deriving from components that are correct if taken separately but that do not satisfy certain requirements when assembled together – (*ii*) generating diagnostic information – in order to pinpoint those

components from which the mismatches arise – and (*iii*) comparing different architectural designs on the basis of non-functional aspects.

Such component-oriented analysis techniques rely on notions borrowed from PA – such as behavioral equivalences [37] – or suitable compositional models – like queueing networks [48]. They can be used for studying various properties, among which e.g. deadlock freedom and typical average performance metrics, and in many cases scale from single architectures to families of architectures called architectural types [13, 1].

1.2 Code Generation Approaches

Among the various approaches for automatically generating code from architectural descriptions, we can distinguish two families on the basis of the distance between the formalism used for describing software architectures and the implementation language in which code is generated.

The first family, characterized by an "exogenous" approach, is the long-distance one. More precisely, in this family the formalism is kept well separated from the implementation language, and descriptions are entirely translated into code. To this family belong properly-defined ADLs endowed with code generation facilities as Aesop [36], C2SADEL [54], and Darwin [51].

The second family, characterized by a "semi-endogenous" approach, is the short-distance one. In this family, the architectural formalism is embedded in the implementation code in form of special comments, as in SyncGen [27], or in form of special keywords and statements, as in ArchJava [3]. Here, only the special symbols are translated into implementation code, while the rest is left unchanged.

While the latter approach can offer a flexible support for software developers – as classical programming techniques and patterns can be applied when designing systems – the level of abstraction of the underlying architectural formalism is usually low. In the case of process algebraic ADLs, the approach typically adopted for generating code is the former. The reason is that such languages are specifically conceived for abstracting high-level properties of entire software systems, hence they are distant from implementation languages.

1.3 Multithreaded Programs

Concerning the kind of code to be generated, we observe that the increasing importance of multimedia and real-time applications often causes the development of concurrent software. This activity is well supported by multithreading. Unlike traditional concurrent programming, where each code unit is a distinct program executed as a distinct process, multithreading is based on the simultaneous execution of several portions of code – often sharing some data – within a single program. The benefit of this technique is evident in multiprocessor/multicore systems, where concurrency between quasi-independent threads allows the running time to be reduced proportionally to the number of processors.

However, even in uniprocessor systems, where parallelism is not real but simulated by means of time sharing, there are several advantages when employing multithreading. First, the management of the system resources is more efficient, because idle times can be reduced by suitably switching the CPU among CPU-bound and I/O-bound threads. Second, the interaction between the program and its user is fully supported, in such a way that no execution of previously started threads must be interrupted or completed before the user needs to interact with the program again. Third, the exchange of structured data among different threads is made easier by the use of shared memory and shared variables within the same program, without resorting to the more complicated data passing techniques typical of multiprocess software systems.

Multithreading offers a good level of flexibility but, on the other hand, more attention must be paid to synchronization issues as these are completely entrusted to the software developer. As a consequence, the software developer should be provided with a suitable support for being confident in the correctness of the way in which the thread synchronization is dealt with. This is especially important when the target is an efficient concurrent program, as this usually requires a complicated combination of several different synchronization techniques, like e.g. sleep and wakeup primitives, semaphores, and monitors.

1.4 Proposed Approach

In order to bridge the gap between system modeling and system implementation, in this thesis we propose an approach for automatically generating multithreaded programs from process algebraic architectural descriptions.

While at the architectural level we concentrate on PADL [13, 1, 12] due to its expressiveness, among the programming languages supporting multithreading we choose Java like in [52] for the following two reasons. First, Java offers a set of mechanisms for the well-structured management of threads and their shared data, which should simplify the code generation task. Second, its object-oriented nature – and specifically its encapsulation capability – makes Java an appropriate candidate for coping with the high level of abstraction typical of process algebraic architectural descriptions during code generation.

The proposed approach is divided into three phases, which will be illustrated by means of a running example based on the PADL description of an audio processing system. In the first phase we develop an architecture-driven technique for thread coordination management. Similarly to previous work (see, e.g., [58]), we advocate the provision of a suitable software package – which we shall call `Sync` – that takes care of the details of thread synchronization by means of architecture-inspired units, in a way that is completely automated and hence transparent to the software developer. Following the same architecture-centric spirit, the use of the package units should be guided by the architectural description of the systems to be developed, as this description is a well suited tool for achieving correct thread coordination in the case of concurrent object-oriented programs.

In the second phase we handle the translation of the algebraically-specified behavior of the individual software components into threads. The separation of thread synchronization management (first phase) from thread behavior generation turns out to be particularly appropriate in order to limit human intervention. In fact, while a completely automated and architecture-driven technique can guarantee correct thread coordination, only a partial translation based on stubs is possible for the generation of threads. Furthermore, we shall see that the preservation of the properties proved on the architectural description of the systems heavily depends on the way in which the stubs are filled in.

In the third phase we discuss the suitability of synthesizing monitors rather than threads from component descriptions that satisfy specific constraints. This overcomes some limitations with respect to [52], where only certain classes of process algebraic descriptions are considered, as well as with respect to the previous two phases, where only threads are taken into account. Since monitors reduce the thread context switch frequency, the synthesis of monitors can improve the performance of the generated code. Moreover the presence of monitors results in a lightweight concurrency control management with respect to package `Sync`, with the monitors themselves constituting explicit coordination areas that were not available in the previous two phases.

The whole approach, implemented as a `Sync`-based translator that we shall call PADL2Java, is finally integrated in TwoTowers [9]. This is a tool for the functional verification, security analysis, and performance evaluation of software architectures described in Æmilia [14, 6], a performance-oriented variant of PADL. The aim of this integration is to provide the software developer with a fully fledged environment for architectural design and code generation.

A further outcome of this research work is the enhancement of the expressiveness of PADL, in which non-synchronous communications have been integrated [12] together with the introduction of a new data type for abstracting and modeling objects and complex data structures in software systems. This results in an increase of the flexibility and effectiveness of the translator PADL2Java.

1.5 Structure of the Thesis

This thesis is organized as follows:

- In Chap. 2 we provide an overview of tools and approaches for the automatic generation of code and for the model transformation.
- In Chap. 3 we recall the process algebraic architectural description language PADL. In the last section of the chapter, the PADL description of an audio processing system is presented, which will be used as a running example throughout the rest of the thesis.

- In Chap. 4, 5, and 6 we illustrate the three phases of the approach. They are based, respectively, on a thread communication model, on a thread behavior model, and on a thread-monitor interaction model, which guide the automatic generation of code.
- In Chap. 7 we present the translator PADL2Java together with its integration in TwoTowers.
- In Chap. 8 we report on two case studies showing the application of the proposed approach to the synthesis of a video animation repainting system and of an implementation of a leader election algorithm.
- Finally, in Chap. 9 we conclude with a summary of results and some remarks about related and future work.

Chapter 2

Automatic Code Generation

In 1975, Fred Brooks pointed out the difference between intrinsic and accidental complexity [18]. Intrinsic complexity is how hard a problem really is. Such a complexity is inherent in the problem and cannot be eliminated by technological or methodological means. Accidental complexity is the unnecessary complexity introduced by a technology or method we adopt. For instance, building construction without using power tools. In our case, the accidental complexity could be the translation of designs into programs without the help of computers.

In other words, the intrinsic complexity of a system may be related to the problem-space, in which designs are expressed in terms of the application domain as, e.g., high-level abstraction models and descriptions. Accidental complexity instead may be related to the solution-space – where designs are realized in terms of the computing technologies domain – when implementation artifacts are manually written even if higher-level abstraction models are available.

Automatic code generation plays an important role to bridge the gap between the problem-space and the solution-space. As observed in [60], the ability to synthesize artifacts from models helps ensure the consistency between application implementations and analysis information associated with functional and QoS requirements captured by models. This automated transformation process is often referred to as “correct-by-construction”, as opposed to conventional handcrafted “construct-by-correction” software development processes that are tedious and error prone.

This chapter provides an overview of tools and approaches introduced in the last decades for automatic code generation, then presents our approach.

2.1 CASE Tools

Computer-Aided Software Engineering (CASE) methods and tools have been proposed from early 1980s in order to raise the level of abstraction in programming and to reduce the time for software development. Using the appropriate tool, a developer can easily express its designs in terms of graphical programming representations, such as state machines, dataflow, or entity-relationship diagrams. The main feature of CASE is to reduce the effort of manually coding, debugging, and porting programs by means of the synthesis of artifacts from their graphical representations.

Although CASE tools attracted considerable attention in the research community and trade literature, this approach wasn't widely adopted in practice [60, 50]. The reasons for this include:

- early CASE tools tried to generate entire applications, including the business logic and the software substrate, which led to inefficient, bloated code that was hard to optimize, validate, evolve, or integrate with code from other sources;

- due to the simplicity of the notations for representing behavior, CASE tools have largely been applicable to a few domains, such as telecom call processing, that map nicely onto state machines;

- due to the lack of powerful and common middleware, CASE tools targeted proprietary execution platforms, which made it hard to integrate the code they generated with other software languages and run-time environments.

However, in 1990s CASE evolved thanks also to the growing expressiveness and power of the target implementation languages – as the Object-Oriented C++ and Java – and of their associated libraries, programming techniques, and patterns [33]. This contributed to raise the level of abstraction of the generated code (see, e.g., [19]) and to shorten the gap between problem-space and solution-space. Unfortunately it was too late. As Martin Fowler said [29], ≪[. . .] *the term CASE has become a dirty word, and vendors try to avoid it now*≫.

2.2 Model-Driven Development and MDA

Model-Driven Development (MDD) [60] refers to a range of development approaches based on the use of software modeling as a primary form of expression. Mainly, the concept of a model is an abstraction of the target software system. Code is generated from the models, ranging from system skeletons to complete software artifacts.

The most popular form of MDD is known as Model-Driven Architecture (MDA) [53], a standard defined by the Object Management Group (OMG) [57]. MDA formalizes several concepts about the use of system models in the software development process. In MDA, a model serves as a prototype and as a proof-of-concept. Two types of models are defined by MDA at different levels of abstraction:

- Platform Independent Model (PIM) is a model that describes the target system without any details about the specifics of the implementation platform;
- Platform Specific Model (PSM) is a model that describes the target system on its intended platform, such as e.g. J2EE, .NET, SOAP.

The process of converting a PIM into a PSM is called model transformation. A PIM can be transformed into multiple PSMs, in order to enable several implementations of the original model on different platforms. In MDA, models are first-class artifacts, integrated into the development process through the chain of transformations from PIM through PSMs to coded applications.

A model, PIM or PSM, is written in a modeling language, and adheres to a meta-model, i.e., an explicit representation of the construct and rules to build semantic models. The OMG does not restrict MDA to any particular language, even though the modeling language must be well defined, which means that it must be precise to allow interpretation by a computer. In particular, Meta Object Facility (MOF)-based languages, such as Unified Modeling Language (UML) [64] or Common Warehouse MetaModel (CWM), are recommended by MDA for describing models. In order to define a standard for transformations

among models written in these languages, OMG has recently released the Query/Views/Transformations (QVT) specification [59].

Some tools have been created to be fully compliant with the QVT specification – see, e.g., SmartQVT [62]. Other transformation specifications, languages and frameworks have also been developed with the same purpose of QVT – see, e.g., the ATLAS Transformation Language (ATL) [47] and the VIsual Automated model TRAnsformations (VIATRA) framework [65].

The MDA based on UML seems currently to be the most promising approach for Model-Driven Development, but some criticisms have been made against it. For instance, as pointed out in [30], the numerous modeling concepts, poorly defined semantics, and lightweight extension mechanisms that UML provides, make learning and applying it in an MDD environment difficult.

2.3 Czarnecki-Helsen Classification of Model Transformation Approaches

In [25, 26], Krzysztof Czarnecki and Simon Helsen propose an interesting classification of the major categories of model transformation approaches. Three categories of transformations are distinguished first:

- **Text-to-model** or **code-to-model**. This category comprises parsing and reverse-engineering technologies.
- **Model-to-text** or **model-to-code**. The target of this transformation is just strings. More precisely, model-to-text approaches usually generate both code and non-code artifacts such as documents.
- **Model-to-model**. This transformation produces its target as an instance of the target meta-model.

Recalled that a meta-model is a precise definition of the constructs and rules needed for creating semantic models, transforming code to model can be viewed as a special case of model-to-model transformation. In this case, in fact, the source meta-model underlying the code is the grammar and the semantics of

the programming language in which the code is written. In the same way, transforming model to code can be viewed as a special case of model-to-model transformation, where a target meta-model is provided for the programming language in which the code has to be generated. As far as the more general model-to-model transformation is concerned, it is important to distinguish between "endogenous" transformations, in which the source and the target meta-models are the same, and "exogenous" transformations, where the two meta-models are different.

The Czarnecki-Helsen classification proceeds by analyzing the model-to-text and the model-to-model transformation approaches. The former is subdivided into:

- **Visitor-based approach**. It consists in providing some visitor mechanism to traverse the internal representation of a model and write text to a text stream.
- **Template-based approach**. In this approach a template is provided that usually consists of the target text containing splices of meta-code to access information from the source and to perform code selection and iterative expansion.

Compared with a visitor-based transformation, the structure of a template resembles more closely the code to be generated.

Model-to-model transformations, instead, are classified into seven different approaches:

- **Direct manipulation approach**. It consists in providing an internal model representation and some mechanisms to manipulate it. It is usually implemented as an object-oriented framework, which may also provide some minimal infrastructure to organize the transformations.
- **Structure-driven approach**. Approaches in this category have two distinct phases: the first phase is concerned with creating the hierarchical structure of the target model. The second phase sets the attributes and references in the target.

- **Operational approach**. It is similar to the direct manipulation approach, but offers more dedicated support for model transformation. Typically, the meta-modeling formalism is extended through a dedicated language in order to facilitate computations. An example would be to extend a query language such as the Object Constraint Language (OCL) with imperative constructs.

- **Template-based approach**. Model templates are models with embedded meta-code that compute the variable parts of the resulting template instances. Model templates are usually expressed in the concrete syntax of the target language, which helps the developer to predict the result of template instantiation. The meta-code can have the form of annotations on model elements. Typical annotations are conditions, iterations, and expressions, all being part of the meta-language.

- **Relational approach**. This category groups declarative approaches in which the main concept is mathematical relations. In general, relational approaches can be seen as a form of constraint solving.

- **Graph-transformation-based approach**. This category of model transformation approaches draws on the theoretical work on graph transformations. In particular, this category operates on typed, attributed, labeled graphs, which can be thought of as formal representations of simplified class models.

- **Hybrid approach**. Hybrid approaches combine different techniques from the previous categories. The different approaches can be combined as separate components or, in a more fine-grained fashion, at the level of individual rules. QVT is an example of a hybrid approach with three separate components, namely Relations, Operational mappings, and Core. An Example of the fine-grained combination is ATL.

2.4 Adopted Model Transformation Approach

The idea at the basis of the first phase of our approach is the provision of a library of software components, i.e. the Java package `Sync`, for adding architectural capabilities to the (target) implementation language in order to shorten the gap with the architectural description language, i.e. PADL, from which the code will be generated. Hence, the whole approach proposed in this thesis can be viewed as a "semi-exogenous" transformation, as opposed to the "semi-endogenous" approach discussed in 1.2.

On the basis of the Czarnecki-Helsen classification, now it can be said that if we consider the target model as text, a "semi-endogenous" approach can be easily tackled by implementing a model-to-text/template-based transformation. Unfortunately, this transformation cannot be applied with the same facility to our "semi-exogenous" approach, as the distance from model to text is still long.

However, thanks to the presence of the package `Sync`, a model-to-text/visitor-based transformation can be implemented very easily in our work. We have to take into account also that the generated code will not be verbose, as the presence of suitable architecture-oriented software components in the target meta-model severely reduces code redundancy.

As we will see in Sect. 7.1, all the three phases of our approach actually use a simple model-to-text/visitor-based transformation for generating code. Before code generation, the third phase also uses a model-to-model/direct manipulation approach for obtaining endogenous transformations of PADL descriptions. This choice has been done because the internal representation of the model is made available by the parser, which is used for reading PADL (textual) descriptions.

Chapter 3

The Architectural Description Language PADL

The process algebraic description language we choose for modeling concurrent software systems is PADL [13, 1]. Our choice is motivated by the expressiveness of the language, which has recently been enhanced thanks also to the contribution of the research work included in this thesis, in particular for what concerns asynchronous and semi-synchronous communications [12].

In this chapter some basic notions of process algebra are briefly recalled (Sect. 3.1) before introducing the textual and graphical notations of PADL (Sect. 3.2), followed by its formal semantics (Sect. 3.3). The semantic treatment of the newly introduced non-synchronous communications is then discussed (Sect. 3.4). In the last section of the chapter (Sect. 3.5) the PADL description of an audio processing system is presented, which will be used as a running example throughout the rest of the thesis.

3.1 Process Algebra

Process algebra [56, 40, 7] provides a set of operators by means of which the behavior of a system can be described in an action-based, compositional way. Given a set *Name* of action names including τ for invisible actions, we will consider a process algebra PA whose syntax is illustrated in Table 3.1.

Standard operational semantic rules map every closed and guarded process term P of PA to a state-transition graph $[\![P]\!]$ called labeled transition system. In

P ::=	$\underline{0}$	inactive process	
$\mid$	B	process constant	$(B \triangleq P)$
$\mid$	$a.P$	action prefix	$(a \in \mathit{Name})$
$\mid$	$P + P$	alternative composition	
$\mid$	$P \parallel_S P$	parallel composition	$(S \subseteq \mathit{Name} - \{\tau\})$
$\mid$	P/H	hiding	$(H \subseteq \mathit{Name} - \{\tau\})$
$\mid$	$P \backslash L$	restriction	$(L \subseteq \mathit{Name} - \{\tau\})$
$\mid$	$P[\varphi]$	relabeling	$(\varphi : \mathit{Name} \rightarrow \mathit{Name},\ \varphi^{-1}(\tau) = \{\tau\})$

Table 3.1. Process term syntax for PA

this graph the states correspond to the process terms derivable from P, the initial state corresponds to P, and each transition is labeled with the corresponding action. Observed that $[\![\underline{0}]\!]$ is a single-state graph with no transitions, we have one basic rule for action prefix and several inductive rules for the other operators:

- $a.P$ can execute an action with name a and then behaves as P:

$$a.P \xrightarrow{a} P$$

- B behaves as the process term occurring in its defining equation:

$$\frac{B \triangleq P \quad P \xrightarrow{a} P'}{B \xrightarrow{a} P'}$$

- $P_1 + P_2$ behaves as either P_1 or P_2 depending on which of them executes an action first (nondeterministic choice):

$$\frac{P_1 \xrightarrow{a} P'}{P_1 + P_2 \xrightarrow{a} P'} \qquad \frac{P_2 \xrightarrow{a} P'}{P_1 + P_2 \xrightarrow{a} P'}$$

- $P_1 \parallel_S P_2$ behaves as P_1 in parallel with P_2 as long as actions are executed whose name does not belong to S:

$$\frac{P_1 \xrightarrow{a} P_1' \quad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1' \parallel_S P_2} \qquad \frac{P_2 \xrightarrow{a} P_2' \quad a \notin S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1 \parallel_S P_2'}$$

Synchronizations are forced between any action executed by P_1 and any action executed by P_2 that have the same name belonging to S:

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{a} P_2' \quad a \in S}{P_1 \parallel_S P_2 \xrightarrow{a} P_1' \parallel_S P_2'}$$

- P/H behaves as P with all the executed actions occurring in H being made invisible:

$$\frac{P \xrightarrow{a} P' \quad a \in H}{P/H \xrightarrow{\tau} P'/H} \qquad \frac{P \xrightarrow{a} P' \quad a \notin H}{P/H \xrightarrow{a} P'/H}$$

- $P \backslash L$ behaves as P with all the actions occurring in L being made disabled:

$$\frac{P \xrightarrow{a} P' \quad a \notin L}{P \backslash L \xrightarrow{a} P' \backslash L}$$

- $P[\varphi]$ behaves as P with all the executed actions being relabeled via φ:

$$\frac{P \xrightarrow{a} P'}{P[\varphi] \xrightarrow{\varphi(a)} P'[\varphi]}$$

Behavioral equivalences [37] are a formal means to compare and manipulate process terms – possibly abstracting from invisible actions – in a way that preserves certain behavioral properties. Among the various equivalences, for PA we consider weak bisimilarity [56], according to which two process terms are equivalent if they are able to mimic each other's visible behavior stepwise.

Definition 3.1 *A relation $\mathcal{R}$ is a weak bisimulation iff for all $(P_1, P_2) \in \mathcal{R}$ and $a \in Name - \{\tau\}$:*

- *whenever $P_1 \xrightarrow{a} P_1'$, then $P_2 \xRightarrow{\tau^* a \tau^*} P_2'$ with $(P_1', P_2') \in \mathcal{R}$;*
- *whenever $P_1 \xrightarrow{\tau} P_1'$, then $P_2 \xRightarrow{\tau^*} P_2'$ with $(P_1', P_2') \in \mathcal{R}$;*
- *whenever $P_2 \xrightarrow{a} P_2'$, then $P_1 \xRightarrow{\tau^* a \tau^*} P_1'$ with $(P_1', P_2') \in \mathcal{R}$;*
- *whenever $P_2 \xrightarrow{\tau} P_2'$, then $P_1 \xRightarrow{\tau^*} P_1'$ with $(P_1', P_2') \in \mathcal{R}$.*

Weak bisimilarity $\approx_{\mathrm{B}}$ is the union of all the weak bisimulations. ■

3.2 PADL Descriptions and Notations

PADL [13, 1, 12] is a process algebraic architectural description language. A PADL description represents an architectural type, which is an intermediate abstraction between a single system and an architectural style [61]. An architectural type consists of a family of software systems sharing certain constraints on the observable behavior of their components as well as on their topology. PADL descriptions can be expressed using either a textual or a graphical notation.

Before giving an overview of the two notations, it must be said that a complete specification of the textual one can be found in [9]. However, the PADL description of the running example presented in Sect. 3.5, as well as the PADL descriptions of the two case studies proposed in Sect. 8.1 and 8.2, are well commented and allow one to understand the whole syntax (both textual and graphical) of PADL.

3.2.1 PADL Textual Notation

As shown in Table 3.2, the textual description of an architectural type in PADL starts with the name and the formal parameters of the architectural type and is composed of three sections.

The first section defines the overall behavior of the system family by means of types of software components and connectors, which are collectively called architectural element types. The definition of an AET starts with its name and its formal parameters and consists of the specification of its behavior and its interactions. The behavior has to be provided in the form of a sequence of behavioral equations written in a verbose variant of PA allowing only for the inactive process (rendered as `stop`), the action prefix operator, the alternative composition operator (rendered as `choice`), and recursion. The available data types are boolean, bounded/unbounded integer, real, list, array, record, and generic object[1].

[1] In the semantic model of PADL, data types having infinite domain, as unbounded integer, real, list, and array/record containing data types having infinite domain, are treated with symbolic value passing techniques [8]. The generic object data type, instead, can assume only

ARCHI_TYPE	◁*name and initialized formal data parameters*▷
ARCHI_BEHAVIOR	
⋮	⋮
ARCHI_ELEM_TYPE	◁*AET name and formal data parameters*▷
BEHAVIOR	◁*sequence of process algebraic defining equations built from stop, action prefix, choice, and recursion*▷
INPUT_INTERACTIONS	◁*input synchronous/semi-synchronous/asynchronous uni/and/or-interactions*▷
OUTPUT_INTERACTIONS	◁*output synchronous/semi-synchronous/asynchronous uni/and/or-interactions*▷
⋮	⋮
ARCHI_TOPOLOGY	
ARCHI_ELEM_INSTANCES	◁*AEI names and actual data parameters*▷
ARCHI_INTERACTIONS	◁*architecture-level AEI interactions*▷
ARCHI_ATTACHMENTS	◁*attachments between AEI local interactions*▷
[BEHAV_MODIFICATIONS	
[BEHAV_HIDINGS	◁*names of actions to be hidden*▷]
[BEHAV_RESTRICTIONS	◁*names of actions to be restricted*▷]
[BEHAV_RENAMINGS	◁*names of actions to be changed*▷]]
END	

Table 3.2. Structure of a PADL textual description

The interactions are those actions occurring in the process algebraic specification of the behavior that act as interfaces for the AET, while all the other actions are assumed to represent internal activities. Each interaction has to be equipped with three qualifiers. The first one establishes whether the interaction is an input or output interaction.

The second qualifier establishes the degree of synchronicity of the interaction. We distinguish among synchronous interactions, semi-synchronous interactions

two values, i.e., `null` or not `null`, hence it is treated, as the boolean and the bounded integer data types, with concrete value passing techniques for finite domains.

(which cause no blocking as they raise an exception if prevented), and asynchronous interactions. The related keywords are `SYNC` (default value), `SSYNC`, and `ASYNC`, respectively.

The third qualifier describes the multiplicity of the communications in which the interaction can be involved. We distinguish among uni/and/or-interactions giving rise to one-to-one, inclusive one-to-many (broadcast-like) and selective one-to-many (server-clients-like) communications. The related keywords are `UNI`, `AND`, and `OR`, respectively. It can also be specified that an output or-interaction depends on an input or-interaction through keyword `DEP` (or-dependencies).

The second section of the textual description defines the architectural topology. This is accomplished in three steps. First we have the declaration of the instances of the AETs – called AEIs – which represent the actual system components and connectors, together with their actual parameters. Then we have the declaration of the architectural (as opposed to local) interactions, which are some of the interactions of the AEIs that act as interfaces for the whole systems of the family. Finally we have the declaration of the architectural attachments among the local interactions of the AEIs, which make the AEIs communicate with each other. An attachment is admissible only if it goes from an output interaction of an AEI to an input interaction of another AEI. Moreover, a uni-interaction can be attached only to one interaction, whereas an and/or-interaction can be attached only to uni-interactions.

A suitable iterative mechanism is available to declare several AEIs of the same type concisely. Each such AEI can then be identified via an index. The same iterative mechanism can be exploited when declaring architectural interactions and attachments involving the AEIs under consideration. From a different viewpoint, this iterative mechanism results in controlled topological variations within the architectural type, like changing the number of AEIs in a ring-like topology or the number of AEIs connected to an and/or-interaction.

The third section, which is optional, defines some variations of the observable behavior of the system family. This is accomplished by declaring some actions occurring in the behavior of certain AEIs to be unobservable, prevented from occurring, or renamed, respectively. This is useful for conducting certain kinds of analysis.

3.2.2 PADL Graphical Notation

Besides the textual notation, PADL comes equipped with a graphical notation that is an extension of the flow graph notation [56]. In an extended flow graph the AEIs are depicted as boxes, the local (resp. architectural) interactions are depicted as small black circles (resp. white squares) on the box border, and the attachments are depicted as directed edges between pairs each composed of a local output interaction and a local input interaction. The small circle/square of a non-synchronous interaction is extended with an arc or a buffer inside the AEI box if the interaction is semi-synchronous or asynchronous, respectively. Finally, the small circle/square of an interaction is extended with a triangle or a bisected triangle outside the AEI box if the interaction is an and-interaction or an or-interaction, respectively.

3.3 The semantics for PADL

The semantics for PADL is given by translation into PA. The meaning of a PADL description is a process term stemming from the parallel composition of the process algebraic specifications of its AEIs, with the synchronization sets being determined by the attachments. This process term is then subject to the possibly declared behavioral variations, which are rendered through the hiding operator, the restriction operator, and the relabeling operator, respectively. In this section we consider only synchronous interactions.

Let $\mathcal{C}$ be an AET with formal parameters $fp_1, \ldots, fp_m$ and behavior given by a sequence $\mathcal{E}$ of process algebraic defining equations containing only dynamic operators. Let C be an AEI of type $\mathcal{C}$ with actual data parameters $ap_1, \ldots, ap_m$. Then the semantics of C in isolation is defined as follows:

$$\boxed{[\![C]\!] \;=\; \mathit{or\text{-}rewrite}_{\emptyset}(\mathcal{E}\{ap_1/fp_1, \ldots, ap_m/fp_m\})}$$

where $\{_/_, \ldots, _/_\}$ denotes a syntactical substitution and function `or-rewrite` replaces each or-interaction with a choice among as many fresh uni-interaction as there are attachments involving the or-interaction.

More precisely, all the or-interactions are rewritten within the body of any defining equation occurring in an AET definition, with or-dependencies managed by keeping track of the fresh input uni-interactions that are currently in force via $A \subseteq \mathit{Name}$ (initially $\emptyset$).

If a is an input or-interaction on which an output or-interaction depends and it is involved in k attachments, we have:

$$\boxed{\begin{array}{rcl} \mathit{or\text{-}rewrite}_A(a.P) & = & \texttt{choice}\{a_1.\mathit{or\text{-}rewrite}_{A \cup \{a_1\} - \{a_j \mid 1 \le j \le k \wedge j \neq 1\}}(P), \\ & & \vdots \\ & & a_k.\mathit{or\text{-}rewrite}_{A \cup \{a_k\} - \{a_j \mid 1 \le j \le k \wedge j \neq k\}}(P)\} \end{array}}$$

If b is an output or-interaction depending on the input or-interaction a and $a_i \in A$, we have:

$$\boxed{\mathit{or\text{-}rewrite}_A(b.P) \; = \; b_i.\mathit{or\text{-}rewrite}_A(P)}$$

If a is an input or-interaction on which no output or-interaction depends or an output or-interaction not depending on any input or-interaction and it is involved in k attachments, we have:

$$\boxed{\begin{array}{rcl} \mathit{or\text{-}rewrite}_A(a.P) & = & \texttt{choice}\{a_1.\mathit{or\text{-}rewrite}_A(P), \\ & & \vdots \\ & & a_k.\mathit{or\text{-}rewrite}_A(P)\} \end{array}}$$

If a is a uni-/and-interaction or an internal action:

$$\boxed{\mathit{or\text{-}rewrite}_A(a.P) \; = \; a.\mathit{or\text{-}rewrite}_A(P)}$$

For the remaining operators, the rewriting process works as follows:

$$\boxed{\begin{array}{rcl} \mathit{or\text{-}rewrite}_A(\texttt{stop}) & = & \texttt{stop} \\ \mathit{or\text{-}rewrite}_A(B(\mathit{actual_par_list})) & = & B_A(\mathit{actual_par_list}) \\ \mathit{or\text{-}rewrite}_A(\texttt{choice}\{P_1, \dots, P_n\}) & = & \texttt{choice}\{\mathit{or\text{-}rewrite}_A(P_1), \\ & & \vdots \\ & & \mathit{or\text{-}rewrite}_A(P_n)\} \end{array}}$$

where $B_\emptyset \equiv B$, while for $A \neq \emptyset$:

$$\boxed{B_A(\mathit{formal_par_list};\ \mathit{local_var_list}) \; = \; \mathit{or\text{-}rewrite}_A(P)}$$

if $B(\mathit{formal_par_list};\ \mathit{local_var_list}) \; = \; P$.

Consider now the AEIs $C_1, \ldots, C_n$ and let us denote by $\mathcal{LI}_{C_i}$ the set of local interactions of C_i and by $\mathcal{LI}_{C_i;C_1,\ldots,C_n}$ the set of local interactions of C_i that are attached to $C_1, \ldots, C_n$:

$$\boxed{\mathcal{LI}_{C_i;C_1,\ldots,C_n} \subseteq \mathcal{LI}_{C_i}}$$

Those interactions need to be rewritten so that each C_i can communicate with the other AEIs despite the fact that attached interactions may have been given different names. Then suitable synchronization sets need to be constructed based on the attachments that contain the renamed interactions. Dot notation and name concatenation are used to ensure renaming uniqueness ($C_i.a_1\#C_j.a_2$ if interaction a_1 of C_i attached to interaction a_2 of C_j).

More precisely, we introduce as many fresh actions as there are maximal sets of attached local interactions in $C_1, \ldots, C_n$:

$$\boxed{\mathcal{S}(C_1, \ldots, C_n)}$$

together with an injective relabeling function for each $\mathcal{LI}_{C_i;C_1,\ldots,C_n}$:

$$\boxed{\varphi_{C_i;C_1,\ldots,C_n} : \mathcal{LI}_{C_i;C_1,\ldots,C_n} \longrightarrow \mathcal{S}(C_1, \ldots, C_n)}$$

such that:

$$\boxed{\varphi_{C_i;C_1,\ldots,C_n}(a_1) \;=\; \varphi_{C_j;C_1,\ldots,C_n}(a_2)}$$

iff $C_i.a_1$ and $C_j.a_2$ are attached to each other or to the same and-interaction.

The interacting semantics of C_i with respect to $C_1, \ldots, C_n$ is defined as follows:

$$\boxed{[\![C_i]\!]_{C_1,\ldots,C_n} \;=\; [\![C_i]\!][\varphi_{C_i;C_1,\ldots,C_n}]}$$

Afterwards, we introduce individual synchronization set of C_i with respect to $C_1, \ldots, C_n$:

$$\boxed{\mathcal{S}(C_i; C_1, \ldots, C_n) \;=\; \varphi_{C_i;C_1,\ldots,C_n}(\mathcal{LI}_{C_i;C_1,\ldots,C_n})}$$

as well as the pairwise synchronization set of C_i and C_j with respect to $C_1, \ldots, C_n$:

$$\boxed{\mathcal{S}(C_i, C_j; C_1, \ldots, C_n) \;=\; \mathcal{S}(C_i; C_1, \ldots, C_n) \cap \mathcal{S}(C_j; C_1, \ldots, C_n)}$$

The interacting semantics of $C'_1, \ldots, C'_{n'}$ with respect to $C_1, \ldots, C_n$ is defined as follows:

$$\begin{array}{rcl} [\![C'_1, \ldots, C'_{n'}]\!]_{C_1,\ldots,C_n} & = & [\![C'_1]\!]_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C'_1,C'_2;C_1,\ldots,C_n)} \\ & & [\![C'_2]\!]_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C'_1,C'_3;C_1,\ldots,C_n) \cup \mathcal{S}(C'_2,C'_3;C_1,\ldots,C_n)} \\ & & \ldots \parallel_{\bigcup_{i=1}^{n'-1} \mathcal{S}(C'_i,C'_{n'};C_1,\ldots,C_n)} [\![C'_{n'}]\!]_{C_1,\ldots,C_n} \end{array}$$

where left associativity of parallel composition is assumed.

Let $\mathcal{A}$ be an architectural description, let $C_1, \ldots, C_n$ be all of its AEIs, and H, L, and φ be possible behavioral variations enforcing hidden actions, restricted actions, and action renamings, respectively. Then, the semantics of $\mathcal{A}$ before behavioral variations is defined as follows:

$$[\![\mathcal{A}]\!]_{\mathrm{bbv}} = [\![C_1, \ldots, C_n]\!]_{C_1,\ldots,C_n}$$

whereas the semantics of $\mathcal{A}$ after behavioral variations is defined as follows:

$$[\![\mathcal{A}]\!]_{\mathrm{abv}} = [\![\mathcal{A}]\!]_{\mathrm{bbv}} \,/\, H \setminus L\, [\varphi]$$

3.4 Handling Non-Synchronous Interactions

In this section we discuss the semantic treatment of the newly introduced semi-synchronous and asynchronous interactions.

Given an output interaction o of an AEI C_1 attached to an input interaction i of an AEI C_2, nine different forms of communication can be established between them, as graphically described in the left-hand side part of Fig. 3.1. The first form of communication in the figure was the only one available in PADL before introducing non-synchronous interactions.

A semi-synchronous interaction s executed by an AEI C of an architectural type $\mathcal{A}$ gives rise to a transition labeled with s within $[\![C]\!]$. In the context of $[\![\mathcal{A}]\!]$ this transition is relabeled with *s_exception* if s cannot be immediately executed.

More formally, suppose that in $\mathcal{A}$ the synchronous output interaction o of an AEI C_1 is attached to the semi-synchronous input interaction i of an AEI C_2, which is the second form of communication depicted in Fig. 3.1. Let us denote by $o\#i$ the fresh action name associated with the synchronization of o with i.

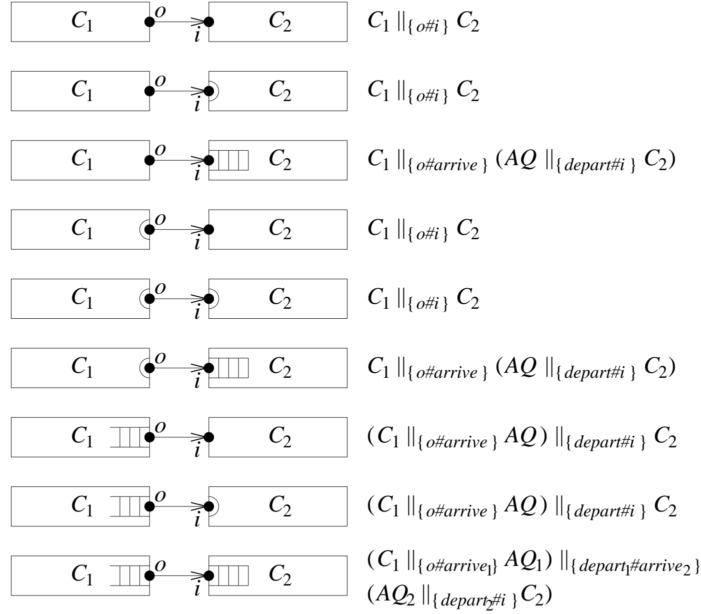

Figure 3.1. Synchronous, semi-synchronous and asynchronous communications

Then the communication of C_1 with C_2 is described by the following two semantic rules, where P_1 (resp. P_2) is the process term representing the current state of $[\![C_1]\!]_{C_1,C_2}$ (resp. $[\![C_2]\!]_{C_1,C_2}$) and $S = \mathcal{S}(C_1, C_2; C_1, C_2)$:

$$\frac{P_1 \xrightarrow{o} P_1' \qquad P_2 \xrightarrow{i} P_2'}{P_1 \parallel_S P_2 \xrightarrow{o\#i} P_1' \parallel_S P_2'} \qquad \frac{P_1 \not\xrightarrow{o} P_1' \qquad P_2 \xrightarrow{i} P_2'}{P_1 \parallel_S P_2 \xrightarrow{i_exception} P_1 \parallel_S P_2'}$$

In the symmetric case of a semi-synchronous output interaction attached to a synchronous input interaction – which corresponds to the fourth form of communication depicted in Fig. 3.1 – we have the following two rules:

$$\frac{P_1 \xrightarrow{o} P_1' \qquad P_2 \xrightarrow{i} P_2'}{P_1 \parallel_S P_2 \xrightarrow{o\#i} P_1' \parallel_S P_2'} \qquad \frac{P_1 \xrightarrow{o} P_1' \qquad P_2 \not\xrightarrow{i} P_2'}{P_1 \parallel_S P_2 \xrightarrow{o_exception} P_1' \parallel_S P_2}$$

Finally, in the case in which both o and i are semi-synchronous – i.e. the fifth form of communication depicted in Fig. 3.1 – we have all the previous semantic

rules together:

$$\frac{P_1 \xrightarrow{o} P_1' \qquad P_2 \xrightarrow{i} P_2'}{P_1 \parallel_S P_2 \xrightarrow{o\#i} P_1' \parallel_S P_2'}$$

$$\frac{P_1 \stackrel{o}{\not\longrightarrow} P_1' \qquad P_2 \xrightarrow{i} P_2'}{P_1 \parallel_S P_2 \xrightarrow{i_exception} P_1 \parallel_S P_2'} \qquad \frac{P_1 \xrightarrow{o} P_1' \qquad P_2 \stackrel{i}{\not\longrightarrow} P_2'}{P_1 \parallel_S P_2 \xrightarrow{o_exception} P_1' \parallel_S P_2}$$

While semi-synchronous input/output interactions are dealt with by means of suitable semantic rules, asynchronous input/output interactions need a different treatment because of the decoupling between the beginning and the end of the communications in which those interactions are involved.

For each asynchronous interaction we have to introduce an additional implicit AEI that behaves as an unbounded buffer, as shown in the third, sixth, seventh, eighth, and ninth form of communication depicted in Fig. 3.1. This AEI is of the following type:

```
ARCHI_ELEM_TYPE       Async_Queue(void)

  BEHAVIOR            Queue(int i := 0; void) =
                        choice
                        {
                          arrive . Queue(i + 1),
                          cond(i > 0) -> depart . Queue(i - 1),
                        }

  INPUT_INTERACTIONS  --- arrive
  OUTPUT_INTERACTIONS --- depart
```

where `arrive` is an always-available synchronous interaction, whereas `depart` is a synchronous interaction enabled only if the buffer is not empty.

In the case of an asynchronous output interaction o, `arrive` is attached to o and `depart` is attached to the input interactions originally attached to o. Moreover, o is implicitly converted into a synchronous uni-interaction, `arrive` is declared as a uni-interaction, and `depart` is declared as a uni/and/or-interaction depending on whether o was a uni/and/or-interaction, respectively.

The case of an asynchronous input interaction i is symmetrical. More precisely, `arrive` is attached to the output interactions originally attached to

i and `depart` is attached to i. Furthermore, i is implicitly converted into a semi-synchronous uni-interaction, `arrive` is declared as a uni/and/or-interaction depending on whether i was a uni/and/or-interaction, respectively, and `depart` is declared as a uni-interaction. Note that i becomes a semi-synchronous interaction because the communications between `depart` and i must not block the AEI executing i whenever the buffer is empty. Thus i is subject to the first group of semantic rules defined in this section.

Due to the way non-synchronous interaction have been handled, we only need to revise the definition of the semantics of an AEI, while all the other definitions in Sect. 3.3 are unchanged. More precisely, we only have to take into account the possible presence of additional implicit AEIs acting as unbounded buffers for asynchronous input/output interactions.

Let $\mathcal{C}$ be an AET with formal parameters $fp_1, \ldots, fp_m$ and behavior given by a sequence $\mathcal{E}$ of process algebraic defining equations. Let C be an AEI of type $\mathcal{C}$ with actual parameters $ap_1, \ldots, ap_m$. If C has $h \in \mathbb{N}_{>0}$ asynchronous input interactions $i_1, \ldots, i_h$ – handled through the related additional implicit AEIs $AQ_1, \ldots, AQ_h$ – and $k \in \mathbb{N}_{>0}$ asynchronous output interactions $o_1, \ldots, o_k$ – handled through the related additional implicit AEIs $AQ'_1, \ldots, AQ'_k$ – then the semantics of C is defined as follows:

$$\begin{array}{rcl} [\![C]\!] & = & (AQ_1 \,\|_\emptyset \ldots \|_\emptyset\, AQ_h) \,\|_{\{depart_1 \# i_1, \ldots, depart_h \# i_h\}} \\ & & \quad or\text{-}rewrite_\emptyset(\mathcal{E}\{ap_1/fp_1, \ldots, ap_m/fp_m\}) \\ & & \qquad \|_{\{o_1 \# arrive'_1, \ldots, o_k \# arrive'_k\}} (AQ'_1 \,\|_\emptyset \ldots \|_\emptyset\, AQ'_k) \end{array}$$

3.5 Running Example: An Audio Processing System

In this section we provide the PADL description of an audio processing system. This will be used throughout the thesis as a running example to illustrate the various phases of our approach to the synthesis of multithreaded Java programs from process algebraic architectural descriptions.

3.5.1 Informal Specification of the Audio Processing System

The audio processing system has to acquire and play a digital audio stream, allowing the user to change in real time the sound effects applied to the stream. More precisely, a dry audio stream, coming from an input audio device, can be modified by a software sound processor according to some effect required by the user and synthesized by an effect generator. Then, the processed audio stream has to be forwarded to an output audio device, which plays it out. Both audio streams are sequences of audio samples. In order to avoid frequent accesses to the audio devices and to allow the sound processor to execute complex operations – like filtering or convolution – that in general require several audio samples, the sequences of audio samples have to be segmented.

The sound processor and the effect generator are specified as two different (and concurrent) entities because the generation of some effects – like high-order equalization filters – may require a large amount of time that, in order to respect real-time constraints and deadlines, should not be debited to the same entity that executes the segment processing.

The audio stream processing can be controlled from outside through a graphical user interface or a peripheral device – like a mixer or a pedal board – endowed with an appropriate controller, or event handler, that interacts with the system console. In particular, the user/controller can start and stop the process and change the sound effect at audio processing time.

As shown in Fig. 3.2, the considered system is made out of five software components: the console `C`, the input audio device driver `IADD`, the sound processor `SP`, the effect generator `EG`, and the output audio device driver `OADD`.

Since the five software components work as autonomous entities that communicate to each other from time to time, it is natural to implement each of them as a thread or as a monitor. More precisely, the sound processor and the effect generator should result in two CPU-bound threads, while the two audio device drivers should result in two I/O-bound threads. The console may also result in an I/O-bound thread. However, since its purpose is simply to mediate

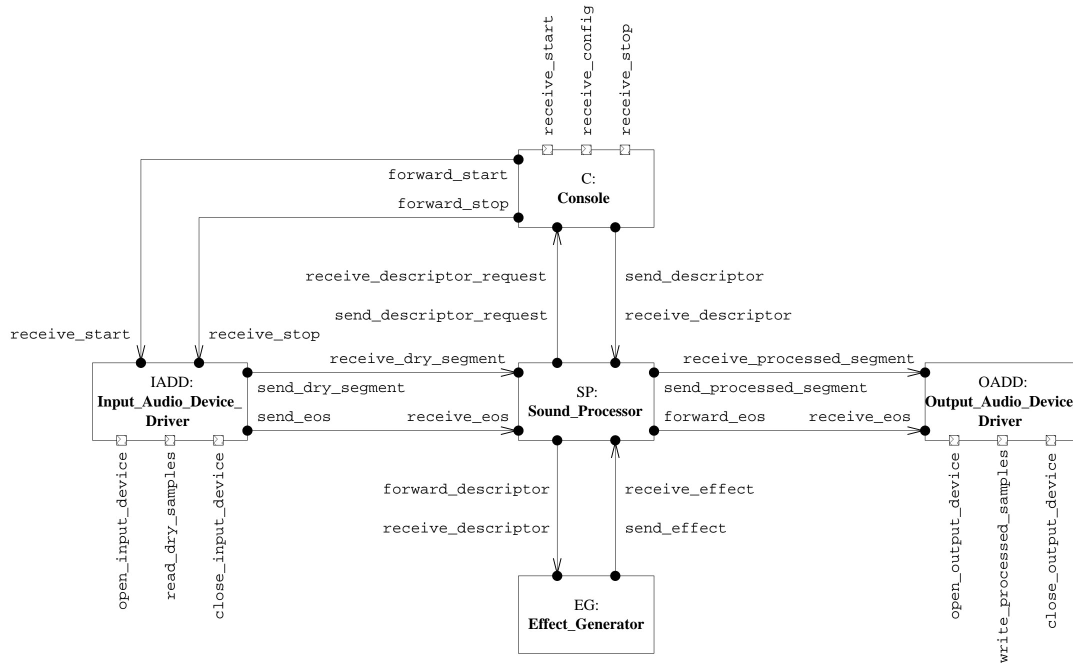

Figure 3.2. Extended flow graph of the audio processing system

the communication among the user/controller – which usually produces a low-rate sequence of events – and the above mentioned threads, it could also be implemented as a monitor.

As far as the communication topology is concerned, only the console can interact with the user/controller by receiving commands related to the start or the stop of the audio processing (`receive_start` and `receive_stop`) or the effect to be applied to the input audio stream, described as a configuration change (`receive_config`). Likewise, only the two audio device drivers can interact with the two audio devices by exchanging audio samples (`open_input_device`, `read_dry_samples`, and `close_input_device` for the `IADD`; `open_output_device`, `write_processed_samples`, and `close_output_device` for the `OADD`).

The console forwards the start and stop signals to the `IADD` (`forward_start` and `forward_stop`), while a descriptor of the desired effect, which summarizes the last configuration changes, is sent to the sound processor (`send_descriptor`) after receiving a request from it (`receive_descriptor_request`). The sound processor forwards descriptors (`forward_descriptor`) to the effect generator

and, when ready, receives from it an effect (`receive_effect`). The `IADD` sends segments of dry audio samples to the sound processor (`send_dry_segment`) and propagates to it the stop signals via end-of-stream signals (`send_eos`). The sound processor in turn sends segments of processed audio samples to the `OADD` (`send_processed_segment`) and propagates to it the stop signals via end-of-stream signals (`forward_eos`).

In order to guarantee the quality of the played audio as well as the correct application of the effects, the five software components must be kept well synchronized. This can be achieved by making each of the two audio device drivers use a standard audio port software utility, like e.g. the Java Media Framework, implementing an internal buffer.

When the input audio device starts, the input buffer has to grow according to the sampling frequency of the input audio device. An input instruction will be used by the `IADD` to read from the input buffer an audio segment composed of a given number of dry samples. This input instruction will block if it requires an audio segment that is longer than the number of unread dry audio samples contained in the input buffer. If no input is requested for a long time, the input buffer will saturate and the oldest unread dry audio samples will be lost due to overwriting.

Conversely, the output buffer has to decrease according to the sampling frequency of the output audio device, which will be fed by the `OADD` through an output instruction that appends to the output buffer an audio segment composed of a certain number of processed audio samples. This output instruction will block if it tries to write an audio segment that is longer than the residual capacity of the output buffer. On the other hand, the output device driver can play out only if sufficiently many processed audio samples are present in the output buffer.

Since the two instructions for interacting with the two audio devices are blocking, the two threads that will implement the two audio device drivers must be kept synchronized at the same sampling frequency, and this synchronization must not to be disrupted by the sound processor – when applying an effect – or by the user/controller via the console – when changing the effect. The synchronization of the two threads is possible if the segment processing time – i.e. the time to apply an effect to a segment plus other operations like handling

descriptors and receiving new effects – is less than the segment playout time (this is an obvious real-time constraint). Moreover, it is demanded that the OADD introduces an artificial delay before playing out the first received audio segment, in order to compensate for a possible increase of the segment processing time incurred by subsequent audio segments to be played out.

3.5.2 PADL Description of the Audio Processing System

The PADL textual description of the audio processing system introduces first of all three parameters: the number of samples per segment (`segment_size`), the artificial delay (`delay`) that will be inserted by the output audio device driver, and the maximum number of configuration changes (`allowed_changes`) that can be received by the console between the processing of two contiguous segments. Here is the PADL description header:

```
ARCHI_TYPE Audio_Processing_System(const integer segment_size    := 1024,  % samples/segment
                                   const integer delay           :=  125,  % msec
                                   const integer allowed_changes :=    3)
```

In the first section of the PADL textual description we specify five AETs. The first of them, `Console`, has a single parameter, as shown in its header:

```
ARCHI_ELEM_TYPE Console(const integer allowed_changes)
```

This AET is defined by means of three behavioral equations: `Start`, `Config_Handling`, and `Descriptor_Handling`. The first equation simply waits for receiving the start command from the user/controller via the input interaction `receive_start`, then forwards the start command to the input audio device driver through the output interaction `forward_start`, and finally invokes the second equation:

```
Start(integer(0..allowed_changes) config_changes := 0;
      void) =
  receive_start . forward_start . Config_Handling(config_changes);
```

The AET parameter `allowed_changes` is used in the second equation to prevent what is called "effect bombing", i.e. a user/controller that transmits too many configurations to the console in a short period of time:

```
Config_Handling(      integer(0..allowed_changes) config_changes;
                local object(Configuration)       console_config) =
  choice
  {
    cond(config_changes < allowed_changes) ->
      receive_config?(console_config) . store_config!(console_config) .
        Config_Handling(config_changes + 1),
    choice
    {
      cond(config_changes > 0) ->
        receive_descriptor_request . Descriptor_Handling(),
      cond(config_changes = 0) ->
        receive_descriptor_request . send_descriptor!(null) . Config_Handling(0)
    },
    receive_stop . forward_stop . Start(config_changes)
  };
```

In fact, in the second equation the input interaction `receive_config` can take place only when the number of configurations `config_changes` received after the first time that the console starts, or after the last communication with the sound processor, is less than `allowed_changes`. This condition is checked within the first branch of the choice building up the second equation. The interaction `receive_config` receives an object `console_config` of type `Configuration`, which is then employed by the internal action `store_config` (symbol "?" is used for input action parameters, while "!" is used for output action parameters).

The second branch of the second equation is a nested choice. Its first branch waits for a descriptor request, then invokes the third equation. However, when no configuration has been received after the first start or after the previous communication with the sound processor, the second branch of the nested choice sends an object `null` to the sound processor, in order to mean that no new effect has to be applied to the stream.

The third branch of the second equation is not guarded by any condition, and it allows a stop transmitted by the user/controller to the input audio device driver to be forwarded as an end-of-stream signal via `send_eos`. This branch terminates with an invocation of the first equation.

The third equation gets an object `effect_descriptor` of type `Descriptor` that summarizes the last configurations received from the user/controller, then sends it to the sound processor, as shown below:

```
Descriptor_Handling(void;
                    local object(Descriptor) effect_descriptor) =
  get_summary_descriptor?(effect_descriptor) .
    send_descriptor!(effect_descriptor) . Config_Handling(0)
```

Finally, we have the definition of the input and output interactions occurring in the three behavioral equations:

```
  INPUT_INTERACTIONS  UNI receive_start;
                          receive_config;
                          receive_descriptor_request;
                          receive_stop

  OUTPUT_INTERACTIONS UNI forward_start;
                          send_descriptor;
                          forward_stop
```

In this case all the interactions are declared to be uni and synchronous (the qualifier `SYNC` can be omitted as it is the default one). All the other actions occurring in the behavioral equations, i.e. `store_config` and `get_summary_descriptor`, are internal actions.

Then we have the definition of the AET for the input audio device driver, which is responsible for opening/closing the input audio device according to the user/controller commands and forwarding the segments of dry audio samples to the sound processor. Its behavior is described by means of two equations, `Idle` and `Busy`, as shown below:

```
ARCHI_ELEM_TYPE Input_Audio_Device_Driver(const integer segment_size)

  BEHAVIOR

    Idle(void;
         void) =
      receive_start . open_input_device!(segment_size) . Busy();

    Busy(void;
         local object(Segment) segment) =
      choice
      {
        read_dry_samples?(segment) . send_dry_segment!(segment) . Busy(),
        receive_stop . close_input_device . send_eos . Idle()
      }
```

```
INPUT_INTERACTIONS  UNI receive_start;
                        read_dry_samples;
                        receive_stop

OUTPUT_INTERACTIONS UNI open_input_device;
                        send_dry_segment;
                        close_input_device;
                        send_eos
```

The AET for the output audio device driver is complementary to the previous one, with some behavioral differences in equation `Idle`. The first difference is that a choice is present in order to receive a processed segment as a start signal, or to receive an end-of-stream signal in the case in which no segment is processed, meaning that a stop signal has been immediately forwarded after the start signal by the console through the input audio device driver and the sound processor. The second difference is that this driver uses the internal action `sleep` to introduce the specified delay, as shown below:

```
ARCHI_ELEM_TYPE Output_Audio_Device_Driver(const integer segment_size,
                                           const integer delay)

  BEHAVIOR

    Idle(void;
         local object(Segment) segment) =
      choice
      {
        receive_processed_segment?(segment) . sleep!(delay) .
          open_output_device!(segment_size) . write_processed_samples!(segment) . Busy(),
        receive_eos . Idle()
      };

    Busy(void;
         local object(Segment) segment) =
      choice
      {
        receive_processed_segment?(segment) . write_processed_samples!(segment) . Busy(),
        receive_eos . close_output_device . Idle()
      }

  INPUT_INTERACTIONS  UNI receive_processed_segment;
                          receive_eos

  OUTPUT_INTERACTIONS UNI open_output_device;
                          write_processed_samples;
                          close_output_device
```

The AET for the sound processor is at the core of the audio processing system. It is in charge of applying the effect chosen by the user/controller to the segments of dry audio samples in order to produce the audio stream to be played out. More precisely, the sound processor periodically receives from the input audio device driver a segment of dry audio samples to process, and sends requests to the console for receiving a descriptor of the effect that has to be applied to the next segments of the stream. In order to guarantee that only one descriptor request is sent to the console (and hence only one effect is received from the effect generator) between the processing of two contiguous dry segments, a boolean variable called `just_done` is employed as shown below:

```
ARCHI_ELEM_TYPE Sound_Processor(const integer segment_size)

 BEHAVIOR

  Segment_Descriptor_Handling(      object(Effect)      effect    := null,
                                    boolean             just_done := false;
                              local object(Segment)     dry_segment,
                              local object(Segment)     processed_segment,
                              local object(Descriptor)  effect_descriptor) =
    choice
    {
      receive_dry_segment?(dry_segment) .
        process_dry_segment!(dry_segment, segment_size, effect) .
          get_processed_segment?(processed_segment) .
            send_processed_segment!(processed_segment) .
              Segment_Descriptor_Handling(effect, false),
      cond(!just_done) ->
        send_descriptor_request . receive_descriptor?(effect_descriptor) .
          Descriptor_Check(effect, effect_descriptor),
      receive_eos . forward_eos . Segment_Descriptor_Handling(effect, false)
    };

  Descriptor_Check(object(Effect)     old_effect,
                   object(Descriptor) effect_descriptor;
                   void) =
    choice
    {
      cond(effect_descriptor != null) ->
        forward_descriptor!(effect_descriptor) . Segment_Effect_Handling(old_effect),
      cond(effect_descriptor = null) ->
        ignore . Segment_Descriptor_Handling(old_effect, true)
    };

  Segment_Effect_Handling(      object(Effect)  old_effect;
                          local object(Segment) dry_segment,
```

```
                         local object(Segment) processed_segment,
                         local object(Effect)  new_effect) =
    choice
    {
      receive_dry_segment?(dry_segment) .
        process_dry_segment!(dry_segment, segment_size, old_effect) .
          get_processed_segment?(processed_segment) .
            send_processed_segment!(processed_segment) .
              Segment_Effect_Handling(old_effect),
      receive_effect?(new_effect) . Segment_Descriptor_Handling(new_effect, true),
      receive_eos . forward_eos . receive_effect?(new_effect) .
        Segment_Descriptor_Handling(new_effect, false)
    }

INPUT_INTERACTIONS  UNI receive_dry_segment;
                        receive_descriptor;
                        receive_effect;
                        receive_eos

OUTPUT_INTERACTIONS UNI send_processed_segment;
                        send_descriptor_request;
                        forward_descriptor;
                        forward_eos
```

The AET for the effect generator simply receives a descriptor from the sound processor, then creates a new effect and sends it back to the sound processor:

```
ARCHI_ELEM_TYPE Effect_Generator(void)

  BEHAVIOR

    Generation(void;
               local object(Descriptor) effect_descriptor
               local object(Effect)     effect) =
      receive_descriptor?(effect_descriptor) . create_new_effect!(effect_descriptor) .
        get_new_effect?(effect) . send_effect!(effect) . Generation()

  INPUT_INTERACTIONS  UNI receive_descriptor

  OUTPUT_INTERACTIONS UNI send_effect
```

In the second section of the PADL textual description of the audio processing system we specify the architectural topology of the system according to the graphical representation of Fig. 3.2. All the input interactions of the console and all the interactions of the two audio device drivers that deal with the two audio devices are declared to be architectural interactions, i.e. interfaces of the whole software system, as shown below:

```
ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES

    C    : Console(allowed_changes);
    IADD : Input_Audio_Device_Driver(segment_size);
    SP   : Sound_Processor(segment_size);
    EG   : Effect_Generator();
    OADD : Output_Audio_Device_Driver(segment_size, delay)

  ARCHI_INTERACTIONS

    C.receive_start;
    C.receive_config;
    C.receive_stop;
    IADD.open_input_device;
    IADD.read_dry_samples;
    IADD.close_input_device;
    OADD.open_output_device;
    OADD.write_processed_samples;
    OADD.close_output_device

  ARCHI_ATTACHMENTS

    FROM C.forward_start            TO IADD.receive_start;
    FROM C.send_descriptor          TO SP.receive_descriptor;
    FROM C.forward_stop             TO IADD.receive_stop;
    FROM IADD.send_dry_segment      TO SP.receive_dry_segment;
    FROM IADD.send_eos              TO SP.receive_eos;
    FROM SP.send_descriptor_request TO C.receive_descriptor_request;
    FROM SP.forward_descriptor      TO EG.receive_descriptor;
    FROM SP.send_processed_segment  TO OADD.receive_processed_segment;
    FROM SP.forward_eos             TO OADD.receive_eos;
    FROM EG.send_effect             TO SP.receive_effect
```

Chapter 4

Thread Coordination Management

The first phase of our approach to the generation of multithreaded object-oriented code from process algebraic architectural descriptions deals with the thread coordination management. This is accomplished by developing a Java package that automatically takes care of the details of thread synchronization. Both the implementation of the package and the use of its units for coordinating the threads are guided by architecture-level abstractions.

In this chapter we present a reference thread communication model (Sect. 4.1) and then we illustrate the implementation of the Java package `Sync` (Sect. 4.2). Four conceptual layers – `Connector`, `Port`, `RunnableElem`, and `RunnableArchi` – that constitute the structure of the package, are presented (Sect. 4.3, 4.4, 4.5, and 4.6, respectively). The usage of the package is exemplified in the last section of this chapter (Sect. 4.7) through the audio processing system introduced in Sect. 3.5.

4.1 Thread Communication Model

The thread communication model adopted by package `Sync` fully complies with the interaction qualifiers of PADL and encompasses two different dimensions.

The first dimension is the thread communication synchronicity and comprises nine values: synchronous to synchronous, synchronous to semi-synchronous, synchronous to asynchronous, semi-synchronous to synchronous, semi-synchronous to semi-synchronous, semi-synchronous to asynchronous, asynchronous to syn-

chronous, asynchronous to semi-synchronous, asynchronous to asynchronous. In a synchronous-to-synchronous communication, both threads wait for the other to become ready. In the semi-synchronous case, the thread checks whether the other is ready and, if not, raises an exception without blocking. In the asynchronous case, the thread simply sends/receives its signal, or message, and then proceeds independently of the status of the other thread, unless at the asynchronous receiving side no signal/message is available and then an exception is raised without blocking.

The second dimension is the thread communication multiplicity and comprises five values: uni to uni, and to uni, uni to and, or to uni, uni to or. In a uni-to-uni communication, only two threads are involved. In an and-to-uni/uni-to-and communication, a thread simultaneously communicates with several other threads. In an or-to-uni/uni-to-or communication, instead, a thread communicates with only one thread selected out of a set of other threads.

Finally, in order to avoid undesired side effects, the adopted thread communication model forbids sharing global variables as well as passing object references while keeping a copy of the references.

4.2 The Java Package `Sync`

The Java package `Sync`, which adheres to the above mentioned communication model, is structured into four conceptual layers: `Connector`, `Port`, `RunnableElem`, and `RunnableArchi`. Each of them corresponds to a different architectural abstraction and comprises a set of components realized through Java classes and interfaces, some of which are visible by the software developer.

The bottom-level layer, called `Connector`, is a set of invisible Java classes and interfaces that are used within `Sync` to perform synchronizations and data transfers between two threads. There are nine classes realizing – consistently with the adopted thread communication model – the nine cases of communication synchronicity.

The second layer, called `Port`, is a set of Java classes and interfaces that realize the abstraction corresponding to a set of statements through which a thread interacts with possibly many other threads. There are eighteen classes combining

– through instances of `Connector` – the five cases of communication multiplicity with the nine cases of communication synchronicity. The eighteen classes manage synchronous/semi-synchronous/asynchronous uni/and/or-interactions both at the sending and at the receiving side.

The third layer, called `RunnableElem`, is an interface derived from the standard `Runnable` interface that realizes the abstraction corresponding to a thread (or a monitor) in a concurrent Java program.

The top-level layer, called `RunnableArchi`, is a `RunnableElem`-derived interface that realizes the abstraction corresponding to a concurrent Java program.

The Java package `Sync` will be seen by the PADL2Java translator as a repository of architectural abstractions guaranteeing the correct and transparent handling of synchronizations and data exchanges among the Java threads (and monitors) of the software system being designed. More precisely, starting from a PADL description, the PADL2Java translator will be in charge of creating as many instances of `RunnableElem` as there are AEIs in the PADL description. Then it will have to create as many instances of `Port` as there are interactions in the AEIs synthesized as `RunnableElem` instances. Finally it will have to create all the instances of `Connector` that are needed to make the AEIs synthesized as `RunnableElem` instances interact – according to the attachments in the PADL description – through their interactions synthesized as `Port` instances.

4.3 The Layer `Connector`

This layer is inspired by the traditional producer-consumer model, where the producer and the consumer represent two different threads. This layer is realized by equipping every `Connector` class with a buffer shared only by the two related producer and consumer threads. Each such class also has two interface methods for accessing the buffer – `send()` and `receive()` – and two interface methods for observing the status of the threads using the `Connector` in the case of and/or-interactions – `obsSnd()` and `obsRcv()`. All of these methods are employed by the upper layer `Port`.

The buffer can store a number of objects that the producer wishes to transfer to the consumer. In the case of a data exchange the related object represents

some data structure, whereas in the case of a pure synchronization a null object is employed. In both cases, mutual exclusion must be enforced on buffer access, i.e. at any time only one of the two involved threads can access the shared buffer. This is achieved by declaring methods `send()` and `receive()` as synchronized methods, so that they result in a monitor-like control structure to which the primitive methods `wait()` and `notify()` can be applied.

The layer `Connector` comprises nine classes realizing the nine cases of communication synchronicity through the two synchronized interface methods `send()` and `receive()` and a number of suitable flags:

1. **Synchronous to Synchronous (S-S)**. In this case, the buffer capacity is one and methods `send()` and `receive()` are implemented according to the usual producer-consumer pattern. More precisely, if `send()` is executed first, it deposits an object into the buffer, then invokes `wait()`; when `receive()` will be executed, it will withdraw the object from the buffer, then will invoke `notify()`. By contrast, if `receive()` is executed first, it invokes `wait()`; when `send()` will be executed, it will deposit an object into the buffer and invokes `notify()`, so that `receive()` can withdraw the object from the buffer.

2. **Synchronous to Semi-Synchronous (S-SS)**. In this case, the buffer capacity is still one. Method `send()` deposits an object into the buffer, then invokes `wait()`. Instead, method `receive()` checks whether the buffer is full. If so, `receive()` withdraws the object from the buffer and then invokes `notify()`. If not, `receive()` raises an exception, which propagates to the layer `Port`.

3. **Synchronous to Asynchronous (S-A)**. In this case, the buffer is unbounded. Method `send()` simply deposits an object into the buffer, without invoking any further method. By contrast, method `receive()` withdraws the first object from the buffer. If the buffer is empty, `receive()` raises an exception, which propagates to the layer `Port`.

4. **Semi-Synchronous to Synchronous (SS-S)**. In this case, the buffer capacity is one. Method `receive()` simply invokes `wait()`. Instead,

method `send()` checks whether the receiving thread is waiting. If so, `send()` deposits an object into the buffer and invokes `notify()`, so that `receive()` can withdraw the object from the buffer. If not, `send()` raises an exception, which propagates to the layer `Port`.

5. **Semi-Synchronous to Semi-Synchronous (SS-SS)**. In this case, the buffer capacity is still one. Method `send()` checks whether the receiving thread is waiting. If so, `send()` deposits an object into the buffer and invokes `notify()`, so that `receive()` can withdraw the object from the buffer. If not, `send()` invokes `wait()` by specifying a timeout; upon the expiration of the timeout an exception is raised, which propagates to the layer `Port`. Instead, method `receive()` checks whether the sending thread is waiting. If so, `receive()` invokes `notify()` followed by `wait()`, so that `send()` can deposit an object and finally invokes `notify()`, in order to cause `receive()` to withdraw the object from the buffer. If not, `receive()` invokes `wait()` by specifying a timeout; upon the expiration of the timeout an exception is raised, which propagates to the layer `Port`.

6. **Semi-Synchronous to Asynchronous (SS-A)**. In this case, the buffer is unbounded. Method `send()` simply deposits an object into the buffer, without invoking any further method. By contrast, method `receive()` withdraws the first object from the buffer. If the buffer is empty, `receive()` raises an exception, which propagates to the layer `Port`. Since the buffer is unbounded, this case behaves as the S-A communication.

7. **Asynchronous to Synchronous (A-S)**. In this case, the buffer is unbounded. Method `send()` deposits an object into the buffer, then invokes `notify()` only if the receiving thread is waiting. By contrast, method `receive()` withdraws the first object from the buffer. If the buffer is empty, `receive()` has to invoke `wait()` before proceeding with the withdrawal.

8. **Asynchronous to Semi-Synchronous (A-SS)**. In this case, the buffer is still unbounded. Method `send()` simply deposits an object into the buffer, without invoking any further method. By contrast, method `receive()` withdraws the first object from the buffer. If the buffer is empty, `receive()`

raises an exception, which propagates to the layer `Port`. Since the buffer is unbounded, this case behaves as the S-A and SS-A communications.

9. **Asynchronous to Asynchronous (A-A)**. In this case, the buffer is unbounded. Method `send()` simply deposits an object into the buffer, without invoking any further method. By contrast, method `receive()` withdraws the first object from the buffer. If the buffer is empty, `receive()` raises an exception, which propagates to the layer `Port`. Since the buffer is unbounded – its capacity has to be considered as the sum of the capacities of the buffers at the sender and at the receiver side – this case behaves as the S-A, SS-A, and A-SS communications.

We conclude by commenting on the other two interface methods of layer `Connector`. Method `obsSnd()` is used by a receiving thread, whereas method `obsRcv()` is used by a sending thread. These two methods are useful at the layer `Port` level for enabling a thread to check the status of a thread connected to it via a synchronous interaction and wait for the latter to become ready to communicate (blocking mode), or connected to it via a semi-synchronous interaction (non-blocking mode). Such two methods are declared as synchronized methods.

4.4 The Layer `Port`

This layer gives rise to suitable objects, each of which will be attached either to a single `Connector` object – if representing a uni-interaction – or to several `Connector` objects – if representing an and-interaction or an or-interaction. Every `Port` object must be instantiated by specifying its owner thread in the `Port` constructor.

The layer `Port` comprises eighteen classes combining the five cases of communication multiplicity with the nine cases of communication synchronicity. Each of the nine `Port` classes related to the sender side is equipped with a `send()` method, whereas each of the nine `Port` classes related to the receiver side is equipped with a `receive()` method. Each of the two methods makes use of the homonymous method of the associated `Connector` objects. More precisely:

- In the case of a `Port` object associated with a uni-interaction, method `send()` (resp. `receive()`) simply invokes the homonymous method of the only `Connector` object involving the `Port` object itself.

- In the case of a `Port` object associated with a synchronous and-interaction, method `send()` (resp. `receive()`) polls all the connected receiving (resp. sending) `Port` objects through the `obsRcv()` (resp. `obsSnd()`) method – in blocking mode – of the related `Connector` objects. When all the connected `Port` objects are ready to interact, method `send()` (resp. `receive()`) invokes the homonymous methods of all the involved `Connector` objects. If the and-interaction is semi-synchronous, method `obsRcv()` (resp. `obsSnd()`) is invoked in non-blocking mode. If the and-interaction is asynchronous, a simple polling phase is needed only for a receiving `Port` object in order to check if at least one of the buffers of the involved `Connector` objects is empty.

- In the case of a `Port` object associated with a synchronous or-interaction, method `send()` (resp. `receive()`) creates as many threads as there are connected `Port` objects, each of which invokes the `obsRcv()` (resp. `obsSnd()`) method – in blocking mode – of the related `Connector` object. The first thread to detect a connected `Port` object ready to interact signals the identifier of the involved `Connector` object, then method `send()` (resp. `receive()`) destroys all the previously created threads and invokes the homonymous method of the involved `Connector` object. If the or-interaction is not synchronous, the additional threads are not needed.

Each of the two methods `send()` and `receive()` of a `Port` object raises an `UnattachedPortException` whenever the `Port` object is not connected to any other `Port` object. This happens if the interaction associated with the `Port` object is an architectural interaction not attached to any other interaction.

A different exception, `NotReadyPortException`, is raised if the associated interaction is semi-synchronous and the connected `Port` objects are not ready to communicate. The same exception is raised by a receiver `Port` object if the

associated interaction is asynchronous and the buffers of the involved `Connector` objects are empty.

Both exceptions are defined as derived classes of a base class called `SyncException`, which is defined within package `Sync`.

4.5 The Layer `RunnableElem`

In this layer a `Runnable`-derived interface is defined instead of directly using the standard `Runnable` interface. As will become clear in Sect. 5 and 6, this is necessary to provide abstractions that are useful to support the generation of Java threads and monitors from the process-algebraically-specified behavior of the corresponding AETs in a PADL description. In particular, the `RunnableElem` interface is equipped with a companion class called `ElemMeth`, which defines a static method called `choice()` for the translation of alternative compositions.

From the code generation viewpoint (i.e. when using package `Sync` in the translation process), each of the Java classes that will be synthesized to implement `RunnableElem` for a specific AET defined in a PADL description will have the same name as the AET and will be structured as shown in Table 4.1.

```
class ◁architectural element type name▷ implements RunnableElem {
  ◁Declaring Behavioral Equations Interfaces▷
  ◁Instantiating Interactions▷
  ◁Declaring Stubs▷
  ◁Defining Constructor▷
  ◁Defining Behavior▷
  ◁Running Element▷
}
```

Table 4.1. Structure of a `RunnableElem`-implementing class

The first section, *Declaring Behavioral Equations Interfaces*, defines an interface called `BehavioralEquationInterface` and declares an equation object of such an interface for each behavioral equation occurring in the AET definition.

The second section, *Instantiating Interactions*, instantiates various `Port` objects of various types (input/output, synchronous/semi-synchronous/asynchronous, uni/and/or) on the basis of the interactions occurring in the AET definition and their qualifiers.

The third section, *Declaring Stubs*, declares two stub objects to be manually filled in later on. As we shall see in Sect. 5.1, one stub object is for the translation of the internal actions occurring in the AET definition, while the other one is for handling exceptions related to the interactions occurring in the AET definition.

The fourth section, *Defining Constructor*, defines the class constructor together with its parameters, which coincide with the parameters of the AET. The constructor declares the parameters as non-public members in order to store their values and make them available throughout the thread execution. Then the constructor invokes the method defined in the next section.

The fifth section, *Defining Behavior*, creates instances of anonymous classes implementing `BehavioralEquationInterface` and assigns them to the previously mentioned equation objects. Each anonymous class translates a different behavioral equation occurring in the AET. The only method declared by the interface, `behavEqCall()`, is defined here for each equation object. As we shall see in Sect. 5, this method implements the behavior formalized by an equation and terminates by storing information about the next equation to be executed together with the actual parameters it needs. This permits the execution of an instance of the class as a state machine, where each state is realized through a different equation object.

Finally, the sixth section, *Running Element*, declares the thread associated with the class itself and defines the public methods `start()`, which starts the previously declared thread, and `join()`, which allows other threads to wait for the end of the execution of the previously declared thread. The public method `run()` is also defined, which is declared in the base interface `Runnable` and is executed first when a Java thread is started. Besides instantiating the two stub classes, this method executes the various equation objects starting from the first one till the point is reached in which a null equation object – which translates the process algebraic term `stop` – is encountered.

In the case in which the AET is synthesized as a monitor rather than a thread, two different classes will be generated: a core monitor class and a wrapper class. The wrapper class, which implements the interface `RunnableElem`, encapsulates the monitor class, which translates the behavior of the AET. The translation of the behavior looks quite different from the code produced for a thread. In fact, as we shall see in Sect. 6, the process algebraic specification of the AET must previously be transformed into monitor normal form, from which a Java monitor can be easily obtained. Another difference between the generation of monitors and of threads is that in a wrapper class method `run()` is intended as a simple starting method, because it only invokes the initialization method of the core monitor class.

4.6 The Layer `RunnableArchi`

This layer derives the `RunnableArchi` interface from the `RunnableElem` interface in order to support hierarchical software development. Thus the `RunnableArchi` interface is fully compatible with the `RunnableElem` interface.

In order to hide the implementation details and to avoid the difficulties stemming from the direct handling of the `Connector` objects, the `RunnableArchi` interface is equipped with a companion class called `ArchiMeth`, which defines a family of five static methods called `attach()`. Each of these methods receives two parameters, which must be a sending `Port` object and a receiving `Port` object, and connects them only if they refer to two different owner threads/monitors, otherwise an exception, `BadAttachmentException`, is raised. According to the adopted communication model, the connection results in one of the following five communication forms: uni-uni, and-uni, uni-and, or-uni, uni-or.

If the two `Port` object parameters are correct, the method `attach()` accomplishes its task in two steps. First, it creates a `Connector` object – and passes to it the references to the two `Port` objects – that realizes the connection between the two `Port` objects in a way that conforms to the synchronicity and multiplicity qualifiers of the associated interactions. Second, it passes a reference to the `Connector` object to both `Port` objects. In this way, the two `Port` objects

can access their common `Connector` object, so that they can communicate using its methods.

In the case in which the connection is between a `Port` object of a thread and a `Port` object of a monitor, method `attach()` behaves differently. Due to the differences between the two `Port` objects, a `Connector` object cannot be used. Instead, the `Port` object of the thread directly invokes the methods defined in the `Port` object of the monitor. To accomplish this, method `attach()` passes a reference to the `Port` object of the monitor to the `Port` object of the thread.

From the code generation viewpoint (i.e. when using package `Sync` in the translation process), the Java class that will be synthesized to implement `RunnableArchi` for the architectural type defined in a specific PADL description will have the same name as the architectural type and will be structured as shown in Table 4.2.

```
public class ◁architectural type name▷ implements RunnableArchi {
  ◁Declaring Runnable Elements▷
  ◁Declaring Architectural Interactions▷
  ◁Defining Constructor▷
  ◁Building Architecture▷ :
    ◁Instantiating Runnable Elements▷
    ◁Assigning Architectural Interactions▷
    ◁Attaching Local Interactions▷
  ◁Running Architecture▷
}
```

Table 4.2. Structure of the `RunnableArchi`-implementing class

The first section, *Declaring Runnable Elements*, declares an object of a `RunnableElem`-implementing class – without instantiating it – for each AEI declared in the PADL description.

The second section, *Declaring Architectural Interactions*, declares a public `Port` object for each architectural interaction declared in the PADL description. Such objects are public because the architectural interactions are the interfaces

of the whole system, hence support must be provided for them to be used for the hierarchical or compositional modeling of complex systems.

The third section, *Defining Constructor*, defines the class constructor together with its parameters, which coincide with the parameters of the architectural type. More precisely, a general class constructor and a default class constructor are defined. The former declares the parameters as non-public members in order to store their values and make them available for the instantiation of the `RunnableElem` objects, then invokes the method defined in the next section. The latter simply invokes the former by passing the actual values of the parameters.

The fourth section, *Building Architecture*, is very similar to the architectural topology section of the PADL description. It is composed of three subsections that constitute the definition of a method called `buildArchiTopology()`, which builds up the architecture topology. In the first subsection, *Instantiating Runnable Elements*, the previously declared `RunnableElem` objects – which can be threads or monitors – are instantiated. In the second one, *Assigning Architectural Interactions*, the previously declared public `Port` objects are assigned through the corresponding `Port` objects of the newly instantiated `RunnableElem` objects. In the third one, *Attaching Local Interactions*, the static method `attach()` defined in the class `ArchiMeth` of package `Sync` is invoked to connect the `Port` objects of the newly instantiated `RunnableElem` objects according to the attachments declared in the PADL description.

Finally, the fifth section, *Running Architecture*, declares the thread associated with the class itself and defines the public methods `start()`, which starts the previously declared thread, and `join()`, which allows other threads to wait for the end of the execution of the previously declared thread. The public method `run()` is also defined, which is declared in the base interface `Runnable` and is executed first when a Java thread is started. This method starts all the previously instantiated `RunnableElem` objects, then waits for the termination of those of them that are realized as threads.

4.7 Audio Processing System: Phase 1

We now exemplify the use of package Sync by means of the top-layer file generated by our approach for the audio processing system introduced in Sect. 3.5. The class defined in the generated file is the RunnableArchi-implementing one, which is called Audio_Processing_System. According to Table 4.2, this class contains several sections related to the AEIs, their architectural interactions, and the attachments among them.

In the first section of the class, a RunnableElem object is declared for each of the five AEIs declared in the PADL description of the audio processing system. Each such object belongs to a RunnableElem-implementing class – corresponding to an AET in the PADL description – which will be widely discussed in Sect. 5 and 6.

```
public class Audio_Processing_System implements RunnableArchi {

  //-------- DECLARING RUNNABLE ELEMENTS --------//
  Console C;
  Input_Audio_Device_Driver IADD;
  Sound_Processor SP;
  Effect_Generator EG;
  Output_Audio_Device_Driver OADD;
```

In the second section, a public Port object is declared for each of the nine architectural interactions declared in the PADL description of the audio processing system.

```
  //--- DECLARING ARCHITECTURAL INTERACTIONS ----//
  public UniSyncReceiverPort C_receive_start;
  public UniSyncReceiverPort C_receive_config;
  public UniSyncReceiverPort C_receive_stop;
  public UniSyncSenderPort IADD_open_input_device;
  public UniSyncReceiverPort IADD_read_dry_samples;
  public UniSyncSenderPort IADD_close_input_device;
  public UniSyncSenderPort OADD_open_output_device;
  public UniSyncSenderPort OADD_write_processed_samples;
  public UniSyncSenderPort OADD_close_output_device;
```

In the third section, a general class constructor and a default class constructor are defined together with some parameters, which coincide with the parameters of the architectural type occurring in the PADL description of the audio processing

system.

```
//---------- DEFINING CONSTRUCTOR -----------//
protected int segment_size;
protected int delay;
protected int allowed_changes;

// GENERAL CONSTRUCTOR:
Audio_Processing_System(int segment_size,
                        int delay,
                        int allowed_changes) {
  this.segment_size = segment_size;
  this.delay = delay;
  this.allowed_changes = allowed_changes;
  buildArchiTopology();
}

// DEFAULT CONSTRUCTOR:
Audio_Processing_System() {
  this(1024,
       125,
       3);
}
```

In the fourth section, method `buildArchiTopology()` instantiates the previously declared five `RunnableElem` objects, assigns the previously declared nine public `Port` objects through the corresponding `Port` objects of the newly instantiated `RunnableElem` objects, and invokes method `attach()` ten times to connect the `Port` objects of the newly instantiated `RunnableElem` objects according to the ten attachments declared in the PADL description of the audio processing system.

```
//---------- BUILDING ARCHITECTURE ----------//
void buildArchiTopology() {

  // INSTANTIATING RUNNABLE ELEMENTS:
  C = new Console(allowed_changes);
  IADD = new Input_Audio_Device_Driver(segment_size);
  SP = new Sound_Processor(segment_size);
  EG = new Effect_Generator();
  OADD = new Output_Audio_Device_Driver(segment_size, delay);

  // ASSIGNING ARCHITECTURAL INTERACTIONS:
  this.C_receive_start = C.receive_start;
  this.C_receive_config = C.receive_config;
  this.C_receive_stop = C.receive_stop;
  this.IADD_open_input_device = IADD.open_input_device;
  this.IADD_read_dry_samples = IADD.read_dry_samples;
```

```
    this.IADD_close_input_device = IADD.close_input_device;
    this.OADD_open_output_device = OADD.open_output_device;
    this.OADD_write_processed_samples = OADD.write_processed_samples;
    this.OADD_close_output_device = OADD.close_output_device;

    // ATTACHING LOCAL INTERACTIONS:
    try {
      ArchiMeth.attach(C.forward_start, IADD.receive_start);
      ArchiMeth.attach(C.send_descriptor, SP.receive_descriptor);
      ArchiMeth.attach(C.forward_stop, IADD.receive_stop);
      ArchiMeth.attach(IADD.send_dry_segment, SP.receive_dry_segment);
      ArchiMeth.attach(IADD.send_eos, SP.receive_eos);
      ArchiMeth.attach(SP.send_descriptor_request, C.receive_descriptor_request);
      ArchiMeth.attach(SP.forward_descriptor, EG.receive_descriptor);
      ArchiMeth.attach(SP.send_processed_segment, OADD.receive_processed_segment);
      ArchiMeth.attach(SP.forward_eos, OADD.receive_eos);
      ArchiMeth.attach(EG.send_effect, SP.receive_effect);
    } catch(BadAttachmentException e) {}
  }
```

In the fifth section, the thread associated with the class is declared and methods `start()`, `join()`, and `run()` are defined, with the last one starting the execution of the previously instantiated five `RunnableElem` objects.

```
  //---------- RUNNING ARCHITECTURE -----------//
  Thread th_Audio_Processing_System = null;

  public void start() {
    (th_Audio_Processing_System = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Audio_Processing_System.join();
  }

  public void run() {
    C.start();
    IADD.start();
    SP.start();
    EG.start();
    OADD.start();
    try {
      C.join();
      IADD.join();
      SP.join();
      EG.join();
      OADD.join();
    } catch(InterruptedException e) {}
  }
}
```

Chapter 5

Thread Behavior Generation

The second phase of our approach to the generation of multithreaded object-oriented code from process algebraic architectural descriptions deals with the translation of the process algebraic specification of the behavior of the AETs into thread classes. Due to the different level of abstraction of an architectural description language and of a programming language, only a partial translation based on stubs is possible, with the preservation of architectural properties depending on the way in which the stubs are filled in by the software developer.

In this chapter we introduce a reference thread generation model (Sect. 5.1) and we show how to synthesize thread method `run()` (Sect. 5.2) and how to translate process algebraic operators (Sect. 5.3-5.6).

Then we present conditions guaranteeing the preservation of architectural properties and we discuss some guidelines to be followed when filling in the stubs (Sect. 5.7). The translation process will be exemplified in the last section of this chapter (Sect. 5.8) through the audio processing system of Sect. 3.5.

5.1 Thread Generation Model

A finite state machine model is adopted to guide the generation of a thread from the description of an AET made out of a sequence of process algebraic defining equations. Large conditional statements or table-based approaches do not guarantee high efficiency when many conditions or associations have to be

checked at run time for each behavioral invocation. Therefore we generate code on the basis of the behavioral pattern “State” [33].

More precisely, each AET is translated into a context class implementing the interface `RunnableElem`, in which each state is defined as an inner class implementing the interface `BehavioralEquationInterface` and defining the method `behavEqCall()`. The idea is that every behavioral equation is translated into an inner state class having the same name as the equation. The code for an inner state class is generated by proceeding by induction on the syntactical structure – stop, behavior invocation, action prefix, and choice – of the process algebraic term occurring on the right-hand side of the corresponding behavioral equation.

The context class also defines the member `nextBehavEq`, a reference to an object implementing the above mentioned interface, and the member `actualPars`, a reference to an array of objects. These references, which are shared and visible by all the inner state classes, are in charge of defining the state transitions, i.e. the next behavioral equation to be executed and the actual parameters to be passed.

We conclude by observing that a different treatment is needed for action prefixes depending on whether the related actions are interactions or internal actions. An interaction is involved in communications, hence it is automatically managed via package `Sync`. However, as we have seen in Sect. 4.4, the execution of an interaction may result in two kinds of exception, whose handling is left to the software developer. Our thread generation model thus includes the possibility of associating an exception handling stub (EHS) with each interaction, to be filled in by the software developer. By contrast, an internal action is not involved at all in communications, as it takes place inside an AET. In this case the architectural description does not provide any information about how to translate the action into a sequence of Java statements. As a consequence, our thread generation model also includes the possibility of associating an internal action stub (IAS) with each internal action, to be filled in by the software developer.

5.2 Synthesizing Thread Method `run()`

The context class corresponding to an AET comprises the constructor and method `run()`. While the former instantiates the inner state classes, as shown in Table 5.1 the latter first of all instantiates the EHSs and the IASs declared as members of the context class.

```
public void run() {
  ◁EHS instantiation▷
  ◁IAS instantiation▷
  nextBehavEq = ◁first behavioral equation▷;
  actualPars = ◁actual parameters of the first behavioral equation▷;
  while (nextBehavEq != null)
    nextBehavEq.behavEqCall();
}
```

Table 5.1. Structure of thread method `run()`

Method `run()` then assigns the state class instance representing the first behavioral equation to `nextBehavEq` and the associated parameters to `actualPars`. A `while` statement carries out the execution of the behavioral equations starting from the first one, by repeatedly invoking method `behavEqCall()` on the state class instance referenced by `nextBehavEq`. When `nextBehavEq` is set to `null`, the execution of method `run()` terminates.

5.3 Translating `stop`

Process term `stop` represents the situation in which no further action can be executed. It is therefore translated by assigning `null` both to `nextBehavEq` and to `actualPars`. As a consequence, when encountering `stop` method `run()` terminates its execution

5.4 Translating Behavioral Invocations

The behavioral invocation $B(\underline{e})$ represents a process term that behaves as the behavioral equation whose identifier is B, when passing the possibly empty sequence of actual parameters $\underline{e}$. A behavioral invocation, which can occur only within the scope of an action prefix operator, is not translated into a method call, as this may result in the generation of inefficient code in case of recursion. Instead, a behavioral invocation is translated into an assignment to `nextBehavEq` of an instance of the inner state class that corresponds to the next behavioral equation, followed by an assignment to `actualPars` of the actual parameters needed by the next behavioral equation.

5.5 Translating Action Prefixes

The action prefix operator is used to represent a process term that can execute an action and then behaves as described by another process term. The translation of the action depends on whether it is an interaction or an internal action.

In the first case, the action is translated into an invocation of method `send()` – if it is an output interaction – or method `receive()` – if it is an input interaction – of the corresponding instance of a class of layer `Port`. The translation must then be completed by filling in the corresponding EHSs.

In the second case, instead, the action translation is completely left to the software developer, as internal actions cannot be treated automatically at all. A method for each of them is placed in a distinct IAS, which has to be filled in by the software developer with the corresponding Java statements. As a consequence, every occurrence of an internal action is translated into an invocation of the related method in an IAS.

5.6 Translating Choices

The choice operator expresses a selection among a certain number of alternative behaviors described through process terms. A choice-based process term is translated into a `switch-case` statement, whose condition is given by an

invocation of the static method `choice()` defined in the class `ElemMeth` of package Sync.

There are two cases that must be addressed in order to translate the choice operator. The first one is the case where every process term involved in the choice starts with an action prefix operator. In this case the method `choice()` is directly employed, which accepts as input an array of objects of class `ChAct`, each of which contains a boolean guard expressing the possible constraint under which the corresponding starting action is enabled (default value `true`). Should one of the starting actions be an interaction, an additional piece of information is contained in the corresponding object, which is a reference to the `Port` object associated with the interaction. Method `choice()` returns the index (within the array) of the starting action selected for execution.

A starting action is enabled (and hence can be selected for execution) if its guard evaluates to `true` and – in the case of a synchronous interaction – the corresponding `Port` object is ready to communicate. If several starting actions are enabled, a probabilistic mechanism is applied to select one of those actions. If all the starting actions with guard evaluating to `true` are synchronous interactions, method `choice()` waits – and the thread that contains it passivates – until one of the associated `Port` objects is ready to communicate. If all the guards of the starting actions evaluate to `false`, method `choice()` returns a negative value.

Based on the index returned by `choice()`, the `switch-case` statement invokes the method associated with the execution of the selected starting action. This method is `send()` or `receive()` in the case of an interaction, whereas for an internal action it is the corresponding method in the related IAS. The invocation of this method is followed in turn by the translation of the process term prefixed by the selected action. In the default clause, which comes into play when a negative value is returned by `choice()`, process term `stop` is invoked by assigning `null` both to `nextBehavEq` and to `actualPars`.

The second case is the one in which some of the process terms involved in the choice do not start with an action prefix operator. If one of these process terms is `stop`, then nothing has to be added for it in the `ChAct` array and the `switch-case` statement, because it is selected by default whenever the other involved process terms cannot be selected. If instead one of these process terms is a nested choice,

then a flattening of the nested choice takes place during the translation. This means that the array of objects of class `ChAct` and the `switch-case` statement for the outer choice are extended in order to include all the alternative starting actions that are contained in the inner choice. The event in which one of the process terms involved in the choice is a behavioral invocation cannot happen, because a behavioral invocation can only occur within an action prefix operator.

5.7 Preservation of Architectural Properties

PADL is equipped with a component-oriented technique based on equivalence checking for verifying the freedom from architectural mismatches [1, 12]. These are the malfunctionings that arise when assembling together several components that are correct if considered in isolation. More precisely, the class of properties dealt with by the technique – which includes for instance deadlock freedom – is characterized by three constraints. First, the properties must be concerned with the interactions, as internal actions cannot affect communications among components. Second, for each property $\mathcal{P}$ in the class, there must exist a behavioral equivalence $\approx_{\mathcal{P}}$ coarser than $\approx_{\rm B}$ (weak bisimilarity [56]) that (*i*) is able to abstract from internal actions, (*ii*) preserves $\mathcal{P}$ in the sense that it never equates two process terms such that one of them satisfies $\mathcal{P}$ while the other does not, and (*iii*) is a congruence with respect to static process algebraic operators. Third, the (action-based) temporal logic in which the properties of the class are expressed must not allow the negation to be freely used.

An important issue is to guarantee that the properties proved at the architectural level are then preserved at the code level. Since we have taken an approach based on automatic code generation, property preservation should be achieved by construction. In other words, the translation from PADL to Java illustrated before should have been defined in a way that ensures property preservation. We now investigate this issue by separately considering the code generated for thread management, the code generated for translating behavioral equations, and the code provided for filling in the stubs.

The code generated for managing the threads cannot infringe the preservation of architectural properties, up to the methods for handling the exceptions that

architectural interactions and semi-synchronous interactions may raise. In fact, the code for thread coordination is completely generated in an automatic way by means of package `Sync`. As far as the system topology is concerned, this is built in the `RunnableArchi`-implementing class in the same way as prescribed by the second section of the PADL description. Moreover, both PADL and `Sync` adhere to the same communication model. On the PADL side, each interaction is given three qualifiers: output vs. input, synchronous vs. semi-synchronous vs. asynchronous (only in the output case), uni vs. and vs. or. Each interaction is then translated into an invocation of method `send()` or `receive()` defined in the corresponding `Port` object, depending on whether it is an output or an input interaction, respectively. Additionally, the kind of this `Port` object – synchronous vs. semi-synchronous vs. asynchronous (only in the output case), uni vs. and vs. or – is the same as that of the interaction.

Each behavioral equation occurring in the description of an AET is translated into an inner state class of the corresponding `RunnableElem`-implementing class. The translation proceeds by induction on the syntactical structure of the process term occurring on the right-hand side of the behavioral equation. The way in which the translation is carried out, together with the way in which the thread execution flow proceeds according to the order established by the invocations of the behavioral equations, ensures the preservation of the process algebraic semantics, up to the methods for translating the internal actions.

As a consequence, the preservation of architectural properties critically depends on the way in which the software developer manually fills in EHSs and IASs. Here we shall consider only IASs, as EHSs can be treated similarly.

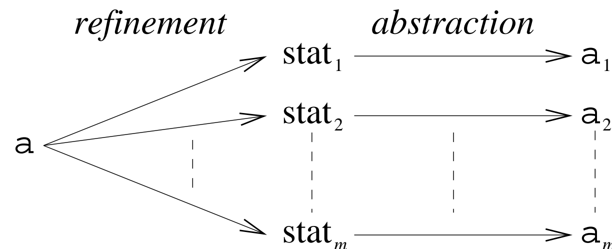

Figure 5.1. Internal action refinement and related statement abstraction

In order to be able to reason about architectural property preservation, we have to compare the internal actions and the corresponding sequences of Java statements on the same process algebraic ground. As shown in Fig. 5.1, the Java statements into which an internal action is refined during the translation process can be abstractly viewed as fresh actions. The following theorem provides a sufficient condition for ensuring the preservation of an architectural property of the considered class. Below, we denote by $_/_$ the hiding operator and by $\simeq_{\rm B}$ the observational congruence of [56].

Theorem 5.1 *Let T be the process algebraic description of the behavior of a thread and let a be an internal action occurring in T. Let $a_1, a_2, \ldots, a_m$ be the fresh actions abstracting the statements into which a is translated and let T' be the process algebraic description of the behavior of the thread obtained from T by replacing every occurrence of $a._$ with $a_1.a_2.\ldots.a_m._$. Let H be the set of internal actions occurring in T or T'. Whenever T satisfies $\mathcal{P}$ and $a.stop/H \simeq_{\rm B} a_1.a_2.\ldots.a_m.stop/H$, then T' satisfies $\mathcal{P}$ as well.*

Proof: Since $\simeq_{\rm B}$ is a congruence with respect to all the process algebraic operators, from $a.stop/H \simeq_{\rm B} a_1.a_2.\ldots.a_m.stop/H$ it follows that $T/H \simeq_{\rm B} T'/H$, hence $T/H \approx_{\rm B} T'/H$. Since $\mathcal{P}$ must be equipped with a weak equivalence $\approx_{\mathcal{P}}$ coarser than $\approx_{\rm B}$, it follows that $T/H \approx_{\mathcal{P}} T'/H$. Since T satisfies $\mathcal{P}$, $\approx_{\mathcal{P}}$ preserves $\mathcal{P}$, and $\mathcal{P}$ can only make assertions about the interactions (which do not belong to H), it follows that T' satisfies $\mathcal{P}$ as well. ■

Note that in the theorem above it is not necessarily the case that all of the actions $a_1, a_2, \ldots, a_m$ associated with the Java statements provided by the software developer belong to H. As an example, one of such actions may correspond to an invocation of `send()` or `receive()` or of a method such as `behavEqCall()` belonging to a state class. Fortunately, in practice both cases are prevented from occurring by the fact that the `Port` objects – which contain methods `send()` and `receive()` – and the `RunnableElem`-implementing class instances – which contain the state classes – are not visible within the stubs.

We conclude by providing some guidelines that the developer should follow when filling in the stubs in order to preserve architectural properties:

- No synchronized methods – like `wait()` and `notify()` – should be defined within the stubs, as they would be internal to the threads but at the same time could affect the way threads communicate with each other.

- No further thread should be created within the stubs, as this would have an observable impact on the system topology and on thread coordination.

- There should be no variables/objects that are visible from several stub classes. This means that all the data shared by several threads should be exchanged only through suitable units of package `Sync`.

- The stub method associated with the first internal action following an invocation of `send()` or `receive()` should copy every object passed in that invocation, and all the stub methods associated with the subsequent internal actions should work on those copies of the objects. This would avoid interferences among threads stemming from the fact that `send()` always keeps a reference to the passed objects – so that it can be defined within `Sync` in a way that supports arbitrarily many parameters of arbitrary types – and such objects may be modified by the stub method associated with some internal action.

- All the exceptions that can be raised when executing a stub method should be caught, or prevented from being raised, inside the stub method itself.

- Non-terminating statements should not occur within stub methods.

5.8 Audio Processing System: Phase 2

We now illustrate the generation of the thread behavior by means of the synthesis of the `RunnableElem`-implementing classes and of their related stub classes for all of the five AETs of the audio processing system introduced in Sect. 3.5. A special emphasis will be placed on explaining the AET `Console`. The reason of this choice is twofold. First, both IAS classes and EHS classes must be generated, as AET `Console` contains both internal actions and architectural interactions. Second, the same AET will be used in Sect. 6 for exemplifying the third phase of

our approach, as the console is a good candidate for the generation of a monitor class. This will permit a comparison between the code generated for a thread and the code generated for a monitor.

According to Table 4.1, the code generated for the `RunnableElem`-implementing class `Console` is composed of several sections. In the first section, the interface `BehavioralEquationInterface` – which declares method `behavEqCall()` – is defined and `Start`, `Config_Handling`, `Descriptor_Handling` – in correspondence with the three behavioral equations occurring in the definition of AET `Console` – and `nextBehavEq` are declared as references to that interface. The array `actualPars` is also declared, which will contain the actual parameters of the next behavioral equation.

```
class Console implements RunnableElem {

  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }
  BehavioralEquationInterface Start, Config_Handling, Descriptor_Handling;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;
```

In the second section, a `Port` object is declared and instantiated for each of the four input interactions and the three output interactions declared in the definition of AET `Console`.

```
  //-------- INSTANTIATING INTERACTIONS ---------//
  UniSyncReceiverPort receive_start =
    new UniSyncReceiverPort(this);
  UniSyncReceiverPort receive_config =
    new UniSyncReceiverPort(this);
  UniSyncReceiverPort receive_descriptor_request =
    new UniSyncReceiverPort(this);
  UniSyncReceiverPort receive_stop =
    new UniSyncReceiverPort(this);

  UniSyncSenderPort forward_start =
    new UniSyncSenderPort(this);
  UniSyncSenderPort send_descriptor =
    new UniSyncSenderPort(this);
  UniSyncSenderPort forward_stop =
    new UniSyncSenderPort(this);
```

In the third section, `internal_Console` and `exception_Console` are declared as references to the stub classes `IAS_Console` and `EHS_Console`, respectively,

which will be instantiated later on.

```
//------------- DECLARING STUBS -------------//
IAS_Console internal_Console;
EHS_Console exception_Console;
```

In the fourth section, the class constructor is declared together with one parameter, which coincide with the parameter of AET `Console`.

```
//----------- DEFINING CONSTRUCTOR ------------//
protected int allowed_changes;

Console(int allowed_changes) {
  this.allowed_changes = allowed_changes;
  defineBehavEquations();
}
```

In the fifth section, method `defineBehavEquations()` is defined, which assigns the instances of new anonymous `BehavioralEquationInterface`-implementing classes to the references `Start`, `Config_Handling`, and `Descriptor_Handling`. The interactions `receive_start` and `forward_start` of the first behavioral equation are translated into invocations of methods `receive()` and `send()`, respectively, which are defined in the corresponding `Port` objects. Since the first interaction is declared as an architectural interaction in the PADL description of the audio processing system, the corresponding invocation of `receive()` may raise an `UnattachedPortException`. This is handled by an invocation of the method `receive_start()` defined in the EHS class instance `exception_Console`. In the translation of the second and third equation, the array `inputPars` of type `Object` is declared in order to store the values of the input action parameters, which are declared as local variables in the behavioral equations themselves. The internal actions `store_config` and `get_summary_descriptor` are translated into invocations of the homonymous methods defined in the IAS class instance `internal_Console`, with the appropriate input or output parameters – preceded by "?" and "!" in the PADL description of the audio processing system, respectively.

```
//------------ DEFINING BEHAVIOR -------------//
void defineBehavEquations() {
```

```
Start =
  new BehavioralEquationInterface() {

    public void behavEqCall() {
      _Start((int)actualPars[0]);
    }

    private void _Start(int config_changes) {
      try {
        receive_start.receive();
      } catch(UnattachedPortException e) {
        exception_Console.receive_start();
      }
      try {
        forward_start.send();
      } catch(SyncException e) {}
      nextBehavEq = Config_Handling;
      actualPars = new Object[] {config_changes};
    }

  }; // end of behavioral equation Start

Config_Handling =
  new BehavioralEquationInterface() {

    public void behavEqCall() {
      _Config_Handling((int)actualPars[0]);
    }

    private void _Config_Handling(int config_changes) {
      Interface_Configuration console_config;
      Object[] inputPars;

      switch (
        ElemMeth.choice(
          new ChAct[] {
            new ChAct(config_changes < allowed_changes, receive_config),
            new ChAct(config_changes > 0, receive_descriptor_request),
            new ChAct(config_changes == 0, receive_descriptor_request),
            new ChAct(true, receive_stop)
          }
        )
      )
      {
        case 0:
          try {
            inputPars = receive_config.receive();
            console_config = (Interface_Configuration)inputPars[0];
          } catch(UnattachedPortException e) {
            inputPars = exception_Console.receive_config();
            console_config = (Interface_Configuration)inputPars[0];
```

```
          }
          internal_Console.store_config(console_config);
          nextBehavEq = Config_Handling;
          actualPars = new Object[] {config_changes + 1};
          break;
        case 1:
          try {
            receive_descriptor_request.receive();
          } catch(SyncException e) {}
          nextBehavEq = Descriptor_Handling;
          actualPars = null;
          break;
        case 2:
          try {
            receive_descriptor_request.receive();
          } catch(SyncException e) {}
          try {
            send_descriptor.send(null);
          } catch(SyncException e) {}
          nextBehavEq = Config_Handling;
          actualPars = new Object[] {0};
          break;
        case 3:
          try {
            receive_stop.receive();
          } catch(UnattachedPortException e) {
            exception_Console.receive_stop();
          }
          try {
            forward_stop.send();
          } catch(SyncException e) {}
          nextBehavEq = Start;
          actualPars = new Object[] {config_changes};
          break;
        default:
          nextBehavEq = null;
          actualPars = null;
      }
    }

  }; // end of behavioral equation Config_Handling

Descriptor_Handling =
  new BehavioralEquationInterface() {

    public void behavEqCall() {
      _Descriptor_Handling();
    }

    private void _Descriptor_Handling() {
      Interface_Descriptor effect_descriptor;
```

```
        Object[] inputPars;

        inputPars = internal_Console.get_summary_descriptor();
        effect_descriptor = (Interface_Descriptor)inputPars[0]
        try {
          send_descriptor.send(effect_descriptor.clone());
        } catch(SyncException e) {}
        nextBehavEq = Config_Handling;
        actualPars = new Object[] {0};
      }

    }; // end of behavioral equation Descriptor_Handling

  }
```

In the sixth section, the thread associated with the class is declared and methods `start()`, `join()`, and `run()` are defined, with the last one executing the various behavioral equations through a `while` statement.

```
  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Console = null;

  public void start() {
    (th_Console = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Console.join();
  }

  public void run() {
    exception_Console = new EHS_Console();
    internal_Console = new IAS_Console();
    nextBehavEq = Start;
    actualPars = new Object[] {0};
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}
```

The stub class `IAS_Console` contains the definition of the methods associated with the two internal actions occurring in the definition of AET `Console`. These methods will have to be filled in by the developer based on the semantics of the internal actions themselves. The developer is also allowed to fill in the body of the constructor of the stub class and to add member declarations whenever needed.

```
class IAS_Console {

  IAS_Console() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void store_config(Interface_Configuration console_config) {
    // FILL IN THE METHOD BODY
  }

  Object[] get_summary_descriptor() {
    Interface_Descriptor effect_descriptor = null;
    // FILL IN THE METHOD BODY
    return new Object[] {effect_descriptor};
  }

}
```

Similarly, the stub class `EHS_Console` contains the definition of the methods associated with the three architectural interactions occurring in the AET `Console` definition. These methods handles exceptions of type `UnattachedPortException` that may be raised by such architectural interactions. As before, the developer will have to fill in these methods as well as the body of the constructor if needed, and is allowed to add member declarations.

```
class EHS_Console {

  EHS_Console() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void receive_start() {
    // FILL IN THE METHOD BODY
  }

  Object[] receive_config() {
    Interface_Configuration console_config = null;
    // FILL IN THE METHOD BODY
    return new Object[] {console_config};
  }

  void receive_stop() {
    // FILL IN THE METHOD BODY
  }

}
```

The code generated for the second AET of the audio processing system,

i.e. Input_Audio_Device_Driver, is the following RunnableElem-implementing class:

```
class Input_Audio_Device_Driver implements RunnableElem
{
  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }

  BehavioralEquationInterface Idle,
                              Busy;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
  UniSyncReceiverPort receive_start =
    new UniSyncReceiverPort(this);
  UniSyncReceiverPort read_dry_samples =
    new UniSyncReceiverPort(this);
  UniSyncReceiverPort receive_stop =
    new UniSyncReceiverPort(this);

  UniSyncSenderPort open_input_device =
    new UniSyncSenderPort(this);
  UniSyncSenderPort send_dry_segment =
    new UniSyncSenderPort(this);
  UniSyncSenderPort close_input_device =
    new UniSyncSenderPort(this);
  UniSyncSenderPort send_eos =
    new UniSyncSenderPort(this);

  //-------------- DECLARING STUBS --------------//
  // No IAS declaration as there are
  // no internal actions.
  EHS_Input_Audio_Device_Driver exception_Input_Audio_Device_Driver;

  //----------- DEFINING CONSTRUCTOR ------------//
  protected int segment_size;

  Input_Audio_Device_Driver(int segment_size) {
    this.segment_size = segment_size;
    defineBehavEquations();
  }

  //------------ DEFINING BEHAVIOR --------------//
  void defineBehavEquations() {

    Idle =
      new BehavioralEquationInterface() {

        public void behavEqCall() {
          _Idle();
```

```
    }

    private void _Idle() {
      try {
        receive_start.receive();
      } catch(SyncException e) {}
      try {
        open_input_device.send(segment_size);
      } catch(UnattachedPortException e) {
        exception_Input_Audio_Device_Driver.open_input_device(segment_size);
      }
      nextBehavEq = Busy;
      actualPars = null;
    }

  }; // end of behavioral equation Idle

Busy =
  new BehavioralEquationInterface() {

    public void behavEqCall() {
      _Busy();
    }

    private void _Busy() {
      Interface_Segment segment;
      Object[] inputPars;
      switch (
        ElemMeth.choice(
          new ChAct[] {
            new ChAct(true, read_dry_samples),
            new ChAct(true, receive_stop)
          }
        )
      ) // Choice body :
      {
        case 0:
          try {
            inputPars = read_dry_samples.receive();
            segment = (Interface_Segment)inputPars[0];
          } catch(UnattachedPortException e) {
            inputPars = exception_Input_Audio_Device_Driver.read_dry_samples();
            segment = (Interface_Segment)inputPars[0];
          }
          try {
            send_dry_segment.send(segment.clone());
          } catch(SyncException e) {}
          nextBehavEq = Busy;
          actualPars = null;
          break;
        case 1:
```

```
            try {
              receive_stop.receive();
            } catch(SyncException e) {}
            try {
              close_input_device.send();
            } catch(UnattachedPortException e) {
              exception_Input_Audio_Device_Driver.close_input_device();
            }
            try {
              send_eos.send();
            } catch(SyncException e) {}
            nextBehavEq = Idle;
            actualPars = null;
            break;
          default:
            nextBehavEq = null;
            actualPars = null;
        }
      }

    }; // end of behavioral equation Busy

  }

  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Input_Audio_Device_Driver = null;

  public void start() {
    (th_Input_Audio_Device_Driver = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Input_Audio_Device_Driver.join();
  }

  public void run() {
    exception_Input_Audio_Device_Driver =
      new EHS_Input_Audio_Device_Driver();
    nextBehavEq = Idle;
    actualPars = null;
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}
```

Note that the AET `Input_Audio_Device_Driver` does not contain internal actions. For this reason, no IAS stub class is generated for it, while the class `EHS_Input_Audio_Device_Driver` is synthesized for handling the excep-

tion `UnattachedPortException` that can be raised by `open_input_device`, `read_dry_samples`, and `close_input_device`:

```
class EHS_Input_Audio_Device_Driver {

  EHS_Input_Audio_Device_Driver() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void open_input_device(int segment_size) {
    // FILL IN THE METHOD BODY
  }

  Object[] read_dry_samples() {
    Interface_Segment segment = null;
    // FILL IN THE METHOD BODY
    return new Object[] {segment};
  }

  void close_input_device() {
    // FILL IN THE METHOD BODY
  }

}
```

The third AET – which is very similar to the previous one – occurring in the audio processing system is the `Output_Audio_Device_Driver`. The code generated for it is:

```
class Output_Audio_Device_Driver implements RunnableElem
{
  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }

  BehavioralEquationInterface Idle,
                              Busy;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
  UniSyncReceiverPort receive_processed_segment =
    new UniSyncReceiverPort(this);
  UniSyncReceiverPort receive_eos =
    new UniSyncReceiverPort(this);

  UniSyncSenderPort open_output_device =
    new UniSyncSenderPort(this);
  UniSyncSenderPort write_processed_samples =
    new UniSyncSenderPort(this);
```

```
UniSyncSenderPort close_output_device =
  new UniSyncSenderPort(this);

//-------------- DECLARING STUBS --------------//
IAS_Output_Audio_Device_Driver internal_Output_Audio_Device_Driver;
EHS_Output_Audio_Device_Driver exception_Output_Audio_Device_Driver;

//----------- DEFINING CONSTRUCTOR ------------//
protected int segment_size;
protected int delay;

Output_Audio_Device_Driver(int segment_size,
                           int delay) {
  this.segment_size = segment_size;
  this.delay = delay;
  defineBehavEquations();
}

//------------ DEFINING BEHAVIOR --------------//
void defineBehavEquations() {

  Idle =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Idle();
      }

      private void _Idle() {
        Interface_Segment segment;
        Object[] inputPars;
        switch (
          ElemMeth.choice(
            new ChAct[] {
              new ChAct(true, receive_processed_segment),
              new ChAct(true, receive_eos)
            }
          )
        ) // Choice body :
        {
          case 0:
            try {
              inputPars = receive_processed_segment.receive();
              segment = (Interface_Segment)inputPars[0];
            } catch(SyncException e) {}
            internal_Output_Audio_Device_Driver.sleep(delay);
            try {
              open_output_device.send(segment_size);
            } catch(UnattachedPortException e) {
              exception_Output_Audio_Device_Driver.open_output_device(segment_size);
            }
```

```
     try {
       write_processed_samples.send(segment.clone());
     } catch(UnattachedPortException e) {
       exception_Output_Audio_Device_Driver.write_processed_samples(segment);
     }
     nextBehavEq = Busy;
     actualPars = null;
     break;
    case 1:
     try {
       receive_eos.receive();
     } catch(SyncException e) {}
     nextBehavEq = Idle;
     actualPars = null;
     break;
    default:
     nextBehavEq = null;
     actualPars = null;
   }
  }

 }; // end of behavioral equation Idle

Busy =
 new BehavioralEquationInterface() {

  public void behavEqCall() {
   _Busy();
  }

  private void _Busy() {
   Interface_Segment segment;
   Object[] inputPars;
   switch (
    ElemMeth.choice(
     new ChAct[] {
      new ChAct(true, receive_processed_segment),
      new ChAct(true, receive_eos)
     }
    )
   ) // Choice body :
   {
    case 0:
     try {
       inputPars = receive_processed_segment.receive();
       segment = (Interface_Segment)inputPars[0];
     } catch(SyncException e) {}
     try {
       write_processed_samples.send(segment.clone());
     } catch(UnattachedPortException e) {
       exception_Output_Audio_Device_Driver.write_processed_samples(segment);
```

```
            }
            nextBehavEq = Busy;
            actualPars = null;
            break;
          case 1:
            try {
              receive_eos.receive();
            } catch(SyncException e) {}
            try {
              close_output_device.send();
            } catch(UnattachedPortException e) {
              exception_Output_Audio_Device_Driver.close_output_device();
            }
            nextBehavEq = Idle;
            actualPars = null;
            break;
          default:
            nextBehavEq = null;
            actualPars = null;
        }
      }

    }; // end of behavioral equation Busy

  }

  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Output_Audio_Device_Driver = null;

  public void start() {
    (th_Output_Audio_Device_Driver = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Output_Audio_Device_Driver.join();
  }

  public void run() {
    internal_Output_Audio_Device_Driver =
      new IAS_Output_Audio_Device_Driver();
    exception_Output_Audio_Device_Driver =
      new EHS_Output_Audio_Device_Driver();
    nextBehavEq = Idle;
    actualPars = null;
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}
```

Since the AET `Output_Audio_Device_Driver` contains the internal action

sleep, the stub class IAS_Output_Audio_Device_Driver is generated as follows:

```
class IAS_Output_Audio_Device_Driver {

  IAS_Output_Audio_Device_Driver() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void sleep(int delay) {
    // FILL IN THE METHOD BODY
  }

}
```

and the class EHS_Output_Audio_Device_Driver for handling the exceptions UnattachedPortException is also generated:

```
class EHS_Output_Audio_Device_Driver {

  EHS_Output_Audio_Device_Driver() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void open_output_device(int segment_size) {
    // FILL IN THE METHOD BODY
  }

  void write_processed_samples(Interface_Segment segment) {
    // FILL IN THE METHOD BODY
  }

  void close_output_device() {
    // FILL IN THE METHOD BODY
  }

}
```

The code generated for the fourth AET of the audio processing system, i.e. Sound_Processor, is as follows:

```
class Sound_Processor implements RunnableElem
{
  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }

  BehavioralEquationInterface Segment_Descriptor_Handling,
                              Descriptor_Check,
                              Segment_Effect_Handling;
  BehavioralEquationInterface nextBehavEq;
```

```
Object[] actualPars;

//-------- INSTANTIATING INTERACTIONS ---------//
UniSyncReceiverPort receive_dry_segment =
  new UniSyncReceiverPort(this);
UniSyncReceiverPort receive_descriptor =
  new UniSyncReceiverPort(this);
UniSyncReceiverPort receive_effect =
  new UniSyncReceiverPort(this);
UniSyncReceiverPort receive_eos =
  new UniSyncReceiverPort(this);

UniSyncSenderPort send_processed_segment =
  new UniSyncSenderPort(this);
UniSyncSenderPort send_descriptor_request =
  new UniSyncSenderPort(this);
UniSyncSenderPort forward_descriptor =
  new UniSyncSenderPort(this);
UniSyncSenderPort forward_eos =
  new UniSyncSenderPort(this);

//-------------- DECLARING STUBS --------------//
IAS_Sound_Processor internal_Sound_Processor;
// No EHS declaration as there are
// no architectural interactions and
// no semi-synchronous interactions.

//----------- DEFINING CONSTRUCTOR ------------//
protected int segment_size;

Sound_Processor(int segment_size) {
  this.segment_size = segment_size;
  defineBehavEquations();
}

//------------ DEFINING BEHAVIOR --------------//
void defineBehavEquations() {

  Segment_Descriptor_Handling =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Segment_Descriptor_Handling((Interface_Effect)actualPars[0],
                                     (boolean)actualPars[1]);
      }

      private void _Segment_Descriptor_Handling(Interface_Effect effect,
                                                boolean just_done) {
        Interface_Segment dry_segment;
        Interface_Segment processed_segment;
        Interface_Descriptor effect_descriptor;
```

```
Object[] inputPars;
switch (
  ElemMeth.choice(
    new ChAct[] {
      new ChAct(true, receive_dry_segment),
      new ChAct(!just_done, send_descriptor_request),
      new ChAct(true, receive_eos)
    }
  )
) // Choice body :
{
  case 0:
    try {
      inputPars = receive_dry_segment.receive();
      dry_segment = (Interface_Segment)inputPars[0] ;
    } catch(SyncException e) {}
    internal_Sound_Processor.process_dry_segment(dry_segment,
                                                  segment_size,
                                                  effect);
    inputPars = internal_Sound_Processor.get_processed_segment();
    processed_segment = (Interface_Segment)inputPars[0] ;
    try {
      send_processed_segment.send(processed_segment.clone());
    } catch(SyncException e) {}
    nextBehavEq = Segment_Descriptor_Handling;
    actualPars = new Object[] {effect,
                               false};
    break;
  case 1:
    try {
      send_descriptor_request.send();
    } catch(SyncException e) {}
    try {
      inputPars = receive_descriptor.receive();
      effect_descriptor = (Interface_Descriptor)inputPars[0];
    } catch(SyncException e) {}
    nextBehavEq = Descriptor_Check;
    actualPars = new Object[] {effect,
                               effect_descriptor};
    break;
  case 2:
    try {
      receive_eos.receive();
    } catch(SyncException e) {}
    try {
      forward_eos.send();
    } catch(SyncException e) {}
    nextBehavEq = Segment_Descriptor_Handling;
    actualPars = new Object[] {effect,
                               false};
    break;
```

```
        default:
          nextBehavEq = null;
          actualPars = null;
      }
    }

  }; // end of behavioral equation Segment_Descriptor_Handling

Descriptor_Check =
  new BehavioralEquationInterface() {

    public void behavEqCall() {
      _Descriptor_Check((Interface_Effect)actualPars[0],
                        (Interface_Descriptor)actualPars[1]);
    }

    private void _Descriptor_Check(Interface_Effect old_effect,
                                   Interface_Descriptor effect_descriptor) {
      switch (
        ElemMeth.choice(
          new ChAct[] {
            new ChAct(effect_descriptor != null, forward_descriptor),
            new ChAct(effect_descriptor == null, null)
          }
        )
      ) // Choice body :
      {
        case 0:
          try {
            forward_descriptor.send(effect_descriptor.clone());
          } catch(SyncException e) {}
          nextBehavEq = Segment_Effect_Handling;
          actualPars = new Object[] {old_effect};
          break;
        case 1:
          internal_Sound_Processor.ignore();
          nextBehavEq = Segment_Descriptor_Handling;
          actualPars = new Object[] {old_effect,
                                     true};
          break;
        default:
          nextBehavEq = null;
          actualPars = null;
      }
    }

  }; // end of behavioral equation Descriptor_Check

Segment_Effect_Handling =
  new BehavioralEquationInterface() {
```

```
public void behavEqCall() {
  _Segment_Effect_Handling((Interface_Effect)actualPars[0]);
}

private void _Segment_Effect_Handling(Interface_Effect old_effect) {
  Interface_Segment dry_segment;
  Interface_Segment processed_segment;
  Interface_Effect effect;
  Object[] inputPars;
  switch (
    ElemMeth.choice(
      new ChAct[] {
        new ChAct(true, receive_dry_segment),
        new ChAct(true, receive_effect),
        new ChAct(true, receive_eos)
      }
    )
  ) // Choice body :
  {
    case 0:
      try {
        inputPars = receive_dry_segment.receive();
        dry_segment = (Interface_Segment)inputPars[0];
      } catch(SyncException e) {}
      internal_Sound_Processor.process_dry_segment(dry_segment,
                                                   segment_size,
                                                   old_effect);
      inputPars = internal_Sound_Processor.get_processed_segment();
      processed_segment = (Interface_Segment)inputPars[0] ;
      try {
        send_processed_segment.send(processed_segment.clone());
      } catch(SyncException e) {}
      nextBehavEq = Segment_Effect_Handling;
      actualPars = new Object[] {old_effect};
      break;
    case 1:
      try {
        inputPars = receive_effect.receive();
        effect = (Interface_Effect)inputPars[0];
      } catch(SyncException e) {}
      nextBehavEq = Segment_Descriptor_Handling;
      actualPars = new Object[] {new_effect,
                                 true};
      break;
    case 2:
      try {
        receive_eos.receive();
      } catch(SyncException e) {}
      try {
        forward_eos.send();
      } catch(SyncException e) {}
```

```
            try {
              inputPars = receive_effect.receive();
              effect = (Interface_Effect)inputPars[0];
            } catch(SyncException e) {}
            nextBehavEq = Segment_Descriptor_Handling;
            actualPars = new Object[] {new_effect,
                                       false};
            break;
          default:
            nextBehavEq = null;
            actualPars = null;
        }
      }

    }; // end of behavioral equation Segment_Effect_Handling

  } // end of method defineBehavEquations()

  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Sound_Processor = null;

  public void start() {
    (th_Sound_Processor = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Sound_Processor.join();
  }

  public void run() {
    internal_Sound_Processor =
      new IAS_Sound_Processor();
    nextBehavEq = Segment_Descriptor_Handling;
    actualPars = new Object[] {null,
                               false};
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}
```

Since the AET `Sound_Processor` does not have unattached ports nor semi-synchronous interactions, no EHS stub class is synthesized for it. Conversely, the stub class `IAS_Sound_Processor` is generated for filling in the internal actions `process_dry_segment`, `get_processed_segment`, and `ignore`:

```
class IAS_Sound_Processor {

  IAS_Sound_Processor() {
```

```
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void process_dry_segment(Interface_Segment dry_segment,
                           int segment_size,
                           Interface_Effect effect) {
    // FILL IN THE METHOD BODY
  }

  Object[] get_processed_segment() {
    Interface_Segment processed_segment = null;
    // FILL IN THE METHOD BODY
    return new Object[] {processed_segment};
  }

  void ignore() {
    // FILL IN THE METHOD BODY
  }

}
```

The last AET of the audio processing system is `Effect_Generator`. The code generated for it is as follows:

```
class Effect_Generator implements RunnableElem
{
  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }

  BehavioralEquationInterface Generation;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
  UniSyncReceiverPort receive_descriptor =
    new UniSyncReceiverPort(this);

  UniSyncSenderPort send_effect =
    new UniSyncSenderPort(this);

  //-------------- DECLARING STUBS --------------//
  IAS_Effect_Generator internal_Effect_Generator;
  // No EHS declaration as there are
  // no architectural interactions and
  // no semi-synchronous interactions.

  //----------- DEFINING CONSTRUCTOR ------------//
  Effect_Generator() {
    defineBehavEquations();
  }
```

```
//----------- DEFINING BEHAVIOR -------------//
void defineBehavEquations() {

  Generation =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Generation();
      }

      private void _Generation() {
        Interface_Descriptor effect_descriptor;
        Interface_Effect effect;
        Object[] inputPars;
        try {
          inputPars = receive_descriptor.receive();
          effect_descriptor = (Interface_Descriptor)inputPars[0];
        } catch(SyncException e) {}
        internal_Effect_Generator.create_new_effect(effect_descriptor);
        inputPars = internal_Effect_Generator.get_new_effect();
        effect = (Interface_Effect)inputPars[0];
        try {
          send_effect.send(effect.clone());
        } catch(SyncException e) {}
        nextBehavEq = Generation;
        actualPars = null;
      }

    }; // end of behavioral equation Generation

}

//-------- RUNNING ELEMENT [thread] ----------//
Thread th_Effect_Generator = null;

public void start() {
  (th_Effect_Generator = new Thread(this)).start();
}

public void join() throws InterruptedException {
  th_Effect_Generator.join();
}

public void run() {
  internal_Effect_Generator =
    new IAS_Effect_Generator();
  nextBehavEq = Generation;
  actualPars = null;
  while (nextBehavEq != null)
    nextBehavEq.behavEqCall();
```

```
  }

} // end of runnable element class Effect_Generator
```

As the previous AET, `Effect_Generator` does not have unattached ports nor semi-synchronous interactions, then no EHS stub class must be synthesized for it. The stub class `IAS_Effect_Generator` is generated instead for filling in the internal actions `create_new_effect` and `get_new_effect`:

```
class IAS_Effect_Generator {

  IAS_Effect_Generator() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void create_new_effect(Interface_Descriptor effect_descriptor) {
    // FILL IN THE METHOD BODY
  }

  Object[] get_new_effect() {
    Interface_Effect effect = null;
    // FILL IN THE METHOD BODY
    return new Object[] {effect};
  }

}
```

Chapter 6

Monitor Synthesis

The third phase of our approach to the generation of multithreaded object-oriented code from process algebraic architectural descriptions deals with the translation of the algebraically-specified behavior of some AETs into monitor classes. In Chap. 5 we observed that the natural candidate for the target of the translation of the process algebraic description of a component is a thread. The approach proposed in the previous chapter, in fact, allows the generation of Java threads from any AET occurring in a PADL description. In many cases, however, it may be more efficient to synthesize some software components as monitors rather than threads. In effect, the performance of the generated code may be improved thanks to the synthesis of monitors as they reduce the thread context switch frequency. Moreover the presence of monitors results in a lightweight concurrency control management with respect to package `Sync`, with the monitors themselves constituting explicit coordination areas that were not available in the previous phases.

What is proposed in this chapter is a general methodology that accompanies the translation process, which in particular should help understanding when and how it is possible to implement a software component as a monitor instead of a thread. A systematic approach is illustrated here for the synthesis of correctly coordinating Java monitors from arbitrary process algebraic component descriptions that satisfy some suitable constraints. As will be discussed in Sect. 6.1, the constraints are related to the fact that a monitor is a passive entity, which typically encapsulates data in a way that guarantees a mutually exclusive

access. In other words, a monitor coordinates the access of the threads to its methods, but its statements are executed by those threads.

Once the above mentioned constraints are satisfied, the process algebraic description of a component can systematically be transformed into a canonical form that we call *monitor normal form* (Sect. 6.2) from which it is easy to synthesize a Java monitor (Sect. 6.3 and 6.4).

The constraints and the approach will be illustrated in the last section of this chapter (Sect. 6.5) by means of the process algebraic specification of the console of the audio processing system introduced in Sect. 3.5.

6.1 Monitor Constraints

In our process algebraic view, both thread and monitor classes should be modeled as AETs. From the code generation viewpoint, threads and monitors have to interact with each other as regulated by the mechanisms provided by the object-oriented target language, i.e. Java. Here, we present a set of constraints under which it is possible to synthesize a correctly coordinating monitor from the process algebraic description of a software component. Before doing so, it is worth to introduce some terminology, then to recall the way in which, in general, threads and monitors interact, and to illustrate the specific interaction model we adopted in order to carry out the third phase of our approach.

6.1.1 Terminology

An architectural element type representing a Java class that extends or implements a thread base class will be called *native-thread type* and will be translated into a *native-thread component*. An architectural element type representing a Java monitor class will instead be called *monitor type* and will be translated into a *monitor component*.

Furthermore, we distinguish between two kinds of component interactions at the Java code level. An *active-control interaction* is performed by a component whenever it starts communicating with another component. A *passive-control interaction* is executed by a component whenever it is waiting for another

component to start communicating with it. In particular, entry and exit points of monitor components will be given by passive-control interactions.

6.1.2 Thread-Monitor Interaction Model

Given a native-thread component T and a monitor component M, the communication between them takes place by means of the component control switch depicted in the sequence diagram of Fig. 6.1. When T intends to communicate with M, T invokes a synchronized method of M – which corresponds to performing an active-control interaction – so that thread t leaves T and waits until M is ready to communicate.

More precisely, in a synchronous model, t waits outside M if another thread is currently running inside M, otherwise it immediately enters M and possibly blocks. This happens when t has to wait for a notification related to a condition synchronization of M that does not hold upon entering M. In a semi-synchronous model, instead, an exception is raised if a condition synchronization for t does not hold and either there is no thread running inside M or the thread currently running inside M leaves it without notifying such a condition synchronization. We recall from [52] that a condition synchronization permits a monitor to block threads until a particular condition holds, such as e.g. a count becoming non-zero, a buffer becoming empty, or new input becoming available.

When M is ready, t takes the control of M and executes a sequence of statements of M corresponding to internal actions. Finally, t possibly notifies one of the threads blocked inside M about the validity of a condition synchronization, then leaves the monitor. The end of the above mentioned statement sequence coincides either with the monitor termination or with the execution of the last statement before a passive-control interaction.

In order to achieve a correct concurrency control, it suffices that the thread taking the control of the monitor executes finitely many statements without moving to another monitor or invoking a method of another thread before leaving the monitor in which it is running. In this way we are guaranteed that a thread will stay within the monitor for a finite amount of time (up to possible condition

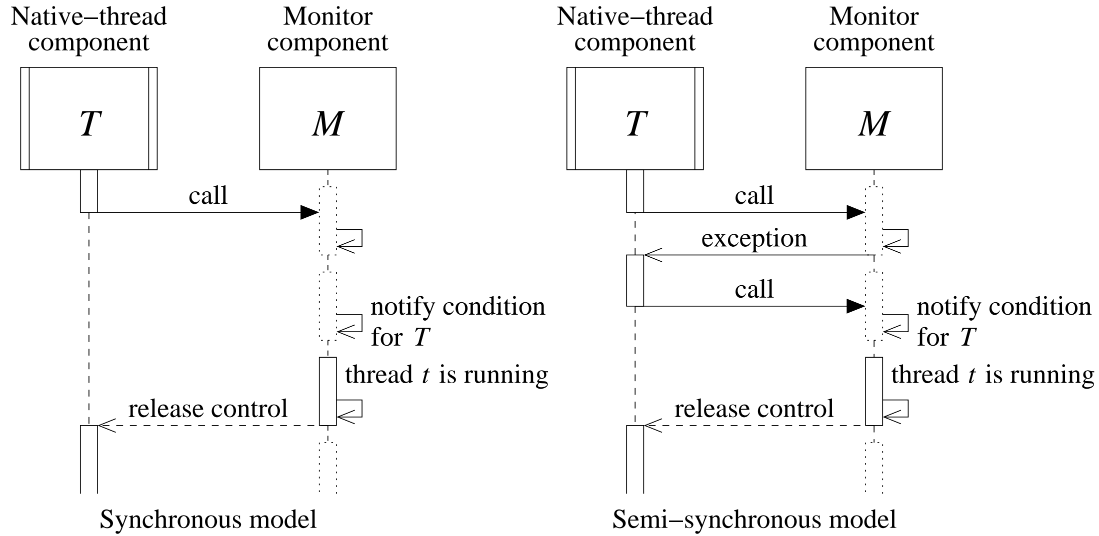

Figure 6.1. Component control switch from native-thread T to monitor M

synchronizations that will never hold) and will not cause any interference between the monitor and other monitors.

Below we establish the constraints that must be satisfied by the process algebraic description of a component in order for it to be synthesized as a correctly coordinating monitor.

6.1.3 Constraint 1: No Cycles of Internal Actions

Since a monitor is a passive entity that coordinates other components, it is desirable that a thread taking the control of the monitor runs inside the monitor only for a finite amount of time. In the worst case, it may happen that the thread blocks forever inside the monitor because of a condition synchronization that will never hold. However, this does not prevent other threads from entering the monitor and running.

In order to achieve finiteness, we need to enforce that the maximum number of consecutive internal actions that can be performed inside a candidate monitor type is finite. This can easily be checked on the process algebraic description of a candidate monitor type by verifying the absence of cycles of internal actions.

With the same approach followed in Sect. 5, each internal action will be

translated into a method of an IAS class to be manually filled in later on. If we adhere to the guidelines of Sect. 5.7, according to which non-terminating statements should be avoided within these methods, a finite sequence of internal actions will be executed in a finite amount of time. In this way the absence of cycles of internal actions proved at the process algebraic level is guaranteed to be preserved at the Java code level.

6.1.4 Constraint 2: No Attached Monitor Type Instances

A thread taking the control of a monitor should not move to another monitor before leaving the monitor in which it is running, otherwise interferences among monitors may arise.

This can be achieved by requiring that no instance of the monitor type can communicate with instances of other monitor types. This can trivially be verified by examining the attachments in the topological section of the process algebraic architectural description in which the monitor type is defined.

6.1.5 Constraint 3: No Non-Synchronous Interactions

Since a monitor is a passive entity, it cannot perform active-control interactions. A monitor can only passively communicate with a thread in a synchronous way and a thread can communicate only synchronously or semi-synchronously with a monitor.

This can trivially be verified on the process algebraic descriptions of a candidate monitor type and of the native-thread types whose instances are attached to instances of the monitor type, through the qualifiers expressing the synchronicity of the communications in which those instances are involved.

6.1.6 Constraint 4: No Non-Disjoint Hybrid Choices

A *hybrid choice* in the process algebraic description of a component is a choice between a non-empty set of interactions and a non-empty set of internal actions. The problem with hybrid choices is that they may hamper the detection of the

action sequence corresponding to the statement sequence that should be executed by a thread running inside a monitor.

In fact, recalled that the monitor entry and exit points are described through passive control interactions, to automatically detect the beginning and the end of the action sequence in a candidate monitor type we need that the sequence is comprised between two passive-control interactions. A choice between a passive-control interaction and an internal action would make it impossible to decide whether the currently running thread has completed its task or not, unless the two actions are preceded by two disjoint conditions.

As a consequence of this constraint, a candidate monitor type can contain only choices among all interactions or all internal actions. This can easily be checked at the process algebraic description level. In addition, hybrid choices are admitted provided that the involved actions are pairwise preceded by disjoint boolean conditions (with each pair being composed by an interaction and an internal action). As we shall see in Sect. 6.2.4, this constraint complies with Lemma 6.3 by guaranteeing the correctness of the transformation of a candidate monitor type into its relative monitor normal form.

6.2 Syntactic Transformation into Monitor Normal Form

Once all the constraints defined in the previous section are satisfied by the process algebraic description of a candidate monitor type, it is possible to proceed to the transformation of the description itself into monitor normal form. Starting from this canonical form, it will be easy to synthesize the Java implementation of the monitor type.

The basic idea behind the monitor normal form is to rewrite the process algebraic description of the monitor type in such a way that all the occurrences of interactions are collected into a single equation. At the code level each of them will correspond to a passive-control interaction. Therefore, if we place all of them at the beginning of a different branch of a choice, we exactly characterize the point at which the monitor is waiting for a thread to take its control.

The process algebraic description in monitor normal form obtained at the end of the rewriting process will be formed by:

- An *interacting choice equation*, which is a choice composed by as many branches as there are occurrences of interactions, with each branch starting with an interaction followed only by internal actions and invocations.
- A group of *setting equations*, which contain the replacement of the branches of the original defining equations copied to the interacting choice equation with an invocation of the latter equation with suitably set parameters.
- A group of *internal equations*, which are the initial parts of the original defining equations that started with an internal action or a choice among internal actions. Original defining equations that does not contain any interactions are internal equations, and are leaved unchanged by the rewriting process.

This monitor normal form can be achieved through a sequence of five steps:

1. rewriting complex choices;
2. splitting defining equations;
3. building the interacting choice equation;
4. building setting equations;
5. rearranging the interacting choice equation.

These steps will be illustrated on the process algebraic description of the AET `Console` of the audio processing system introduced in Sect. 3.5. As anticipated in Sect. 5.8, in fact, the `Console` could be implemented as a Java monitor instead of a thread. In order to do so, now we know we must assess the constraints that guarantee the derivability of a monitor from the description of the `Console`. If this stage succeeds, we can proceed in rewriting the process algebraic description of the `Console` into the corresponding monitor normal form.

But it is very easy to check that the Console satisfies all the monitor constraints. In fact, its only internal actions are store_config and get_summary_descriptor and the process algebraic description of its behavior has no cycles involving only occurrences of these two actions. If all the other AETs except for at most Effect_Generator are synthesized as threads, it has no monitor type instances attached to its only instance C. All of its interactions are synchronous and no interaction of the AEIs attached to its only instance C is asynchronous. Finally, the process algebraic description of its behavior has no hybrid choices.

6.2.1 Step 1: Rewriting Complex Choices

The first step is a preliminary step whose purpose is to facilitate the handling of the branches of the choices in the subsequent steps. If a choice among interactions is written in an abbreviated notation, such a choice must be expanded. Likewise, if a nested choice among interactions occurs, such a choice must be flattened. Possible nested conditions occurring before a branch of the original choice, in flattened choices will be logically conjuncted into a single condition guarding the same branch.

In the description of Console, the outer choice of defining equation Config_Handling contains a nested choice (the choice between two occurrences of receive_descriptor_request). The equivalent flattened choice obtained after the application of the first step is the following:

```
choice
{
  cond(config_changes < allowed_changes) ->
    receive_config?(console_config) . store_config!(console_config) .
      Config_Handling(config_changes + 1),
  cond(config_changes > 0) ->
    receive_descriptor_request . Descriptor_Handling(),
  cond(config_changes = 0) ->
     receive_descriptor_request . send_descriptor!(null) . Config_Handling(0),
  receive_stop . forward_stop . Start(config_changes)
}
```

Lemma 6.1 *Let M be the original process algebraic description of a monitor type and M_1 be the output of step 1. Then the labeled transition system underlying M_1 is isomorphic to the labeled transition system underlying M.*

Proof: Trivial, because step 1 simply expands abbreviated choices or flattens nested choices. ∎

6.2.2 Step 2: Splitting Defining Equations

The second step splits any defining equation at each point in which it contains an interaction or a (flat) choice among interactions that does not occur at the beginning of the equation. This is necessary for building the interacting choice equation, as each of its branches must start with an interaction.

Each inner interaction or choice among interactions is moved together with what follows it into a new equation. The moved block is replaced by an invocation of the new equation. At the end of this splitting process, every interaction will occur only at the beginning of some equation and will be followed only by internal actions and invocations.

All the defining equations of `Console` need to be split as they contain some inner interaction. The equation `Start` thus becomes:

```
Start(integer(0..allowed_changes) config_changes := 0;
      void) =
  receive_start . Split_1_Start(config_changes);

Split_1_Start(integer(0..allowed_changes) config_changes;
              void) =
  forward_start . Config_Handling(config_changes);
```

while the equation `Config_Handling` becomes:

```
Config_Handling(      integer(0..allowed_changes) config_changes;
                local object(Configuration) console_config) =
  choice
  {
    cond(config_changes < allowed_changes) ->
      receive_config?(console_config) . store_config!(console_config) .
          Config_Handling(config_changes + 1),
    cond(config_changes > 0) ->
      receive_descriptor_request . Descriptor_Handling(),
    cond(config_changes = 0) ->
      receive_descriptor_request . Split_1_Config_Handling(),
    receive_stop . Split_2_Config_Handling(config_changes)
```

```
  };

Split_1_Config_Handling(void;
                        void) =
  send_descriptor!(null) . Config_Handling(0);

Split_2_Config_Handling(integer(0..allowed_changes) config_changes;
                        void) =
  forward_stop . Start(config_changes);
```

and, finally, the equation `Descriptor_Handling` becomes:

```
Descriptor_Handling(void;
                    local object(Descriptor) effect_descriptor) =
  get_summary_descriptor?(effect_descriptor) .
    Split_1_Descriptor_Handling(effect_descriptor);

Split_1_Descriptor_Handling(object(Descriptor) effect_descriptor;
                            void) =
  send_descriptor!(effect_descriptor) . Config_Handling(0);
```

Note that the formal parameters and the local variables declared in the original equations are properly redeclared and passed throughout the splitted ones.

After this step the internal equations, i.e. `Descriptor_Handling`, contain only internal actions while the other, non-internal, equations start with an interaction or a choice among interactions.

Lemma 6.2 *Let M_2 be the output of step 2. Then the labeled transition system underlying M_2 is isomorphic to the labeled transition system underlying M_1.*
Proof: In the simplest case we split a branch of an original defining equation that contains an interaction not occurring at the beginning of the branch itself. More precisely, if a branch of an equation E is of the form:

$$\beta_1.\beta_2.\ \ldots\ .\beta_n.\alpha.P$$

where β_k is an internal action for each $k = 1, \ldots, n$ and α is an interaction, then the following new equation is defined:

$$Split_E = \alpha.P$$

and the branch is rewritten into:

$$\beta_1.\beta_2.\ \ldots\ .\beta_n.Split_E$$

The rewritten branch and the new equation considered as a whole are clearly

isomorphic to the original branch. Similar is the case in which there is a choice among interactions instead of a single interaction α. ■

6.2.3 Step 3: Building the Interacting Choice Equation

The third step builds the interacting choice equation by suitably merging into a single equation all the branches of the non-internal equations obtained at the end of the second step.

In order to preserve the semantics of the original defining equations, the interacting choice equation needs to keep track of the current state of the monitor type. This state can be encoded through a bounded integer parameter `nieq`, which represents the non-internal equation describing the current behavior, and a set of boolean parameters `g_`, which represent the guards expressing the enabledness of the interactions occurring at the beginning of the considered non-internal equation.

We observe that parameter `nieq` is strictly necessary because the same set of interactions may be enabled in several different non-internal equations. By contrast, the guards `g_` are useful – once the monitor normal form has been achieved – to implement the condition synchronizations of the monitor.

If a non-internal equation starts with a single interaction, then the whole right-hand side of the equation becomes a branch of the interacting choice equation. Similarly, if it starts with a (flat) choice among interactions, then each branch of the choice becomes a branch of the interacting choice equation. Finally, if it starts with a (flat) disjoint hybrid choice, only the branches starting with an interaction become branches of the interacting choice equation. [1]

Each branch of the interacting choice equation is preceded by a boolean expression, which is the logical conjunction of:

- The equality check for `nieq` to correspond to the value associated with the non-internal equation that contained the considered branch.
- The guard `g_` associated with the interaction at the beginning of the branch.

[1] We will see that this does not disrupt the semantics of the disjoint hybrid choice.

- A boolean condition possibly inherited from the original branch.

All the equations obtained for `Console` at the end of the second step are non-internal except for `Descriptor_Handling`. The non-internal equations are numbered 0 to 5 according to the order in which they appear in the sequence and their branches are merged into the following interacting choice equation:

```
Inter_Ch_Eq(      integer(0..5)                nieq,
                  boolean                      g_receive_start,
                  boolean                      g_receive_config,
                  boolean                      g_receive_descriptor_request,
                  boolean                      g_receive_stop,
                  boolean                      g_forward_start,
                  boolean                      g_send_descriptor,
                  boolean                      g_forward_stop,
                  integer(0..allowed_changes)  config_changes,
                  object(Descriptor)           effect_descriptor;
            local object(Configuration)        console_config) =
  choice
  {
    cond((nieq = 0) && g_receive_start) ->
      receive_start . Split_1_Start(config_changes),
    cond((nieq = 1) && g_forward_start) ->
      forward_start . Config_Handling(config_changes),
    cond((nieq = 2) && g_receive_config && (config_changes < allowed_changes)) ->
      receive_config?(console_config) . store_config!(console_config) .
        Config_Handling(config_changes + 1),
    cond((nieq = 2) && g_receive_descriptor_request && (config_changes > 0)) ->
      receive_descriptor_request . Descriptor_Handling(),
    cond((nieq = 2) && g_receive_descriptor_request && (config_changes = 0)) ->
      receive_descriptor_request . Split_1_Config_Handling(),
    cond((nieq = 2) && g_receive_stop) ->
      receive_stop . Split_2_Config_Handling(config_changes),
    cond((nieq = 3) && g_send_descriptor) ->
      send_descriptor!(null) . Config_Handling(0),
    cond((nieq = 4) && g_forward_stop) ->
      forward_stop . Start(config_changes),
    cond((nieq = 5) && g_send_descriptor) ->
      send_descriptor!(effect_descriptor) . Config_Handling(0)
  }
```

From the point of view of the correctness of the transformation, this step has to be considered together with the next one.

6.2.4 Step 4: Building Setting Equations

The fourth step completes the work done in the previous step as a consequence of the fact that the interacting choice equation does not replace the non-internal equations. This step is necessary because invocations of those equations are still present both in the interacting choice equation and in the internal equations.

The right-hand side of each non-internal equation is rewritten in such a way that all of its branches that have been copied to the interacting choice equation are replaced by a single invocation of the latter equation with suitably set actual values for parameters `nieq` and `g_`. For this reason we refer to each of these rewritten equations as a setting equation.

The actual value of `nieq` passed to the interacting choice equation is the value associated with the non-internal equation. The actual values of the boolean guards are set as follows. If an interaction does not occur at the beginning of any copied branch of the non-internal equation, then the corresponding guard `g_` is set to false. Instead, if it occurs there and at least one of its occurrences is not guarded by any boolean condition, the corresponding guard `g_` is set to true. Finally, if it occurs there and all of its occurrences are guarded by some boolean condition, then the corresponding guard `g_` is set to the disjunction of these conditions (if at least one of them holds true, then the interaction is enabled).

If a non-internal equation starts with a single interaction or a (flat) choice among interactions, its right-hand side is entirely replaced by an invocation of the interacting choice equation with the actual values for `nieq` and `g_` set as explained above. Instead, if a non-internal equation starts with a (flat) disjoint hybrid choice, all the internal branches are preserved. By contrast, the interacting branches are replaced by an invocation of the interacting choice equation, preceded by the disjunction of all the boolean conditions associated with them. In this way the semantics of the selection between the group of internal branches and the group of interacting branches is preserved, with the selection within the latter group being deferred to the interacting choice equation.

The non-internal equations of `Console` are transformed into the following setting equations:

```
Start(integer(0..allowed_changes) config_changes := 0;
      void) =
  Inter_Ch_Eq(0,
              true, false, false, false, false, false, false,
              config_changes,
              null);

Split_1_Start(integer(0..allowed_changes) config_changes;
              void) =
  Inter_Ch_Eq(1,
              false, false, false, false, true, false, false,
              config_changes,
              null);

Config_Handling(integer(0..allowed_changes) config_changes;
                void) =
  Inter_Ch_Eq(2,
              false,
              (config_changes < allowed_changes),
              ((config_changes > 0) || (config_changes = 0)),
              true, false, false, false,
              config_changes,
              null);

Split_1_Config_Handling(void;
                        void) =
  Inter_Ch_Eq(3,
              false, false, false, false, false, true, false,
              0,
              null);

Split_2_Config_Handling(integer(0..allowed_changes) config_changes;
                        void) =
  Inter_Ch_Eq(4,
              false, false, false, false, false, false, true,
              config_changes,
              null);

Split_1_Descriptor_Handling(object(Descriptor) effect_descriptor;
                            void) =
  Inter_Ch_Eq(5,
              false, false, false, false, false, true, false,
              0,
              effect_descriptor)
```

Lemma 6.3 *Let $M_{3,4}$ be the output of the steps 3 and 4. Then the labeled transition system underlying $M_{3,4}$ is isomorphic to the labeled transition system underlying M_2.*

Proof: The outcome of step 3 is the following interacting choice equation:

$$
\begin{array}{l}
Inter_Ch_Eq(integer(0..n-1)\ nieq, boolean\ g_\alpha_0, \ldots, boolean\ g_\alpha_{m-1}) = \\
\quad choice \\
\quad \{ \\
\qquad cond((nieq = 0)\ \&\&\ g_\alpha_{0,1}\ \&\&\ c_{0,1}) \ \rightarrow \ \alpha_{0,1}.P_{0,1}, \\
\qquad \vdots \\
\qquad cond((nieq = i)\ \&\&\ g_\alpha_{i,j}\ \&\&\ c_{i,j}) \ \rightarrow \ \alpha_{i,j}.P_{i,j}, \\
\qquad \vdots \\
\qquad cond((nieq = n-1)\ \&\&\ g_\alpha_{n-1,b_{n-1}}\ \&\&\ c_{n-1,b_{n-1}}) \ \rightarrow \ \alpha_{n-1,b_{n-1}}.P_{n-1,b_{n-1}} \\
\quad \}
\end{array}
$$

where:

- n is the number of non-internal equations of M_2;
- m is the number of interactions occurring in those non-internal equations;
- b_i is the number of interacting branches of non-internal equation i ($0 \leq i \leq n-1$);
- $\alpha_{i,j} \in \{\alpha_0, \ldots, \alpha_{m-1}\}$ is the interaction occurring at the beginning of branch j ($0 \leq j \leq b_i - 1$) of non-internal equation i ($0 \leq i \leq n-1$);
- $c_{i,j}$ is the boolean condition possibly inherited from branch j ($0 \leq j \leq b_i - 1$) of non-internal equation i ($0 \leq i \leq n-1$).

In step 4 non-internal equation i ($0 \leq i \leq n-1$) is rewritten into a setting equation with the following right-hand side part:

$$
\begin{array}{l}
choice \\
\{ \\
\quad cond(\bigvee_{j=0}^{b_i-1} c_{i,j}) \ \rightarrow \\
\qquad Inter_Ch_Eq(i, \bigvee_{j \in \{0..b_i-1 \mid \alpha_{i,j} = \alpha_0\}} c_{i,j}, \ldots, \bigvee_{j \in \{0..b_i-1 \mid \alpha_{i,j} = \alpha_{m-1}\}} c_{i,j}), \\
\quad cond(c_{i,b_i}) \ \rightarrow \ \beta_{i,b_i}.P_{i,b_i}, \\
\qquad \vdots \\
\quad cond(c_{i,b_i+d_i-1}) \ \rightarrow \ \beta_{i,b_i+d_i-1}.P_{i,b_i+d_i-1} \\
\}
\end{array}
$$

where:

- the first branch replaces all the interacting branches of non-internal equation i, which have been copied to the interacting choice equation;
- $\bigvee_{j=0}^{b_i-1} c_{i,j}$ is the disjunction of the boolean conditions of all the interacting branches of non-internal equation i;
- $\bigvee_{j\in\{0..b_i-1 \mid \alpha_{i,j}=\alpha_k\}} c_{i,j}$ is the disjunction of the boolean conditions of the interacting branches starting with interaction α_k $(0 \leq k \leq m-1)$;
- d_i is the number of internal branches of non-internal equation i ($d_i = 0$ if there are no such branches).

Note that the last monitor constraint guarantees the disjointness of the set of boolean conditions associated with the interacting branches from the set of boolean conditions associated with the internal branches:

$$\bigvee_{j=0}^{b_i-1} c_{i,j} \wedge \bigvee_{j=b_i}^{b_i+d_i-1} c_{i,j} = \mathit{false}$$

The interacting choice equation and the group of setting equations of $M_{3,4}$ considered as a whole are clearly isomorphic to the group of non-internal equations of M_2. ■

6.2.5 Step 5: Rearranging the Interacting Choice Equation

The fifth step performs a number of operations on the interacting choice equation. On the one hand, the branches are lexicographically sorted on the basis of their starting interactions. This sorting aims at facilitating code generation, as all the branches starting with the same interaction shall be translated into a single synchronized method (corresponding to a passive-control interaction) of a Java monitor class.

On the other hand, some optimizations are useful to simplify the structure of the interacting choice equation and thus of the monitor to be synthesized. First,

if all the branches starting with the same interaction are associated with a single value of `nieq`, the check on `nieq` can be removed from those branches. The reason is that such an interaction occurred only in a single non-internal equation, hence the guard `g_` associated with it can be true only when `nieq` has the considered value.

Second, if an interaction occurs at the beginning of only one of the branches associated with a specific value of `nieq` and that branch is preceded by an inherited boolean condition, that condition can be removed. In fact, the same condition is already contained in the actual value for the guard `g_` passed by the setting equation associated with the considered branch.

Third, if several branches are identical up to their boolean expressions – i.e. different values of `nieq` or different inherited boolean conditions – these branches can be collapsed into a single one. The new branch will be preceded by a boolean expression that includes, besides the check on `g_`, the disjunction of the checks on the different values of `nieq` and the disjunction of the inherited conditions. If the interaction occurs only at the beginning of this new branch, by virtue of the first two optimizations the two above-mentioned disjunctions can be removed.

In the case of the interacting choice equation of `Console`, the first optimization can be applied to all the branches except for the two branches starting with `send_descriptor`. Among the three branches preceded by an inherited boolean condition, the second optimization can be applied only to the branch starting with `receive_config`. The third optimization does not come into play for the two branches starting with `receive_descriptor_request` and for the two branches starting with `send_descriptor`, as they are slightly different. The rearranged interacting choice equation is as follows:

```
Inter_Ch_Eq(       integer(0..5)                nieq,
                   boolean                      g_receive_start,
                   boolean                      g_receive_config,
                   boolean                      g_receive_descriptor_request,
                   boolean                      g_receive_stop,
                   boolean                      g_forward_start,
                   boolean                      g_send_descriptor,
                   boolean                      g_forward_stop,
                   integer(0..allowed_changes)  config_changes,
                   object(Descriptor)           effect_descriptor;
             local object(Configuration)        console_config) =
```

```
choice
{
  cond(g_forward_start) ->
    forward_start . Config_Handling(config_changes),
  cond(g_forward_stop) ->
    forward_stop . Start(config_changes),
  cond(g_receive_config) ->
    receive_config?(console_config) . store_config!(console_config) .
      Config_Handling(config_changes + 1),
  cond(g_receive_descriptor_request && (config_changes > 0)) ->
    receive_descriptor_request . Descriptor_Handling(),
  cond(g_receive_descriptor_request && (config_changes = 0)) ->
    receive_descriptor_request . Split_1_Config_Handling(),
  cond(g_receive_start) ->
    receive_start . Split_1_Start(config_changes),
  cond(g_receive_stop) ->
    receive_stop . Split_2_Config_Handling(config_changes),
  cond((nieq = 3) && g_send_descriptor) ->
    send_descriptor!(null) . Config_Handling(0),
  cond((nieq = 5) && g_send_descriptor) ->
    send_descriptor!(effect_descriptor) . Config_Handling(0)
}
```

Lemma 6.4 *Let M' be the output of step 5, i.e. the process algebraic description in monitor normal form corresponding to M. Then the labeled transition system underlying M' is isomorphic to the labeled transition system underlying $M_{3,4}$.*

Proof: Observed that sorting the branches of the interacting choice equation does not alter the semantics at all, let us consider the three optimizations. The first optimization takes place whenever all the branches starting with the same interaction α_k $(0 \leq k \leq m-1)$ are associated with a single value h $(0 \leq h \leq n-1)$ of *nieq*. In fact, whenever a setting equation $i \neq h$ invokes the interacting choice equation, then $\bigvee_{j \in \{0..b_i-1 | \alpha_{i,j} = \alpha_k\}} c_{i,j} = \mathit{false}$ because the index set of the disjunction is empty. This means that $i \neq h$ implies $g_\alpha_k = \mathit{false}$. Instead, when $i = h$ and $\alpha_{i,j} = \alpha_k$ for some $j = 0..b_i - 1$, the check on *nieq* becomes redundant because $(\mathit{nieq} = i)$ && $g_\alpha_{i,j}$ coincides with $g_\alpha_{i,j}$.

The second optimization takes place whenever an interaction α_k occurs at the beginning of only one of the branches associated with a specific value i $(0 \leq i \leq n-1)$ of *nieq* and that branch is preceded by an inherited boolean condition $c_{i,j'}$ $(0 \leq j' \leq b_i - 1)$. In this case, for setting equation i it holds $\bigvee_{j \in \{0..b_i-1 | \alpha_{i,j} = \alpha_k\}} c_{i,j} = c_{i,j'}$, so $g_\alpha_{i,j'}$ is set to $c_{i,j'}$ when this setting equation invokes the interacting choice equation. Therefore $g_\alpha_{i,j'}$ && $c_{i,j'}$ coincides with

$c_{i,j'}$ && $c_{i,j'}$ and hence the check on $c_{i,j'}$ becomes redundant.

The third optimization takes place whenever several branches are identical up to their boolean expressions. In the simplest case we have two branches like the following:

$$\begin{array}{lcl} cond((nieq = i)\ \&\&\ g_\alpha_{i,j}\ \&\&\ c_{i,j}) & \rightarrow & \alpha_{i,j}.P_{i,j} \\ cond((nieq = h)\ \&\&\ g_\alpha_{h,l}\ \&\&\ c_{h,l}) & \rightarrow & \alpha_{h,l}.P_{h,l} \end{array}$$

where $\alpha_{i,j} = \alpha_{h,l} = \alpha_k$ and $P_{i,j} = P_{h,l} = P$. The resulting collapsed branch:

$$cond(((nieq = i)\ ||\ (nieq = h))\ \&\&\ g_\alpha_k\ \&\&\ (c_{i,j}\ ||\ c_{h,l})) \ \rightarrow\ \alpha_k.P$$

is trivially isomorphic to the two original branches considered as a whole. The same argument applies to an arbitrary number of branches that are identical up to their boolean expressions. ■

6.2.6 Correctness of the Transformation

The syntactic transformation of the process algebraic description of a monitor type into monitor normal form is correct in the following sense:

Theorem 6.1 *Let M be the original process algebraic description of a monitor type and let M' be the process algebraic description of the monitor normal form obtained by applying to M the syntactic transformation. Then the labeled transition system underlying M' is isomorphic to the labeled transition system underlying M.*

Proof: It follows from the Lemmas 6.1, 6.2, 6.3, and 6.4. ■

6.3 Generating the Core Monitor Class

The application of the steps illustrated in the previous section allows the process algebraic description of any AET satisfying the monitor constraints to be rewritten into its semantically equivalent monitor normal form. In this section we show how to synthesize a Java monitor class from a process algebraic description in monitor normal form. We will refer to this class as the *core monitor class* to distinguish it from the *monitor wrapper class*. The latter, synthesized in order

to make a monitor component fully compliant with the communication model described in Sect. 4.1, will be explained in the Sect. 6.4.

```
class ◁architectural element type name▷_Monitor {
  ◁Declaring Stubs▷
  ◁Defining Constructor▷
  ◁Defining Behavior▷
  ◁Starting Monitor▷
}
```

Table 6.1. Structure of a core monitor class

The core monitor class is structured as shown in Table 6.1. The first section, *Declaring Stubs*, declares an object of an external IAS class to be manually filled in later on for translating the internal actions occurring in the AET definition. We point out that no EHS classes are declared for a monitor. In fact, exceptions of type `UnattachedPortException` do not need to be handled. The reason is that a monitor can only have passive-control interactions triggered by active-control interactions of threads, so if one of such monitor interactions is not attached it will never be activated. Likewise, exceptions of type `NotReadyPortException` do not need to be handled because all the monitor interactions must be synchronous. However, we will see that exceptions may be raised by a monitor if a thread interacts semi-synchronously with it.

The second section, *Defining Constructor*, defines the class constructor together with its parameters. This section starts with the declaration/definition of some non-public members. First of all we have the declaration of a private integer variable `nieq` and of a private boolean array `guard[]`, which translate the corresponding parameters of the interacting choice equation. Then we have the definition of some integer symbolic constants, which represent the values that `nieq` can take on and the values that an index for `guard[]` can take on. Additional protected members are finally declared, which translate the formal parameters of the AET and all the other parameters occurring in the interacting choice equation. The constructor simply assigns the value of its parameters to the homonymous protected members.

The third section, *Defining Behavior*, starts with the definition of a private method called `checkGuard()`, which checks the condition synchronizations in order to decide whether a thread can enter the monitor or not. Then this section contains the translation of the setting and internal equations as a set of non-public methods, followed by the translation of the interacting choice equation as a set of public synchronized methods, each corresponding to a different interaction occurring in the AET definition.

Finally, the fourth section, *Starting Monitor*, defines the public method `startMonitor()`, which instantiates the external IAS object and invokes the method corresponding to the first (setting or internal) equation occurring in the description of the AET in monitor normal form. This equation corresponds to the first equation in the original description of the AET, hence it is the first one to be executed.

The synthesis of the monitor is accomplished through a sequence of five steps:

1. translating internal actions into stub class methods;
2. declaring IAS stub and synthesizing the monitor class constructor;
3. translating setting and internal equations;
4. translating the interacting choice equation;
5. synthesizing the starting method;

which guide the automated generation of the Java code.

6.3.1 Translating Internal Actions into Stub Class Methods

With the same approach followed in Sect. 5, each internal action of the process algebraic description of a monitor type will be synthesized in the Java monitor class as an invocation of a method defined in an external IAS stub class. In this way the software developer can subsequently fill in the methods associated with the internal actions, without any intervention on the main monitor class.

6.3.2 Declaring IAS Stub and Synthesizing the Monitor Class Constructor

Besides the definition of the constructor, at the beginning of the monitor class there is the declaration/definition of some non-public members. The first member is an object of the stub class for the internal actions, IAS, that will be instantiated by the method `startMonitor()`. Then, an integer variable `nieq` and a boolean array `guard[]` are declared as private, which translate the relative parameters of the interacting choice equation, together with the definition of some integer constants associated with the setting equations, which represent the values that `nieq` can take on and the values that an index for `guard[]` can take on. Additional protected members are declared that translate the formal parameters of the AET and of all the other parameters occurring in the interacting choice equation. The constructor simply assigns the value of its parameters to the homonymous protected members of the class translating the AET formal parameters.

6.3.3 Translating Setting and Internal Equations

The setting and internal equations of the monitor normal form are translated into non-public methods of the Java monitor class. Since these equations do not contain interactions, only sequences of/choices among internal actions have to be considered during their translation.

While a sequence of internal actions can easily be synthesized as a sequence of invocations of the associated stub methods, a choice among internal actions has to be treated carefully. In fact, even if the branches of the choice can be guarded by some conditions, it is not necessarily the case that such conditions are disjoint. In this case the static method `ElemMeth.choice()` provided by the package `Sync`, and introduced in Sect. 5.6, is used and a randomly generated index is employed in a `switch-case` statement to select the translation of a branch of the choice, among the branches whose conditions hold true.

An invocation of a setting or internal equation is simply translated into an invocation of the monitor class method translating the equation itself. As far as the setting equations are concerned, each of them always contains an invocation of the interacting choice equation, which corresponds to the fact that

the thread currently running inside the monitor is on the verge of leaving it. That invocation is translated into a sequence of assignment statements in which the integer variable `nieq` and the boolean array `guard[]` are set to the corresponding actual parameters specified in the invocation. Since before leaving the monitor the thread has to notify the other threads possibly blocked inside the monitor, the assignment statement sequence is followed by the invocation of the Java method `notifyAll()`. This method wakes up all the threads waiting inside the monitor but the unblocking conditions – which have just been updated by setting the boolean array `guard[]` – will be handled inside the synchronized methods translating the interacting choice equation.

6.3.4 Translating the Interacting Choice Equation

Any group of branches of the interacting choice equation that start with the same interaction is translated into a public synchronized method of the monitor class. The resulting methods basically translate the communication of the passive-control interactions of the monitor type with the active-control interactions of native-thread types to which the passive-control ones are attached.

At the beginning of each such method, a private method called `checkGuard()` is invoked, which is defined in the Java monitor class as follows:

```
private void checkGuard(int guardIndex, boolean blocking)
               throws InterruptedException,
                      SemisyncInteractionException {
  if (blocking)
    while (!guard[guardIndex])
      wait();
  else
    if (!guard[guardIndex])
      throw new SemisyncInteractionException();
}
```

The index of the guard of a public synchronized method associated with an interaction is passed to `checkGuard()` together with a boolean value, `blocking`, that indicates if a thread intends to communicate synchronously (when true) or semi-synchronously (when false) with the monitor.

In the synchronous case, if the `guard[index]` holds true, a thread can enter the monitor without blocking. Otherwise it blocks on the Java method

wait() until another thread leaves the monitor by setting the same guard to true and notifying about this event. Note also that the standard Java InterruptedException, which must be handled whenever the method wait() is used, is delegated to a method up the call stack, i.e. a method of the monitor wrapper class.

In the semi-synchronous case, instead, when the guard[index] evaluates to false an exception called SemisyncInteractionException [2] is raised by the method checkGuard(). The reason is that in this case, when the condition synchronization is false, an entering thread has to exit the monitor without blocking.

After the invocation of the method checkGuard(), within the method associated with an interaction we have a chain of if-else if statements, which handle the selection among the branches (starting with the considered interaction) based on the value of nieq. For those branches sharing the same value of nieq a nested selection statement is necessary, which is based on inherited conditions.

It is worth to point out that a more advanced synchronization mechanism, instead of the method checkGuard(), has been designed for observing the condition synchronization in a blocking or in a non-blocking mode and subsequently – when the observation succeeds – for reserving the access at the monitor only to the same client (i.e. a specific port of an external thread) that has made the observation. This mechanism is useful at the layer RunnableElem, in particular it is exploited by the monitor wrapper class, when a monitor interaction is attached with and/or ports belonging to external threads, in order to allow the polling techniques exposed in Sect. 4.4. However, this requires the generation of a further class for storing the status of the monitor after any update (i.e. a change of the array guard[]) and for providing additional methods of observation, of reservation cancellation, and of actual access to the interactions. Moreover, additional public methods have to be defined into the core monitor class itself, in order to make

[2] Do not confuse the exception SemisyncInteractionException – whose class is synthesized when at least one RunnableElem is realized as a monitor – with the exception NotReadyPortException provided by the package Sync. However, we shall see that when the former propagates to a method send() or receive() of a port of the monitor wrapper class, it will be re-raised as the latter exception.

the observation mechanism available to the monitor wrapper class. Due to the technical details of such a technique, here we have reported only its simplified version.

6.3.5 Synthesizing the Starting Method

The public method `startMonitor()` is in charge of the instantiation of the stub class IAS for the internal actions and of the invocation of the method corresponding to the – internal or setting – equation of the monitor normal form associated with the first equation of the original process algebraic description.

6.4 Generating the Monitor Wrapper Class

The core monitor class is accompanied by a further class that wraps it in order to adapt it to the instances of `RunnableElem` declared in the `RunnableArchi` class. This wrapper class, whose structure is shown in Table 6.2, implements a `RunnableElem` interface and declares/defines a set of members that permit the communication between the monitor and the threads attached to it using overloaded versions of method `attach()` of class `ArchiMeth` of package `Sync`.

```
class ◁ architectural element type name ▷ implements RunnableElem {
  ◁Declaring Monitor▷
  ◁Instantiating Interactions▷
  ◁Defining Constructor▷
  ◁Running Element▷
}
```

Table 6.2. Structure of a monitor wrapper class

The first section of the wrapper class, *Declaring Monitor*, simply contains a declaration of an object of the core monitor class.

The section *Instantiating Interactions* is the most important one. A number of monitor port objects, whose classes and interfaces are defined in the package `Sync` as deriving from the thread ports, is declared and instantiated as there are

the different interactions occurring in the original PADL specification of the AET. More precisely, to each monitor port object reference an instance of a different new anonymous class is assigned, which derives from the appropriate monitor port base class provided by the package `Sync`.

All the monitor port base classes provide a public method called `setBlocking()` that sets a protected boolean variable `isBlocking`. The value of `isBlocking` indicates if an attached thread wants to interact synchronously (if true) or semi-synchronously (if false). Any anonymous class overrides the method `send()` or `receive()` of its port base class. Such an overridden method contains an invocation of a public synchronized method of the core monitor object, which is the method associated to the wanted interaction. The synthesis of the advanced observation/reservation mechanism discussed in Sect. 6.3.4 also allows the appropriate overriding of the methods `obsSnd()` and `obsRcv()`, which are exploited by external and/or thread ports for applying the polling techniques described in Sect. 4.4.

As anticipated in Sect. 4.6, when a monitor port and a thread port are attached through the method `ArchiMeth.attach()` no connectors are instantiated. In this case, in fact, the reference of the monitor port is passed to the thread port, instead of a reference of a connector. The method `ArchiMeth.attach()` also sets the value of the monitor port member `isBlocking`, accordingly to the synchronicity degree of the thread port. In this way, the thread port can directly invoke the method `send()` or `receive()` of the monitor port and, through it, the public synchronized method of the core monitor associated with the interaction. For this reason, conversely to thread ports, monitor ports implements the method `send()` for input interactions, and the method `receive()` for output interactions. Actually, monitor ports are driven by external threads of control.

In the third section, *Defining Constructor*, the monitor wrapper class constructor is defined with the same formal parameters as declared in the constructor of the core monitor class. The constructor of the wrapper class simply instantiates a monitor object by forwarding to it the formal parameters.

The last section, *Running Element*, defines the methods `start()`, `join()`, and `run()`. These methods are implemented in order to initialize the core monitor class and to make the wrapper fully compliant with the code generated

for the RunnableArchi-implementing class, which invoke these methods on all RunnableElem objects. In wrapper classes, the method start() simply invokes the method run() of the class itself. The method join() does nothing, while the method run() invokes the method startMonitor() of the core monitor object. Note that the thread of control that invokes the method run() – which is the thread of the owner RunnableArchi-implementing class – will be released immediately after executing in the core monitor class a method that translates a setting equation, possibly preceded by methods translating internal equations. Therefore, the method run() can be considered as a simple initialization method for the core monitor class.

6.5 Audio Processing System: Phase 3

In this section, the synthesis of a monitor class and of its wrapper is illustrated for the AET Console of the audio processing system introduced in Sect. 3.5. Thus a comparison of the code generated for a monitor with the code generated for a thread, illustrated in Sect. 5.8, will be allowed.

According to Table 6.1, the code generated for the core monitor class Console_Monitor, is composed of several sections. In the first section, internal_Console is declared as a reference to the stub class IAS_Console.

In the second section, instead, the integer variable nieq and the boolean array guard[] are declared, together with the definition of the integer constants associated with the setting equations, which represent the values that nieq can take on and the values that an index for guard[] can take on. Then, the integer variable allowed_changes is declared that translate the homonymous formal parameter of the Console together with the translation of the parameters config_changes and effect_descriptor of the interacting choice equation. Finally, the constructor assigns the value of its parameter allowed_changes to the homonymous protected class member:

```
class Console_Monitor {

  //------------- DECLARING STUBS -------------//
  IAS_Console internal_Console;
```

```
//---------- DEFINING CONSTRUCTOR -----------//
private int nieq;
private boolean[] guard;
private final static int _Start                    = 0,
                         _Split_1_Start             = 1,
                         _Config_Handling           = 2,
                         _Split_1_Config_Handling   = 3,
                         _Split_2_Config_Handling   = 4,
                         _Split_1_Descriptor_Handling = 5;
private final static int _receive_start            = 0,
                         _receive_config           = 1,
                         _receive_descriptor_request = 2,
                         _receive_stop             = 3,
                         _forward_start            = 4,
                         _send_descriptor          = 5,
                         _forward_stop             = 6;
protected int allowed_changes;
protected int config_changes;
protected Interface_Descriptor effect_descriptor;

Console_Monitor(int allowed_changes) {
  this.allowed_changes = allowed_changes;
}
```

The third section of the Console_Monitor is composed of the definition of the method checkGuard(), illustrated in Sect. 6.3:

```
//------------ DEFINING BEHAVIOR ------------//
private void checkGuard(int guardIndex, boolean blocking)
              throws InterruptedException,
                     SemisyncInteractionException {
  if (blocking)
    while (!guard[guardIndex])
      wait();
  else
    if (!guard[guardIndex])
      throw new SemisyncInteractionException();
}
```

which is followed by the translation of the setting and of the local equations, and by the translation of the equations deriving from the interacting choice equation.

The translation of the setting equation Start contains an assignment of the integer parameter config_changes to the homonymous class member, then the integer variable nieq is set with the integer constant _Start associated to the equation itself, and the array guard[] is set with the boolean values for the enabling of the seven interactions of the console. The last statement is an

invocation of the method `notifyAll()` for notifying the other threads possibly blocked inside the monitor:

```
protected synchronized void Start(int config_changes) {
  this.config_changes = config_changes;
  nieq = _Start;
  guard = new boolean[] {true, false, false, false, false, false, false};
  notifyAll();
}
```

Similar to `Start` is the translation of the other setting equations:

```
protected synchronized void Split_1_Start(int config_changes) {
  this.config_changes = config_changes;
  nieq = _Split_1_Start;
  guard = new boolean[] {false, false, false, false, true, false, false};
  notifyAll();
}

protected synchronized void Config_Handling(int config_changes) {
  this.config_changes = config_changes;
  nieq = _Config_Handling;
  guard = new boolean[] {false,
                         (config_changes < allowed_changes),
                         ((config_changes > 0) || (config_changes == 0)),
                         true, false, false, false};
  notifyAll();
}

protected synchronized void Split_1_Config_Handling() {
  nieq = _Split_1_Config_Handling;
  guard = new boolean[] {false, false, false, false, false, true, false};
  notifyAll();
}

protected synchronized void Split_2_Config_Handling(int config_changes) {
  this.config_changes = config_changes;
  nieq = _Split_2_Config_Handling;
  guard = new boolean[] {false, false, false, false, false, false, true};
  notifyAll();
}

protected synchronized void
          Split_1_Descriptor_Handling(Interface_Descriptor effect_descriptor) {
  this.effect_descriptor = effect_descriptor;
  nieq = _Split_1_Descriptor_Handling;
  guard = new boolean[] {false, false, false, false, false, true, false};
  notifyAll();
}
```

The translation of the only internal equation of the monitor normal form of the console, `Descriptor_Handling`, contains an invocation of the stub method `get_summary_descriptor()` in the IAS class instance `internal_Console`:

```
protected synchronized void Descriptor_Handling() {
  Interface_Descriptor effect_descriptor;
  Object[] inputPars;
  inputPars = internal_Console.get_summary_descriptor();
  effect_descriptor = (Interface_Descriptor)inputPars[0];
  Split_1_Descriptor_Handling(effect_descriptor);
}
```

The translation of the first branch of the interacting choice equation is a public method with the same name as the interaction, `receive_start()`, and with the boolean parameter `blocking` which indicates if the interacting thread communicates in synchronous (when true) or in semi-synchronous (when false) mode. The first statement of the method is an invocation of `CheckGuard()`, where value 0 is the index associated with `receive_descriptor_request` within the array `guard[]`. Then, the method simply invokes the setting equation `Split_1_Start()`:

```
public synchronized void receive_start(boolean blocking)
                           throws InterruptedException,
                                  SemisyncInteractionException {
  checkGuard(_receive_start, blocking);
  Split_1_Start(this.config_changes);
```

The other branches of the interacting choice equation are translated in a similar way. Note that the method `receive_config` also contains an invocation of the method `store_config()` on the object `internal_Console`, which translates the corresponding internal action. Note also that the methods `receive_descriptor_request()` and `send_descriptor()` contain `if-else if` chains in order to handle the selection among the next statements to be executed, which translates the corresponding original branches of the interacting choice equation, including their conditions:

```
public synchronized void receive_config(boolean blocking,
                                        Interface_Config console_config)
                           throws InterruptedException,
                                  SemisyncInteractionException {
  checkGuard(_receive_config, blocking);
```

```
  internal_Console.store_config(console_config);
  Config_Handling(this.config_changes + 1);
}

public synchronized void receive_descriptor_request(boolean blocking)
                          throws InterruptedException,
                                 SemisyncInteractionException {
  checkGuard(_receive_descriptor_request, blocking);
  if (this.config_changes > 0)
    Descriptor_Handling();
  else if (this.config_changes == 0)
    Split_1_Config_Handling();
}

public synchronized void receive_stop(boolean blocking)
                          throws InterruptedException,
                                 SemisyncInteractionException {
  checkGuard(_receive_stop, blocking);
  Split_2_Config_Handling(this.config_changes);
}

public synchronized void forward_start(boolean blocking)
                          throws InterruptedException,
                                 SemisyncInteractionException {
  checkGuard(_forward_start, blocking);
  Config_Handling(this.config_changes);
}

public synchronized Object[] send_descriptor(boolean blocking)
                              throws InterruptedException,
                                     SemisyncInteractionException {
  Interface_Descriptor effect_descriptor;
  checkGuard(_send_descriptor, blocking);
  if (nieq == _Split_1_Config_Handling) {
      effect_descriptor = null;
      Config_Handling(0);
  }
  else if (nieq == _Split_1_Descriptor_Handling) {
      effect_descriptor = this.effect_descriptor;
      Config_Handling(0);
  }
  return new Object[] {effect_descriptor.clone()};
}

public synchronized void forward_stop(boolean blocking)
                          throws InterruptedException,
                                 SemisyncInteractionException {
  checkGuard(_forward_stop, blocking);
  Start(this.config_changes);
}
```

In the fourth section of the core monitor class `Console_Monitor`, the public method `startMonitor()` is defined that instantiate the object `internal_Console` of the class `IAS_Console`. Then the method `Start()` is invoked, which corresponds to the first equation of the original PADL specification, with the same initialization parameter:

```
  //------------ STARTING MONITOR -------------//
  public void startMonitor() {
    internal_Console = new IAS_Console();
    Start(0);
  }

}
```

Recalled that the PADL specification of the console of the audio processing system contains the local actions `store_config` and `get_summary_descriptor`, the synthesized Java class `IAS_Console` is exactly the same as illustrated in Sect. 5.8:

```
class IAS_Console {

  IAS_Console() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void store_config(Interface_Configuration console_config) {
    // FILL IN THE METHOD BODY
  }

  Object[] get_summary_descriptor() {
    Interface_Descriptor effect_descriptor = null;
    // FILL IN THE METHOD BODY
    return new Object[] {effect_descriptor};
  }

}
```

As far as the monitor wrapper class `Console` is concerned, it is composed of four sections according to Table 6.2. In particular, the `RunableElem`-implementing class, contains a first section in which the object `core_monitor_Console` of class `Console_Monitor` is declared:

```
class Console implements RunnableElem {
```

```
//------------ DECLARING MONITOR ------------//
Console_Monitor core_monitor_Console;
```

Then, in the second section, a set of monitor ports is declared, each of which handles a different interaction. The input port `receive_start`, associated to the homonymous interaction, is instantiated as a new class of type `UniSyncReceiverMonitorPort`, where the method `send()` is overridden in order to invoke the corresponding method on the object `core_monitor_Console`:

```
//-------- INSTANTIATING INTERACTIONS ---------//
public UniSyncReceiverMonitorPort receive_start =
        new UniSyncReceiverMonitorPort(this) {
  public synchronized void send()
                          throws InterruptedException,
                                 NotReadyPortException {
    try {
      core_monitor_Console.receive_start(isBlocking);
    } catch(SemisyncInteractionException e) {
      throw new NotReadyPortException(e);
    }
  }
};
```

The other ports are declared in the same section. Note that the ports `forward_start`, `send_descriptor`, and `forward_ports` are output ports of class `UniSyncSenderMonitorPort`, an then the method `receive()` is overridden into their anonymous classes:

```
public UniSyncReceiverMonitorPort receive_config =
        new UniSyncReceiverMonitorPort(this) {
  public synchronized void send(Object[] inputPars)
                          throws InterruptedException,
                                 NotReadyPortException {
    try {
      core_monitor_Console.receive_config(isBlocking,
                                (Interface_Config)inputPars[0]);
    } catch(SemisyncInteractionException e) {
      throw new NotReadyPortException(e);
    }
  }
};

public UniSyncReceiverMonitorPort receive_descriptor_request =
        new UniSyncReceiverMonitorPort(this) {
  public synchronized void send()
                          throws InterruptedException,
```

```
                                    NotReadyPortException {
    try {
      core_monitor_Console.receive_descriptor_request(isBlocking);
    } catch(SemisyncInteractionException e) {
      throw new NotReadyPortException(e);
    }
  }
};

public UniSyncReceiverMonitorPort receive_stop =
         new UniSyncReceiverMonitorPort(this) {
  public synchronized void send()
                            throws InterruptedException,
                                   NotReadyPortException {
    try {
      core_monitor_Console.receive_stop(isBlocking);
    } catch(SemisyncInteractionException e) {
      throw new NotReadyPortException(e);
    }
  }
};

public UniSyncSenderMonitorPort = forward_start
         new UniSyncSenderMonitorPort(this) {
  public synchronized Object[] receive()
                                 throws InterruptedException,
                                        NotReadyPortException {
    try {
      core_monitor_Console.forward_start(isBlocking);
    } catch(SemisyncInteractionException e) {
      throw new NotReadyPortException(e);
     }
    return null;
  }
};

public UniSyncSenderMonitorPort = send_descriptor
         new UniSyncSenderMonitorPort(this) {
  public synchronized Object[] receive()
                                 throws InterruptedException,
                                        NotReadyPortException {
    Object[] outputPars;
    try {
      outputPars = core_monitor_Console.send_descriptor(isBlocking);
    } catch(SemisyncInteractionException e) {
      throw new NotReadyPortException(e);
    }
    return outputPars;
  }
};
```

```
public UniSyncSenderMonitorPort = forward_stop
        new UniSyncSenderMonitorPort(this) {
  public synchronized Object[] receive()
                              throws InterruptedException,
                                    NotReadyPortException {
    try {
      core_monitor_Console.forward_stop(isBlocking);
    } catch(SemisyncInteractionException e) {
      throw new NotReadyPortException(e);
    }
    return null;
  }
};
```

In the third section the monitor wrapper class constructor `Console()` is defined with the integer parameter `allowed_changes`. The constructor instantiates the object `core_monitor_Console` of class `Console_Monitor` by forwarding the parameter to it:

```
//----------- DEFINING CONSTRUCTOR ------------//
Console(int allowed_changes) {
  core_monitor_Console = new Console_Monitor(allowed_changes);
}
```

Finally, in the fourth section, the methods `start()`, `join()`, and `run()` are defined. The method `start()` simply invokes the method `run()` of the class itself, the method `join()` does nothing, and the method `run()` invokes the method `startMonitor()` of the monitor object `core_monitor_Console`:

```
  //--------- RUNNING ELEMENT [monitor] ---------//
  public void start() {
    run();
  }

  public void join() throws InterruptedException { }

  public void run() {
    core_monitor_Console.startMonitor();
  }
}
```

Chapter 7

The Translator PADL2Java

The approach illustrated in Chap. 4, 5, and 6 has been implemented in a translator called PADL2Java, which is a command-line application written in Java – the same language as its target. The implementation of PADL2Java and the approaches adopted for generating code are discussed in Sect 7.1.

As far as the application of PADL2Java is concerned, the structure of the code it produces and the related file dependencies are illustrated in Sect. 7.2. In Sect. 7.3 we discuss some issues concerned with the generated code and with the translation of data types provided by PADL. Special emphasis will be placed on the generic object data type, which has recently been introduced in PADL for abstracting and modeling objects and complex data structures in order to improve the effectiveness of the code generated by the translator. Then, in Sect. 7.4 we present the options offered by PADL2Java, and we exemplify them in Sect. 7.5 through the audio processing system of Sect. 3.5. Finally, the architectural description language Æmilia [14], a performance-oriented variant of PADL, is briefly introduced in Sect. 7.6 before illustrating, in Sect. 7.7, the integration of PADL2Java in the architecture-centric verification tool TwoTowers [9].

7.1 Implementation of PADL2Java

The translator PADL2Java is composed of a set of Java classes created by the parser generator JavaCC [46] from the grammar specification of PADL. Other classes, based on the pattern "Visitor" [33], have been developed for analyzing

and transforming the internal representation of a parsed PADL description and for generating code from it.

In terms of the model transformation approaches, as exposed in Sect. 2.4, the translator PADL2Java follows a model-to-text/visitor-based approach for generating code, and a model-to-model/direct manipulation approach for obtaining the endogenous transformation that rewrites an AET into its relative monitor normal form.

In order to facilitate the development of the visitor classes, a further template-based transformation has been adopted during the implementation of PADL2Java. Since JavaCC generates a visitor interface in which an overloaded method `visit()` is declared for each different node-type of the internal representation, a very simple tool called "visitorExpander" has been developed to be used in conjunction with JavaCC. This tool is in charge of generating concrete visitor classes from the visitor interface and from a template where a single generic method `visit()` is defined. Each of these classes can then be inherited by a more advanced visitor class in which only a subset of the `visit()` methods needs to be overridden, depending on the specific task to be executed.

In particular, two concrete base classes have been generated with the tool visitorExpander. The first one, which simply explores the internal representation, has been employed as a base class for developing a family of analyzer- and code generator-visitors. The second one, which during its visit duplicates each node of the internal representation, has been used instead as a base class for developing a family of manipulator-visitors.

The rules adopted by the two families of visitor for the code generation and for the internal structure manipulation are those described in Chap. 4, 5, 6, and in next Sect. 7.3 and 7.4.

7.2 Structure of the Generated Code

As shown in the upper part of Fig. 7.1, PADL2Java synthesizes a class implementing the interface `RunnableArchi` plus as many classes implementing the interface `RunnableElem` as there are AETs in the PADL description. It is worth reminding that, in order to provide a Java abstraction for representing

hierarchical/compound architectural types, the interface `RunnableArchi` has been defined by extending the interface `RunnableElem`, which extends in turn the standard interface `Runnable`.

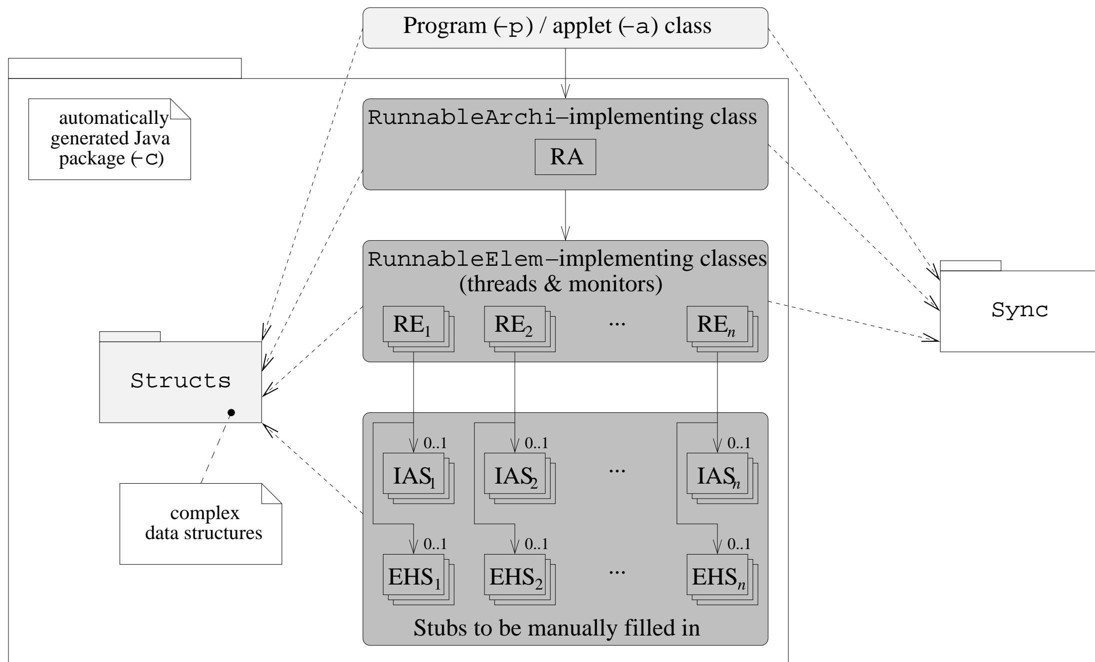

Figure 7.1. Generated Java classes/packages and dependencies from package `Sync`

The instances generated from the classes implementing `RunnableElem`, i.e. threads and monitors, are guaranteed to interact as expected thanks to the generation of suitable `Port` and `Connector` objects from package `Sync`. While `Port` is a family of public interfaces and classes used by the code generated by PADL2Java, `Connector` is a family of private pieces of software that are accessible only within package `Sync`. `Connector` classes are transparently instantiated whenever two thread `Port` objects are attached to each other.

The lower part of Fig. 7.1 shows the stub classes to be manually filled in. These are generated for managing the internal actions of the AETs implemented as thread and monitor classes (IAS) and for handling the exceptions that may be raised by the interactions implemented within the thread classes (EHS) – i.e. `UnattachedPortException` and `NotReadyPortException`.

7.3 Translating Data Types

Most of the data types provided by PADL – such as boolean, (bounded) integer, real, list, and array – can be trivially translated into built-in Java data types. By contrast, the record data type and the generic object data type provided by PADL are managed by PADL2Java by generating ad-hoc classes and interfaces through package `Structs` depicted in Fig. 7.1.

Each record type occurring in a PADL description is simply realized as a Java class. Each attribute of this class synthesizes a different record field according to the data type of the field.

Concerning the generic object data type, this has recently been introduced in PADL in order to enhance the expressiveness of the language and the flexibility of the translator. In fact, software systems may require the exchange of complex data among threads/monitors. The generic object mediation allows the developer to define easily new data types, which can then be handled by the translator.

From the architectural viewpoint, a datum of type `object(`◁*type id*▷`)` is an entity of a not completely specified data type identified by ◁*type id*▷, with no specific semantics nor specific operations associated with it. Generic objects can only be declared and checked for equality to `null` in PADL descriptions. Thanks to the parametricity of a generic object, type checking can be applied in order to detect possible mismatches among objects of different generic types.

From the code generation viewpoint, different generic object types are translated into different interfaces within `Structs` that provide the abstractions with which new Java classes can be implemented by the developer. Each new class realizes – or wraps an instance of – a different Java user-defined data type. Thanks to the interfaces, instances of these classes can be properly exchanged among threads/monitors through `Connector` and `Port` objects, and can be exchanged with the stubs that translate the internal actions. In order to manage the case of instance wrapping, each generated interface declares a method called `getInstance()`, which the developer will have to define in such a way that it returns the desired instance.

Since generic objects to be transferred through `Connector`/`Port` objects must enable the production of deep copies of themselves, all the interfaces generated

within package `Structs` extend the standard interface `Cloneable` and declare the copy constructor method `clone()` as public. By doing so, the developer is forced to define such a method for each different user-defined data type by taking care of the correctness of the copy. This complies with one of the guidelines of Sect. 5.7 for guaranteeing the preservation of the properties proved at the architectural design level. In fact, the thread communication model adopted in Sect. 4.1 does not admit passing object references while keeping a copy of the references.

7.4 Translation Options

Each of the classes generated by PADL2Java for a given PADL description is stored into a distinct `.java` file. As shown in Fig. 7.1, all the resulting files are contained in a single package, which by default has the same name as the PADL description source file. Further `.java` files may be generated depending on the translation option specified upon invoking PADL2Java.

The available options are `-c`, `-p`, and `-a`. Option `-c` is the default one. When using this option, no further class is generated outside the package.

If option `-p` is used, a full Java program is synthesized. This is achieved by PADL2Java through the generation of a further public class that contains only method `main()`. This acts as a wrapper for the `RunnableArchi`-implementing class and contains two sections. In the first one an instance of that class is created, while in the second one that instance is started.

Finally, if option `-a` is used, a Java applet is synthesized. This is achieved by PADL2Java through the generation of a further public class that is derived from the standard `JApplet` class and contains three sections that give rise to a wrapper for the `RunnableArchi`-implementing class.

In the first section, an object of the `RunnableArchi`-implementing class is declared. Unlike the program wrapper, here the previously mentioned object is not instantiated, as this will be done by the developer within a suitable method defined in the third section.

In the second section, a `Port` object is declared for each of the architectural interactions declared in the `RunnableArchi`-implementing class. These can be attached to the architectural interactions and then used by the developer

within suitable methods defined in the third section. The idea behind such additional interactions is to allow the applet wrapper to dispatch suitable commands to the right `RunnableElem` objects depending on the events that are caught. Each of the additional interactions is semi-synchronous – hence it cannot block the applet wrapper – and uni – hence it can surely be attached to the corresponding architectural interaction if necessary. In order to help the developer to correctly attach the additional interactions to the corresponding architectural interactions, each additional interaction has the same name as the corresponding architectural interaction preceded by the prefix `to_` (resp. `from_`) if the corresponding architectural interaction is an input (resp. output) interaction.

In the third section, the stubs for the typical methods of applet classes are added, with some comments to remind the developer to define them if needed. Such methods are related to the initialization, activation, deactivation, and destruction of an applet.

7.5 Audio Processing System: Completing Code Generation

We now exemplify the effect of the three translation options by means of the audio processing system introduced in Sect. 3.5. We suppose that the PADL description of the audio processing system is stored in a file called `audio.padl`.

The following command:

```
java PADL2Java audio.padl -c
```

generates a package, i.e. a directory, having the same name as the `.padl` source file, which contains all of the classes – some of which have been shown in Sect. 4.7, 5.8, and 6 – that have to be synthesized for the whole architectural type and for each of its AETs, including the stubs classes and the internal package `Structs`. Each class is stored in a `.java` file having the same name as the class.

The following command:

```
java PADL2Java audio.padl -p <program name>.java
```

generates the same package as before plus an external `.java` file. This contains

the following code importing the `RunnableArchi`-implementing class defined in the package:

```
import audio.Audio_Processing_System;

public class <program name> {
  public static void main(String args[]) {

    //------- INSTANTIATING ARCHITECTURE -------//
    Audio_Processing_System archiInstance = new Audio_Processing_System();

    //---------- RUNNING ARCHITECTURE ----------//
    archiInstance.start();
    try {
      archiInstance.join();
    } catch(InterruptedException e) {}

  }
}
```

The following command:

```
java PADL2Java audio.padl -a <applet name>.java
```

generates the same package as before plus an external `.java` file. This contains the following code, where also package `Sync` is imported in order to make available the `Port` classes implementing semi-synchronous uni-interactions:

```
import Sync.*;
import audio.Audio_Processing_System;
import javax.swing.JApplet;

public class <applet name> extends JApplet {

  //---------- DECLARING ARCHITECTURE ----------//
  Audio_Processing_System archiInstance;

  //---------- DECLARING APPLET PORTS ----------//
  UniSemisyncSenderPort to_receive_start;
  UniSemisyncSenderPort to_receive_config;
  UniSemisyncSenderPort to_receive_stop;
  UniSemisyncReceiverPort from_open_input_device;
  UniSemisyncSenderPort to_read_dry_samples;
  UniSemisyncReceiverPort from_close_input_device;
  UniSemisyncReceiverPort from_open_output_device;
  UniSemisyncReceiverPort from_write_processed_sample;
  UniSemisyncReceiverPort from_close_output_device;
```

```
  //--------- DEFINING APPLET MEMBERS ---------//
  public void init() {
   // FILL IN THE METHOD BODY IF NEEDED
  }
  public void start() {
   // FILL IN THE METHOD BODY IF NEEDED
  }
  public void stop() {
   // FILL IN THE METHOD BODY IF NEEDED
  }
  public void destroy() {
   // FILL IN THE METHOD BODY IF NEEDED
  }

}
```

7.6 Æmilia: A Performance-Oriented Variant of PADL

Before illustrating the integration of PADL2Java in TwoTowers, it is worth recalling Æmilia, a performance-oriented variant of PADL. Every action in Æmilia is composed not only of the name of the action, but also of the duration of the action. Normally the duration of an action is exponentially distributed, so that with each Æmilia description it is possible to associate a performance model in the form of a continuous-time Markov chain. In many cases it is also possible to associate a more component-oriented model like a queueing network [6]. A companion language called MSL [2] is available to allow the designer to express in a component-oriented fashion also the performance measures of interest for an Æmilia specification.

7.7 Integration of PADL2Java in TwoTowers

In order to implement an architecture-centric approach going from software specification to software implementation in a way that supports property prediction and preservation, PADL2Java has been integrated in TwoTowers. This is an open-source software tool for the functional verification, security analysis, and performance evaluation of systems modeled in Æmilia.

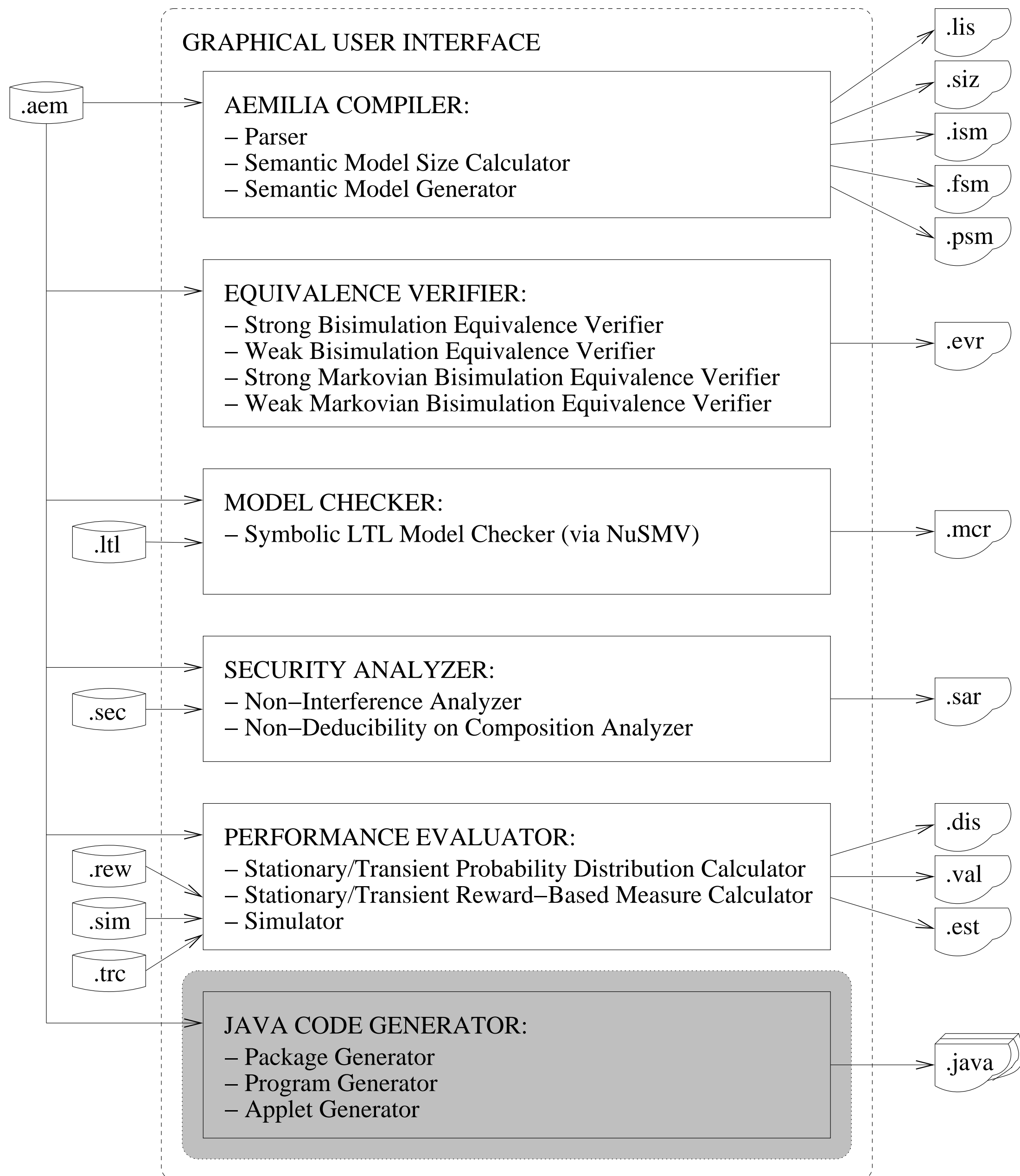

Figure 7.2. Architecture of TwoTowers 6.0

As shown in Fig. 7.2, TwoTowers is equipped with a simple graphical user interface through which the user can invoke several routines by means of suitable menus. The graphical user interface takes care of the integrated management of the various file types needed by the different routines. These belong to the Æmilia compiler, the equivalence verifier, the model checker, the security analyzer, the performance evaluator, and the novel Java code generator.

The compiler is in charge of parsing Æmilia descriptions stored in `.aem` files and signaling possible lexical, syntax and static semantic errors through a `.lis` file. If an Æmilia description is correct the compiler can generate its integrated, functional or performance semantic model, which is written to a `.ism`, `.fsm` or `.psm` file, respectively. As a faster option that does not require printing the state space onto a file, the compiler can show only the size – in terms of number of states and transitions – of the semantic model, which is written to a `.siz` file.

The equivalence verifier [23] checks whether two correct, finite-state Æmilia descriptions are equivalent according to one of four different behavioral equivalences: strong bisimulation equivalence, weak bisimulation equivalence, strong Markovian bisimulation equivalence, and weak Markovian bisimulation equivalence. The result of the verification is written to a `.evr` file. In the case of non-equivalence a distinguishing modal logic formula is reported as well, which is expressed in a verbose variant of the Hennessy-Milner logic or one of its probabilistic extensions.

The model checker [22] verifies through the BDD-based routines of NuSMV [21] whether a set of functional properties expressed through verbose LTL formulas, which are stored in a `.ltl` file, are satisfied by a correct, finite-state Æmilia description. The result of the check, together with a counterexample for each property that is not met, is written to a `.mcr` file.

The security analyzer checks through the equivalence verifier whether a correct, finite-state Æmilia description satisfies non-interference or non-deducibility on composition [31], both of which establish the absence of illegal information flows from high security system components to low security system components. This requires the classification in an additional `.sec` file of the system activities that are high and low with respect to the security level. The result of the analysis

is written to a `.sar` file, together with a modal logic formula expressed in a verbose variant of the Hennessy-Milner logic to explain a possible security violation.

The performance evaluator assesses the quantitative characteristics of correct, finite-state and performance closed Æmilia descriptions. First, it can calculate the stationary/transient probability distribution [63] for the state space of the performance semantic model of an Æmilia description, which is written to a `.dis` file. Second, the performance evaluator can calculate for an Æmilia description a set of instant-of-time, stationary/transient performance measures specified through state and transition rewards stored in a `.rew` file [41]. The values of the measures are written to a `.val` file. Third, the performance evaluator can estimate via discrete event simulation [66] the mean, variance or distribution of a set of performance measures specified through an extension of state and transition rewards, which are stored in a `.sim` file together with the number and the length of the simulation runs. The simulation can be applied also to Æmilia descriptions with infinitely many states and general distributions and can be trace driven, in which case the traces are stored in `.trc` files. The result of the simulation is written to a `.est` file together with its confidence interval.

Finally, the new Java code generator is the translator PADL2Java that we have described in this chapter. Even if PADL2Java has been initially designed for PADL descriptions, it can also synthesize a Java package out of a correct Æmilia description. In fact, syntactical differences between PADL and Æmilia descriptions are ignored during the parsing process. A package synthesized with PADL2Java can be augmented with a class containing method `main()` or a `JApplet`-derived class, thus resulting in a Java program or a Java applet, respectively.

Chapter 8

Case Studies

In this chapter two case studies are presented. The first one, illustrated in Sect. 8.1, is related to the synthesis of a video animation repainting system. Before generating code, the model checker provided by TwoTowers is used for verifying some properties of the modeled system.

The second case study, illustrated in Sect. 8.2, is related to the implementation of a leader election algorithm proposed in [32]. Some probabilistic analysis will be done both at the process algebraic description level – through the performance evaluator provided by TwoTowers – and at the implementation level – where the generated code will be used for producing execution traces.

8.1 A Video Animation Repainting System

This section provides the PADL description of a video animation repainting system inspired by [52]. The related Java code is then synthesized with PADL2Java. Before the synthesis, the deadlock freedom of the whole system will be verified at the process algebraic description level using the model checker provided by TwoTowers.

8.1.1 Informal Specification of the Video Animation Repainting System

The software generated for the implementation of the video animation repainting system will have to graphically show the behavior and the evolution of a set of

interacting elements called *actors*, each of which has its own rules and states. The state of an actor, which is controlled by an autonomous thread, may be graphically represented with changing colors, positions, and shapes (as e.g. in videogames, screen savers, and dynamic diagrams representations). Another feature that the generated software will have to possess is the ability of taking snapshots of the graphical animations.

There are several approaches to the design of animation systems. The first one consists of modeling such systems as discrete-time systems, in which a common clock coordinates the behavior of the various actors. In such systems a discrete model is adopted in which the passage of time is signaled by successive ticks of a clock, so that the actors (threads) become aware of the passage of time by sharing a global tick action. This model offers a simple mechanism for synchronizing screen updates with actor activities, and allows a consistent collective actor state configuration to be visualized.

However, this approach does not apply to actors working in continuous time whose behavior is required to be visualized in continuous time. In fact, introducing discrete-time constraints may considerably alter the original system properties. The problem with continuous-time systems is to take snapshots of consistent configurations. If one does not care about a perfect consistency, one of the other approaches that we are going to present can be adopted.

The second approach to visualize the system evolution is to repaint the screen every time that an actor changes its state. The repainting is driven by the change event. Note that the consistency of a configuration is not guaranteed if two or more actors change at the same time, as only a single event at a time can be captured. This situation usually occurs when the change is due to a synchronization among the actors. This approach is not efficient in general, because a high configuration change rate – which can considerably grow during synchronizations – produces large context-switching and graphical overheads that degrade the performance of the whole system.

The third approach for visualizing continuous-time systems consists of sampling periodically the configuration and, if changed, to paint it to the screen. This approach improves the performance of the previous one, as the repainting process has a fixed maximum rate. The problem with this solution is that not

all the configurations will be visualized, but in the case of video animations this can be overtaken with a suitable sampling rate (e.g. 25 Hz). The consistency of a configuration is not guaranteed in this case either, in fact the sampling event may occur when only a subset of the simultaneous mutant actors have notified, or are able to notify on request, their change.

In order to define a repainting system for video animation endowed with a snapshot capturer, it is convenient to combine the first approach (discrete-time systems) with the third one (continuous-time systems with fixed sampling rate). The advantage of using these two techniques is that it is possible to capture snapshots of consistent configurations without introducing strong discrete-time constraints on the underlying dynamic system. In fact, while the animation requires an actor only to notify information about its state whenever it changes, the snapshot capturing may be imposed only at significant consistent states, not for all changes. This means that, in order to discretize the model of a given dynamic system, it is sufficient to set up a few synchronization points among the actors (a *director* will be introduced for this purpose), in which the configuration has to be consistent and the snapshots can be taken. Note that to obtain a fully discrete-time system would require, instead, the addition of synchronization points after each action of each actor.

8.1.2 PADL Description of the Video Animation Repainting System

The architecture of the video animation repainting system is illustrated in Fig. 8.1. The lower part of the diagram contains a set of interacting actors (`A[1], ..., A[n_actors]`) that constitute the dynamic system to be visualized on the screen.

Each actor can freely interact with any other one, but the director (`D`), which can be considered as a generalized system clock but also as a first-class actor, can sometimes call the other actors in order to synchronize them. The `State_Repository SR`, represented in the upper part of the diagram, is the core of the repainting system: it can receive and store at any time the information about the current state that each actor communicates through the connector

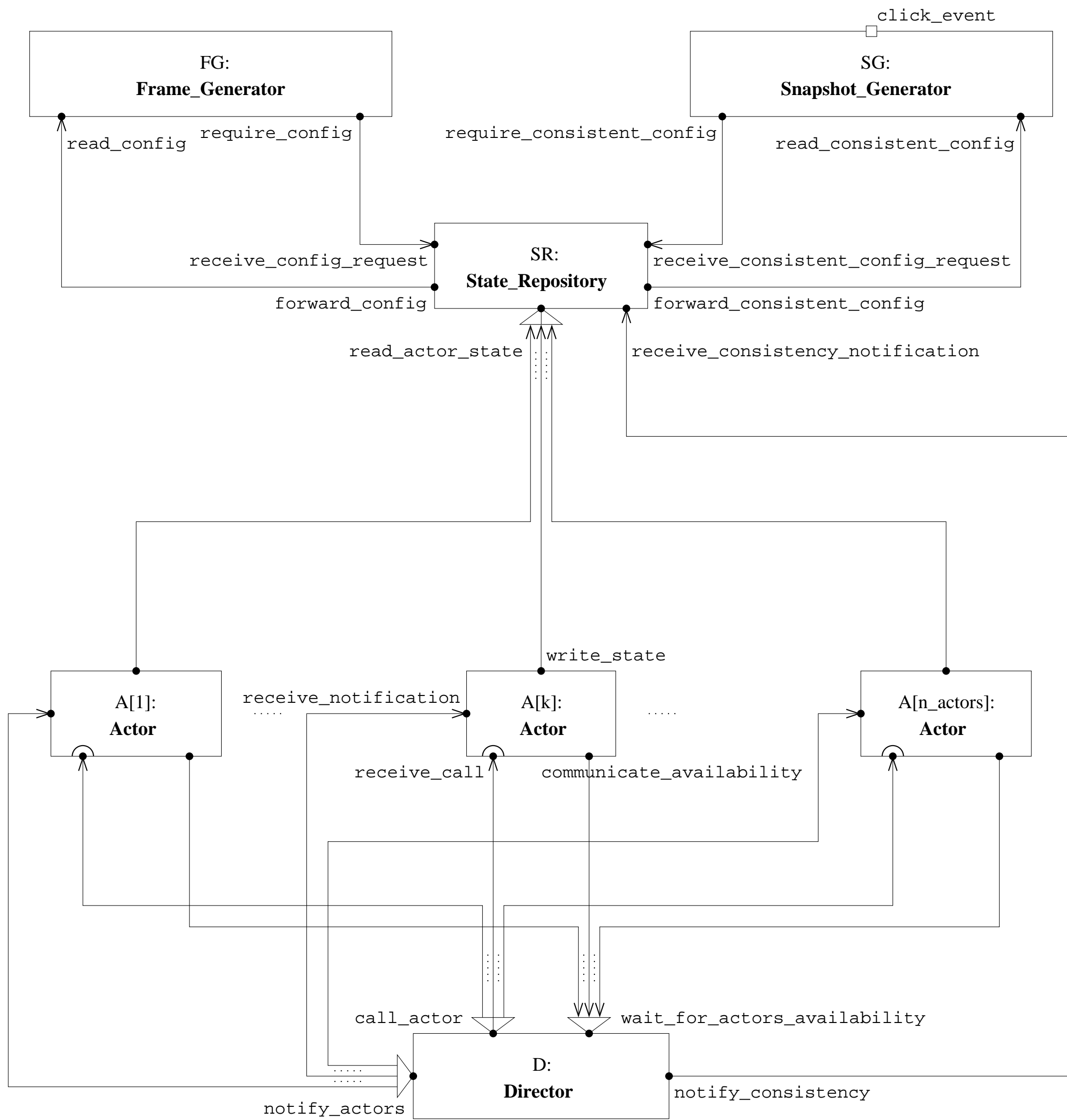

Figure 8.1. Extended flow graph of the video animation repainting system

`read_actor_state`. In general, the global state configuration stored within the state repository is inconsistent.

However, since the consistency of the global state configurations is a concern of the actor community, from time to time the director synchronizes the actors by calling them one by one (`call_actor`) until the last actor has been called, and by waiting for their availability (`wait_for_actors_availability`). When an actor is called, it communicates its state to the state repository, then signals its availability to the director. Only when all the actors are available (`communicate_availability`), the director signals to the state repository that the configuration is consistent (`notify_consistency`). The state repository stores and marks as a "consistent global state configuration" the configuration built from the last state communicated by each actor. After the notification of consistency to the state repository, the director releases the actors (`notify_actors`) all together, so that they can revert to their usual activities.

The two remaining architectural elements involved in the video animation repainting system are a `frame_generator` (`FG`) and a `snapshot_generator` (`SG`). The former (resp. latter) asks the state repository for the last stored global state configuration (resp. the last stored consistent global state configuration) via `config_request` (resp. `consistent_config_request`), then waits for a response via `config` (resp. `consistent_config`). Other differences between the two generators are that, for acquiring the configuration, the frame generator is driven by an internal clock, while the snapshot generator is driven by an external event (`click_event`). Finally, each configuration will be then exploited by these generators for creating a frame to be painted on the screen or a snapshot to be stored somewhere (e.g. the clipboard).

The topology of the system is illustrated in Fig. 8.1. Note that the interaction `receive_call` with which an actor is attached to the synchronous call of the director `call_actor` has been modeled as semi-synchronous. By doing so, an actor must respond to a call only when the director is waiting for, and does not block otherwise. On the other side, the director blocks until the faster actor responds, then a new call for the next actor can be done.

The PADL textual description of the system is provided below, in which `n_actors` and `clock_period` are parameters of the whole specification and

interaction `click_event` of `SG` is the only architectural interaction:

```
ARCHI_TYPE Video_Animation_Repainting_System(const integer n_actors   :=  3,
                                             const integer clock_rate := 25)

ARCHI_BEHAVIOR

  ARCHI_ELEM_TYPE State_Repository(void)

    BEHAVIOR

      Initialization(void;
                     void) =
        store_consistent_config!(null) .
          Receiving(false);

      Receiving(boolean consistent_config_ready;
                local object(ActorState) actor_state,
                local object(Configuration) config) =
        choice
        {
          read_actor_state?(actor_state) .
            store_actor_state!(actor_state) .
              Receiving(consistent_config_ready),
          receive_config_request .
            get_config?(config) .
              forward_config!(config) .
                Receiving(consistent_config_ready),
          receive_consistency_notification .
            mark_config_as_consistent .
              Receiving(true),
          cond (consistent_config_ready) ->
            receive_consistent_config_request .
              get_last_consistent_config?(config) .
                forward_consistent_config!(config) .
                  Receiving(true)
        }

    INPUT_INTERACTIONS  UNI receive_config_request;
                            receive_consistent_config_request;
                            receive_consistency_notification
                        OR  read_actor_state

    OUTPUT_INTERACTIONS UNI forward_config;
                            forward_consistent_config


  ARCHI_ELEM_TYPE Frame_Generator(const integer clock_rate)

    BEHAVIOR

      Initialization(void;
```

```
                 void) =
      start_internal_clock!(clock_rate) .
        Generation();

    Generation(void;
               local object(Configuration) config) =
      wait_for_tick.
        require_config .
          read_config?(config) .
            generate_and_print_frame!(config) .
              Generation()

  INPUT_INTERACTIONS  UNI read_config

  OUTPUT_INTERACTIONS UNI require_config


ARCHI_ELEM_TYPE Snapshot_Generator(void)

  BEHAVIOR

    Snapshot(void;
             local object(Configuration) config) =
      click_event .
        require_consistent_config .
          read_consistent_config?(config) .
            generate_and_print_snapshot!(config) .
              Snapshot()

  INPUT_INTERACTIONS  UNI click_event;
                          read_consistent_config

  OUTPUT_INTERACTIONS UNI require_consistent_config


ARCHI_ELEM_TYPE Actor(const integer actor_id)

  BEHAVIOR

    Initialization(void;
                   void) =

      store_identifier!(actor_id) .
        Free_Acting();

    Free_Acting(void;
                local object(ActorState) actor_state) =
      play?(actor_state) .
        receive_call .
          choice {
                 cond(!receive_call.success) ->
```

```
                  write_state!(actor_state) . Free_Acting(),
                cond(receive_call.success) ->
                  prepare_to_snapshot?(actor_state) .
                    write_state!(actor_state) . Availability()
              };

    Availability(void;
                 void) =
      communicate_availability .
        receive_notification .
          Free_Acting()

  INPUT_INTERACTIONS  UNI SSYNC receive_call;
                          SYNC  receive_notification

  OUTPUT_INTERACTIONS UNI write_state;
                          communicate_availability


ARCHI_ELEM_TYPE Director(const integer n_actors)

  BEHAVIOR

    Initialization(void;
                   void) =
      start .
        Preparation();

    Preparation(void;
                void) =
      do_something .
        Calling_Actors(n_actors);

    Calling_Actors(integer(0..n_actors) n_actors_to_call;
                   void) =
      call_actor .
        choice
        {
          cond(n_actors_to_call > 0) -> call_actor .
            Calling_Actors(n_actors_to_call - 1),
          cond(n_actors_to_call = 0) -> wait_for_actors_availability .
            Notification()
        };

    Notification(void;
                 void) =
      notify_consistency .
        notify_actors .
          Preparation()

  INPUT_INTERACTIONS  AND wait_for_actors_availability
```

```
    OUTPUT_INTERACTIONS UNI notify_consistency
                        AND notify_actors
                        OR  call_actor

ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES

    SR: State_Repository();
    FG: Frame_Generator(clock_rate);
    SG: Snapshot_Generator();
    FOR_ALL i IN 1 .. n_actors
      A[i] : Actor(i);
    D: Director(n_actors)

  ARCHI_INTERACTIONS

    SG.click_event

  ARCHI_ATTACHMENTS

    FROM SR.forward_config              TO FG.read_config;
    FROM SR.forward_consistent_config   TO SG.read_consistent_config;
    FROM FG.require_config              TO SR.receive_config_request;
    FROM SG.require_consistent_config   TO SR.receive_consistent_config_request;
    FOR_ALL i IN 1 .. n_actors
      FROM A[i].write_state             TO SR.read_actor_state;
    FOR_ALL i IN 1 .. n_actors
      FROM A[i].communicate_availability TO D.wait_for_actors_availability;
    FOR_ALL i IN 1 .. n_actors
      FROM D.call_actor                 TO A[i].receive_call;
    FROM D.notify_consistency           TO SR.receive_consistency_notification;
    FOR_ALL i IN 1 .. n_actors
      FROM D.notify_actors              TO A[i].receive_notification

END
```

Note that the iterative mechanism presented in Sect.3.2.1 has been used for declaring several AEIs of type `Actor` concisely, in a way that any AEI can be identified via an index. The same iterative mechanism is exploited for declaring the architectural attachments involving all the instances of `Actor` and other AEIs – i.e., the `State_Repository SR` and the `Director D`.

8.1.3 Verifying Properties of the Video Animation Repainting System

An important property of the PADL specification of the video animation repainting system has been verified through the LTL model checker of TwoTowers. This is the deadlock freedom of the whole system. The specification of this property, written in a `.ltl` file, is as follows:

```
PROPERTY deadlock_freedom IS
  DEADLOCK_FREE;
```

and the outcome of its verification is:

```
Validity of the properties for Video_Animation:

- Property "deadlock_freedom" is satisfied.
```

Once the property above has been verified at the architectural level, we can proceed with the synthesis of the Java code thanks to PADL2Java.

8.1.4 Synthesizing the Video Animation Repainting System

The following code is synthesized using PADL2Java. First of all, the class `Video_Animation_Repainting_System` is generated that translates the architectural topology of the homonymous architectural type:

```
public class Video_Animation_Repainting_System implements RunnableArchi {

  //-------- DECLARING RUNNABLE ELEMENTS --------//
  State_Repository SR;
  Frame_Generator FG;
  Snapshot_Generator SG;
  Sync.util.HashArray<Actor> A;
  Director D;

  //--- DECLARING ARCHITECTURAL INTERACTIONS ----//
  public UniSyncReceiverPort SG_click_event;

  //----------- DEFINING CONSTRUCTOR ------------//
  protected int n_actors;
  protected int clock_rate;
```

```
// GENERAL CONSTRUCTOR:
Video_Animation_Repainting_System(int n_actors,
                                  int clock_rate) {
  this.n_actors = n_actors;
  this.clock_rate = clock_rate;
  buildArchiTopology();
}

// DEFAULT CONSTRUCTOR:
Video_Animation_Repainting_System() {
  this(3,
       25);
}

//----------- BUILDING ARCHITECTURE -----------//
void buildArchiTopology() {

  //  INSTANTIATING RUNNABLE ELEMENTS:
  SR = new State_Repository();
  FG = new Frame_Generator(clock_rate);
  SG = new Snapshot_Generator();
  A = new Sync.util.HashArray<Actor>();
  for (int i = 1; i <= n_actors; i++)
    A.put(i,
          new Actor(i));
  D = new Director(n_actors);

  //  ASSIGNING ARCHITECTURAL INTERACTIONS:
  this.SG_click_event = SG.click_event;

  //  ATTACHING LOCAL INTERACTIONS:
  try {
    ArchiMeth.attach(SR.forward_config, FG.read_config);
    ArchiMeth.attach(SR.forward_consistent_config, SG.read_consistent_config);
    ArchiMeth.attach(FG.require_config, SR.receive_config_request);
    ArchiMeth.attach(SG.require_consistent_config, SR.receive_consistent_config_request);
    for (int i = 1; i <= n_actors; i++)
      ArchiMeth.attach(A.get(i).write_state, SR.read_actor_state);
    for (int i = 1; i <= n_actors; i++)
      ArchiMeth.attach(A.get(i).communicate_availability, D.wait_for_actors_availability);
    for (int i = 1; i <= n_actors; i++)
      ArchiMeth.attach(D.call_actor, A.get(i).receive_call);
    ArchiMeth.attach(D.notify_consistency, SR.receive_consistency_notification);
    for (int i = 1; i <= n_actors; i++)
      ArchiMeth.attach(D.notify_actors, A.get(i).receive_notification);
  } catch(BadAttachmentException e) {}
}

//----------- RUNNING ARCHITECTURE  -----------//
Thread th_Video_Animation = null;
```

```
    public void start() {
      (th_Video_Animation = new Thread(this)).start();
    }

    public void join() throws InterruptedException {
      th_Video_Animation.join();
    }

    public void run() {
      SR.start();
      FG.start();
      SG.start();
      for (int i = 1; i <= n_actors; i++)
        A.get(i).start();
      D.start();
      try {
        SR.join();
        FG.join();
        SG.join();
        for (int i = 1; i <= n_actors; i++)
          A.get(i).join();
        D.join();
      } catch(InterruptedException e) {}
    }
  }
```

Note that the iterative mechanism of PADL for declaring several AEIs of the same type (`Actor`) and for declaring their attachments, on the implementation side is translated into equivalent iterative statements that instantiate objects and attach their ports. The generic Java class `HashArray<>` provided by the package `Sync` is used by the class `Video_Animation_Repainting_System` for storing the instances of `Actor`. The advantage of using such a hash table-based array instead of an usual array is that the former works efficiently in presence of sparse indexes, which can occur in PADL descriptions.

The `State_Repository` of the original PADL description is translated into a monitor because all the monitor constraints are satisfied for this AET. We point out, in fact, that its internal actions are `store_identifier`, `play`, and `prepare_to_snapshot` and the process algebraic description of its behavior has no cycles involving only occurrences of these actions. If all the other AETs are synthesized as threads, it has no monitor type instances attached to its only instance `SR`. All of its interactions are synchronous and no interaction of the AEIs attached to its only instance `SR` is asynchronous. Finally, the process algebraic

description of its behavior has no hybrid choices.

Once the AET `State_Repository` has been rewritten in monitor normal form, the core monitor class `State_Repository_Monitor` is generated as follows:

```
class State_Repository_Monitor {

  //------------- DECLARING STUBS -------------//
  IAS_State_Repository internal_State_Repository;

  //---------- DEFINING CONSTRUCTOR -----------//
  private int nieq;
  private boolean[] guard;
  private final static int _Receiving         = 0,
                           _Split_1_Receiving = 1,
                           _Split_2_Receiving = 2;
  private final static int _receive_config_request            = 0,
                           _receive_consistent_config_request = 1,
                           _receive_consistency_notification  = 2,
                           _read_actor_state                  = 3,
                           _forward_config                    = 4,
                           _forward_consistent_config         = 5;
  protected boolean consistent_config_ready;
  protected Interface_Configuration split_config;

  State_Repository_Monitor() { }

  //------------ DEFINING BEHAVIOR ------------//
  private void checkGuard(int guardIndex, boolean blocking)
               throws InterruptedException,
                      SemisyncInteractionException {
    if (blocking)
      while (!guard[guardIndex])
        wait();
    else
      if (!guard[guardIndex])
        throw new SemisyncInteractionException();
  }

  protected synchronized void Initialization() {
    internal_State_Repository.store_consistent_config(null);
    Receiving(false);
  }

  protected synchronized void Receiving(boolean consistent_config_ready) {
    this.consistent_config_ready = consistent_config_ready;
    nieq = _Receiving;
    guard = new boolean[] {true, true && consistent_config_ready, true, true, false, false};
    notifyAll();
  }
```

```
protected synchronized void Split_1_Receiving(boolean consistent_config_ready,
                                              Interface_Configuration split_config) {
  this.consistent_config_ready = consistent_config_ready;
  this.split_config = split_config;
  nieq = _Split_1_Receiving;
  guard = new boolean[] {false, false, false, false, true, false};
  notifyAll();
}

protected synchronized void Split_2_Receiving(Interface_Configuration split_config) {
  this.split_config = split_config;
  nieq = _Split_2_Receiving;
  guard = new boolean[] {false, false, false, false, false, true};
  notifyAll();
}

public synchronized void receive_config_request(boolean blocking)
                           throws InterruptedException,
                                  SemisyncInteractionException {
  Object[] inputPars;
  checkGuard(_receive_config_request, blocking);
  inputPars = internal_State_Repository.get_config();
  this.split_config = (Interface_Configuration)inputPars[0];
  Split_1_Receiving(this.consistent_config_ready,
                    this.split_config);
}

public synchronized void receive_consistent_config_request(boolean blocking)
                           throws InterruptedException,
                                  SemisyncInteractionException {
  Object[] inputPars;
  checkGuard(_receive_consistent_config_request, blocking);
  inputPars = internal_State_Repository.get_last_consistent_config();
  this.split_config = (Interface_Configuration)inputPars[0];
  Split_2_Receiving(this.split_config);
}

public synchronized void receive_consistency_notification(boolean blocking)
                           throws InterruptedException,
                                  SemisyncInteractionException {
  checkGuard(_receive_consistency_notification, blocking);
  internal_State_Repository.mark_config_as_consistent();
  Receiving(true);
}

public synchronized void read_actor_state(boolean blocking,
                                          Interface_ActorState actor_state)
                           throws InterruptedException,
                                  SemisyncInteractionException {
  checkGuard(_read_actor_state, blocking);
  internal_State_Repository.store_actor_state(actor_state);
```

```
    Receiving(this.consistent_config_ready);
  }

  public synchronized Object[] forward_config(boolean blocking)
                              throws InterruptedException,
                                     SemisyncInteractionException {
    Interface_Configuration split_config;
    checkGuard(_forward_config, blocking);
    split_config = (Interface_Configuration)this.split_config.clone();
    Receiving(this.consistent_config_ready);
    return new Object[] {split_config};
  }

  public synchronized Object[] forward_consistent_config(boolean blocking)
                              throws InterruptedException,
                                     SemisyncInteractionException {
    Interface_Configuration split_config;
    checkGuard(_forward_consistent_config, blocking);
    split_config = (Interface_Configuration)this.split_config.clone();
    Receiving(this.consistent_config_ready);
    return new Object[] {split_config};
  }

  //------------- STARTING MONITOR --------------//
  public void startMonitor() {
    internal_State_Repository = new IAS_State_Repository();
    Initialization();
  }
}
```

The monitor wrapper class `State_Repository` associated to the core monitor class `State_Repository_Monitor` is generated as follows:

```
class State_Repository implements RunnableElem {

  //------------- DECLARING MONITOR -------------//
  State_Repository_Monitor core_monitor_State_Repository;

  //-------- INSTANTIATING INTERACTIONS ---------//
  public UniSyncReceiverMonitorPort receive_config_request =
    new UniSyncReceiverMonitorPort(this) {
      public synchronized void send()
                              throws InterruptedException,
                                     NotReadyPortException {
        try {
          core_monitor_State_Repository.receive_config_request(isBlocking);
        } catch(SemisyncInteractionException e) {
          throw new NotReadyPortException(e);
        }
      }
```

```
  };

public UniSyncReceiverMonitorPort receive_consistent_config_request =
  new UniSyncReceiverMonitorPort(this) {
    public synchronized void send()
                           throws InterruptedException,
                                  NotReadyPortException {
      try {
        core_monitor_State_Repository.receive_consistent_config_request(isBlocking);
      } catch(SemisyncInteractionException e) {
        throw new NotReadyPortException(e);
      }
    }
  };

public UniSyncReceiverMonitorPort receive_consistency_notification =
  new UniSyncReceiverMonitorPort(this) {
    public synchronized void send()
                           throws InterruptedException,
                                  NotReadyPortException {
      try {
        core_monitor_State_Repository.receive_consistency_notification(isBlocking);
      } catch(SemisyncInteractionException e) {
        throw new NotReadyPortException(e);
      }
    }
  };

public OrSyncReceiverMonitorPort read_actor_state =
  new OrSyncReceiverMonitorPort(this) {
    public synchronized void send(Object[] inputPars)
                           throws InterruptedException,
                                  NotReadyPortException {
      try {
        core_monitor_State_Repository.read_actor_state(isBlocking,
                                       (Interface_ActorState)inputPars[0]);
      } catch(SemisyncInteractionException e) {
        throw new NotReadyPortException(e);
      }
    }
  };

public UniSyncSenderMonitorPort forward_config =
  new UniSyncSenderMonitorPort(this) {
    public synchronized Object[] receive()
                               throws InterruptedException,
                                      NotReadyPortException {
      Object[] outputPars;
      try {
        outputPars = core_monitor_State_Repository.forward_config(isBlocking);
      } catch(SemisyncInteractionException e) {
```

```
        throw new NotReadyPortException(e);
      }
      return outputPars;
    }
  };


public UniSyncSenderMonitorPort forward_consistent_config =
  new UniSyncSenderMonitorPort(this) {
    public synchronized Object[] receive()
                               throws InterruptedException,
                                      NotReadyPortException {
      Object[] outputPars;
      try {
        outputPars = core_monitor_State_Repository.forward_consistent_config(isBlocking);
      } catch(SemisyncInteractionException e) {
        throw new NotReadyPortException(e);
      }
      return outputPars;
    }
  };

//----------- DEFINING CONSTRUCTOR ------------//
Console(int allowed_changes) {
  core_monitor_State_Repository = new State_Repository_Monitor();
}

//--------- RUNNING ELEMENT [monitor] ---------//
public void start() {
  run();
}

public void join() throws InterruptedException { }

public void run() {
  core_monitor_State_Repository.startMonitor();
}
}
```

All the other AETs occurring in the PADL description of the video animation repainting system are translated into threads. As far as the `Frame_Generator` and the `Snapshot_Generator` classes are concerned, whose descriptions are very similar, the code generated for them is:

```
class Frame_Generator implements RunnableElem {

  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }
  BehavioralEquationInterface Initialization,
```

```
                            Generation;
BehavioralEquationInterface nextBehavEq;
Object[] actualPars;

//-------- INSTANTIATING INTERACTIONS ---------//
UniSyncReceiverPort read_config =
  new UniSyncReceiverPort(this);

UniSyncSenderPort require_config =
  new UniSyncSenderPort(this);

//-------------- DECLARING STUBS --------------//
IAS_Frame_Generator internal_Frame_Generator;
// No EHS declaration as there are
// no architectural interactions and
// no semi-synchronous interactions.

//----------- DEFINING CONSTRUCTOR ------------//
protected int clock_rate;

Frame_Generator(int clock_rate) {
  this.clock_rate = clock_rate;
  defineBehavEquations();
}

//------------- DEFINING BEHAVIOR -------------//

void defineBehavEquations() {

  Initialization =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Initialization();
      }

      private void _Initialization() {
        internal_Frame_Generator.start_internal_clock(clock_rate);
        nextBehavEq = Generation;
        actualPars = null;
      }

    }; // end of behavioral equation Initialization

  Generation =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Generation();
      }
```

```
        private void _Generation() {
          Interface_Configuration config;
          Object[] inputPars;
          internal_Frame_Generator.wait_for_tick();
          try {
            require_config.send();
          } catch(SyncException e) {}
          try {
            inputPars = read_config.receive();
            config = (Interface_Configuration)inputPars[0];
          } catch(SyncException e) {}
          internal_Frame_Generator.generate_and_print_frame(config);
          nextBehavEq = Generation;
          actualPars = null;
        }

      }; // end of behavioral equation Generation

  }

  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Frame_Generator = null;

  public void start() {
    (th_Frame_Generator = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Frame_Generator.join();
  }

  public void run() {
    internal_Frame_Generator =
      new IAS_Frame_Generator();
    nextBehavEq = Initialization;
    actualPars = null;
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}


class Snapshot_Generator implements RunnableElem {

  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }
  BehavioralEquationInterface Snapshot;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
```

```
UniSyncReceiverPort click_event =
  new UniSyncReceiverPort(this);
UniSyncReceiverPort read_consistent_config =
  new UniSyncReceiverPort(this);

UniSyncSenderPort require_consistent_config =
  new UniSyncSenderPort(this);

//------------- DECLARING STUBS -------------//
IAS_Snapshot_Generator internal_Snapshot_Generator;
EHS_Snapshot_Generator exception_Snapshot_Generator;

//---------- DEFINING CONSTRUCTOR -----------//
Snapshot_Generator() {
  defineBehavEquations();
}

//------------ DEFINING BEHAVIOR ------------//
void defineBehavEquations() {

  Snapshot =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Snapshot();
      }

      private void _Snapshot() {
        Interface_Configuration config;
        Object[] inputPars;
        try {
          click_event.receive();
        } catch(UnattachedPortException e) {
          exception_Snapshot_Generator.click_event();
        }
        try {
          require_consistent_config.send();
        } catch(SyncException e) {}
        try {
          inputPars = read_consistent_config.receive();
          config = (Interface_Configuration)inputPars[0];
        } catch(SyncException e) {}
        internal_Snapshot_Generator.generate_and_print_snapshot(config);
        nextBehavEq = Snapshot;
        actualPars = null;
      }

    }; // end of behavioral equation Snapshot

}
```

```
  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Snapshot_Generator = null;

  public void start() {
    (th_Snapshot_Generator = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Snapshot_Generator.join();
  }

  public void run() {
    internal_Snapshot_Generator =
      new IAS_Snapshot_Generator();
    exception_Snapshot_Generator =
      new EHS_Snapshot_Generator();
    nextBehavEq = Snapshot;
    actualPars = null;
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}
```

The class `Actor`, generated from the homonymous AET, contains the semi-synchronous port `receive_call` endowed with the boolean method `success()` according to the boolean value `success` of the related PADL semi-synchronous interaction. The following code is produced for `Actor`:

```
class Actor implements RunnableElem {

  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }
  BehavioralEquationInterface Initialization,
                              Free_Acting,
                              Availability;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
  UniSemisyncReceiverPort receive_call =
    new UniSyncReceiverPort(this);
  UniSyncReceiverPort receive_notification =
    new UniSyncReceiverPort(this);

  UniSyncSenderPort write_state =
    new UniSyncSenderPort(this);
  UniSyncSenderPort communicate_availability =
    new UniSyncSenderPort(this);
```

```
//------------- DECLARING STUBS -------------//
IAS_Actor internal_Actor;
EHS_Actor exception_Actor;

//---------- DEFINING CONSTRUCTOR -----------//
protected int actor_id;

Actor(int actor_id) {
  this.actor_id = actor_id;
  defineBehavEquations();
}

//------------ DEFINING BEHAVIOR ------------//
void defineBehavEquations() {

  Initialization =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Initialization();
      }

      private void _Initialization() {
        internal_Actor.store_identifier(actor_id);
        nextBehavEq = Free_Acting;
        actualPars = null;
      }

    }; // end of behavioral equation Initialization

  Free_Acting =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Free_Acting();
      }

      private void _Free_Acting() {
        Interface_ActorState actor_state;
        Object[] inputPars;
        inputPars = internal_Actor.play();
        actor_state = (Interface_ActorState)inputPars[0];
        try {
          receive_call.receive();
        catch(NotReadyPortException e) {
          exception_Actor.receive_call();
        }
        switch (
          ElemMeth.choice(
            new ChAct[] {
```

```
            new ChAct(!receive_call.success(), write_state),
            new ChAct(receive_call.success(), null)
          }
        )
      ) // Choice body :
      {
        case 0:
          try {
            write_state.send(actor_state);
          } catch(SyncException e) {}
          nextBehavEq = Free_Acting;
          actualPars = null;
          break;
        case 1:
          inputPars = internal_Actor.prepare_to_snapshot();
          actor_state = (Interface_ActorState)inputPars[0];
          try {
            write_state.send(actor_state);
          } catch(SyncException e) {}
          nextBehavEq = Availability;
          actualPars = null;
          break;
        default:
          nextBehavEq = null;  // STOP
          actualPars = null;
      }
    }

  }; // end of behavioral equation Free_Acting

 Availability =
   new BehavioralEquationInterface() {

     public void behavEqCall() {
       _Availability();
     }

     private void _Availability() {
       try {
         communicate_availability.send();
       } catch(SyncException e) {}
       try {
         receive_notification.receive();
       } catch(SyncException e) {}
       nextBehavEq = Free_Acting;
       actualPars = null;
     }

   }; // end of behavioral equation Availability

}
```

```
  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Actor = null;

  public void start() {
    (th_Actor = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Actor.join();
  }

  public void run() {
    internal_Actor =
      new IAS_Actor();
    exception_Actor =
      new EHS_Actor();
    nextBehavEq = Initialization;
    actualPars = null;
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}
```

The last AET, i.e. `Director`, is translated as follows:

```
class Director implements RunnableElem {

  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }
  BehavioralEquationInterface Initialization,
                              Preparation,
                              Calling_Actors,
                              Notification;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
  AndSyncReceiverPort wait_for_actors_availability =
    new AndSyncReceiverPort(this);

  UniSyncSenderPort notify_consistency =
    new UniSyncSenderPort(this);
  AndSyncSenderPort notify_actors =
    new AndSyncSenderPort(this);
  OrSyncSenderPort call_actor =
    new OrSyncSenderPort(this);

  //-------------- DECLARING STUBS --------------//
  IAS_Director internal_Director;
```

```
// No EHS declaration as there are
// no architectural interactions and
// no semi-synchronous interactions.

//----------- DEFINING CONSTRUCTOR ------------//
protected int n_actors;

Director(int n_actors) {
  this.n_actors = n_actors;
  defineBehavEquations();
}

//------------- DEFINING BEHAVIOR -------------//
void defineBehavEquations() {

  Initialization =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Initialization();
      }

      private void _Initialization() {
        internal_Director.start();
        nextBehavEq = Preparation;
        actualPars = null;
      }

    }; // end of behavioral equation Initialization

  Preparation =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Preparation();
      }

      private void _Preparation() {
        internal_Director.do_something();
        nextBehavEq = Calling_Actors;
        actualPars = new Object[] {n_actors};
      }

    }; // end of behavioral equation Preparation

  Calling_Actors =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Calling_Actors((int)actualPars[0]);
      }
```

```
      private void _Calling_Actors(int n_actors_to_call) {
        try {
          call_actor.send();
        } catch(SyncException e) {}
        switch (
          ElemMeth.choice(
            new ChAct[] {
              new ChAct(n_actors_to_call > 0, call_actor),
              new ChAct(n_actors_to_call == 0, wait_for_actors_availability)
            }
          )
        ) // Choice body :
        {
          case 0:
            try {
              call_actor.send();
            } catch(SyncException e) {}
            nextBehavEq = Calling_Actors;
            actualPars = new Object[] {n_actors_to_call - 1};
            break;
          case 1:
            try {
              wait_for_actors_availability.receive();
            } catch(SyncException e) {}
            nextBehavEq = Notification;
            actualPars = null;
            break;
          default:
            nextBehavEq = null;  // STOP
            actualPars = null;
        }
      }

    }; // end of behavioral equation Calling_Actors

  Notification =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Notification();
      }

      private void _Notification() {
        try {
          notify_consistency.send();
        } catch(SyncException e) {}
        try {
          notify_actors.send();
        } catch(SyncException e) {}
        nextBehavEq = Preparation;
```

```
        actualPars = null;
      }

    }; // end of behavioral equation Notification

  }

  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Director = null;

  public void start() {
    (th_Director = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Director.join();
  }

  public void run() {
    internal_Director =
      new IAS_Director();
    nextBehavEq = Initialization;
    actualPars = null;
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}
```

Finally, seven stub classes are generated for handling the internal actions and the exceptions of the previous classes. More precisely, five IAS classes are generated associated to all of the original AETs, while only two EHS classes are generated associated to `Snapshot_Generator` and to `Actor`:

```
class IAS_State_Repository {

  IAS_State_Repository() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void store_consistent_config(Interface_Configuration config) {
    // FILL IN THE METHOD BODY
  }

  void store_actor_state(Interface_ActorState actor_state) {
    // FILL IN THE METHOD BODY
  }

  Object[] get_config() {
```

```
    Interface_Configuration config = null;
    // FILL IN THE METHOD BODY
    return new Object[] {config};
  }

  void mark_config_as_consistent() {
    // FILL IN THE METHOD BODY
  }

  Interface_Configuration get_last_consistent_config() {
    Interface_Configuration config = null;
    // FILL IN THE METHOD BODY
    return new Object[] {config};
  }
}


class IAS_Frame_Generator {

  IAS_Frame_Generator() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void start_internal_clock(int clock_rate) {
    // FILL IN THE METHOD BODY
  }

  void wait_for_tick() {
    // FILL IN THE METHOD BODY
  }

  void generate_and_print_frame(Interface_Configuration config) {
    // FILL IN THE METHOD BODY
  }
}


class IAS_Snapshot_Generator {

  IAS_Snapshot_Generator() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void generate_and_print_snapshot(Interface_Configuration config) {
    // FILL IN THE METHOD BODY
  }
}


class EHS_Snapshot_Generator {
```

```
  EHS_Snapshot_Generator() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void click_event() {
    // FILL IN THE METHOD BODY
  }
}


class IAS_Actor {

  IAS_Actor() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void store_identifier(int actor_id) {
    // FILL IN THE METHOD BODY
  }

  Object[] play() {
    Interface_ActorState actor_state = null;
    // FILL IN THE METHOD BODY
    return new Object[] {actor_state};
  }

  Object[] prepare_to_snapshot() {
    Interface_ActorState actor_state = null;
    // FILL IN THE METHOD BODY
    return new Object[] {actor_state};
  }
}


class EHS_Actor {

  EHS_Actor() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void receive_call() {
    // FILL IN THE METHOD BODY
  }
}


class IAS_Director {

  IAS_Director() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }
```

```
    void start() {
      // FILL IN THE METHOD BODY
    }

    void do_something() {
      // FILL IN THE METHOD BODY
    }
  }
```

Once the stub classes for handling local actions and exceptions of the generated video animation repainting system are filled in according to the guidelines provided in Sect. 5.7, deadlock freedom is preserved at the code level.

8.2 A Leader Election System

Leader election is the problem of electing a unique process leader in a network of processes. The leader must know that it has been elected and the other processes must know that they have not been elected. Leader election algorithms require that all processes have the same local algorithm and that each computation terminates, with one process elected as leader.

In this section a specification of a leader election system based on an algorithm proposed in [32] is provided. Then, the related Java code is synthesized from the specification with PADL2Java. Since some properties will be evaluated at the process algebraic description level by means of the performance evaluator provided by TwoTowers, the description language Æmilia will be used instead of PADL.

8.2.1 The Fokkink-Pang Leader Election Algorithm

In literature, many algorithms have been proposed for the problem of leader election. They vary in communication synchronicity (synchronous vs. asynchronous), process names (unique identities vs. anonymous), and network topology (e.g. ring, tree, complete graph). Sometimes the processes in a network cannot be distinguished by means of unique identities. In an "anonymous network", processes do not carry an identity. For asynchronous anonymous network, it has

been proved [5] that there does not exist a terminating algorithm for electing a leader. According to this result, a “Las Vegas” algorithm – i.e., an algorithm in which the probability of termination is greater than zero and all terminal configurations are correct – is the best possible option.

In [32], Fokkink and Pang present two probabilistic leader election algorithms for anonymous unidirectional rings with asynchronous communication channels that simplify the algorithm proposed by Itai and Rodeh in [44] by means of the introduction of FIFO queues as channels. All of these algorithms are “Las Vegas” with probability of termination equal to one. However, conversely to the Itai-Rodeh algorithm, the Fokkink-Pang algorithms are finite-state and can be analyzed using explicit state space exploration. In particular, the probabilistic symbolic model checker PRISM [49] has been used by the authors for this purpose.

In [44, 32] a probabilistic leader election algorithm for anonymous unidirectional rings is defined as a system in which each process selects a random identity from a finite domain, and sends a message around the ring bearing its identity. A process that detect a name clash, meaning that the process receives a message with its own identity, starts a new election round. A process that receives a message with an identity larger than its own identity knows that it cannot be a leader. The process with the largest identity becomes the leader. It is assumed that the size of the ring is known to all processes, so that each process can recognize its own message by means of a hop counter that is part of the message.

The Itai-Rodeh algorithm contains a third piece of information in addition to the random identity and to the hop counter, which is the round number. This is useful when an old message, that has been overtaken by other messages in the ring, results in a situation where no leader is elected. This makes the Itai-Rodeh algorithm infinite-state. Fokkink and Pang claim that round numbers can be omitted from the message, with the assumption that communication channels among two contiguous processes are FIFO. Basically, the two algorithms $\mathcal{A}$ and $\mathcal{B}$ proposed by Fokkink and Pang are two different adaptations of the Itai-Rodeh algorithm that are correct in the presence of FIFO channels.

In this case study, only the second one – i.e. the algorithm $\mathcal{B}$ – is proposed, whose steps are summarized as follows:

- Initially, all processes are active, and each process p_i randomly selects its identity $id_i \in \{0, \ldots, k-1\}$ and sends the message $(id_1, 1)$.

- Upon receipt of a message (id_1, hop), a passive process p_i ($state_i = passive$) passes on the message, increasing the counter hop by one. An active process p_i ($state_i = active$) behaves according to one of the following steps:

  – if $hop = n$, then p_i becomes the leader ($state'_i = leader$);

  – if $id = id_i$ and $hop < n$, then p_i selects a new random identity $id'_i \in \{0, \ldots, k-1\}$ and sends the message $(id'_1, 1)$;

  – if $id > id_i$, then p_i becomes passive ($state'_i = passive$) and passes on the message $(id_1, hop + 1)$;

  – if $id < id_i$, then p_i purges the message.

8.2.2 Modeling a Leader Election System

Before modeling a system based on the algorithm $\mathcal{B}$ proposed by Fokkink-Pang, some considerations must be done in order to allow finite-state analysis and probabilistic measurement at the process algebraic description level. Through appropriate analysis and measurement tools, we will then be able to check the correctness of our model and to estimate the probability of electing a leader at a certain time, once the election has been started.

The first point to take into consideration is that the communication channels should be modeled with AETs that behaves as bounded FIFO queues. In fact, even if asynchronous connectors could be used instead of such AETs, bounded queues explicitly points out the sequentiality of the communications between processes and, in particular, leads to a finite-state system.

Second, since processes are anonymous, and since they can exchange messages among them only through a ring-topology network, it may be useful to introduce an external supervisor for checking the correctness of the election. In practice, each process simply communicates to the supervisor if it has been elected or not. When the last process communicates its status, the supervisor knows if one (and only one) leader has been elected.

Third, the description language Æmilia should be used instead of PADL in order to exploit the performance evaluator provided by TwoTowers, with which we will be able to accomplish probabilistic measurements on our system. As explained in [9], in fact, the performance evaluator is based on the performance semantic model of Æmilia. This model can be extracted in the form of a Markov chain [63] only when an Æmilia description is performance closed, i.e., when no passive transitions and no non-determinism arises because of some boolean condition – occurring in a behavioral choice – that cannot be statically evaluated. If only immediate transitions occur in the performance semantic model, they are interpreted as taking one time unit and the model corresponds to a discrete-time Markov chain, with the transitions labeled with the probabilities of the corresponding actions.

Hence, in order to analyze a discrete-time performance model of the leader election system, all the internal actions of the Æmilia description should be declared as immediate using the qualifier "`inf`", which means that they are performed at infinite rate. As far as the interactions are concerned, since an immediate action can interact only with a passive one, all the output interactions should be qualified as "`inf`", while all the input interactions should be declared as passive using the qualifier "_".

From the point of view of the code generator PADL2Java, we recall from 7.7 that the syntactical differences between a PADL description and its improved version in Æmilia are simply ignored during the parsing process.

8.2.3 Æmilia Description of the Leader Election System

The architecture of the leader election system is illustrated in Fig. 8.2. The upper part of the diagram shows a ring of processes (`P[1], ..., P[n_procs]`) where a bounded FIFO queue (`BF[1], ..., BF[n_procs]`) is interposed between each couple of contiguous processes. Each process sends a message to a queue through the interaction `send_message` and receives a message from a queue through the interaction `receive_message`.

In order to stop all the processes when a leader is elected, all processes and all queues have been endowed with additional interactions, i.e., an input interaction

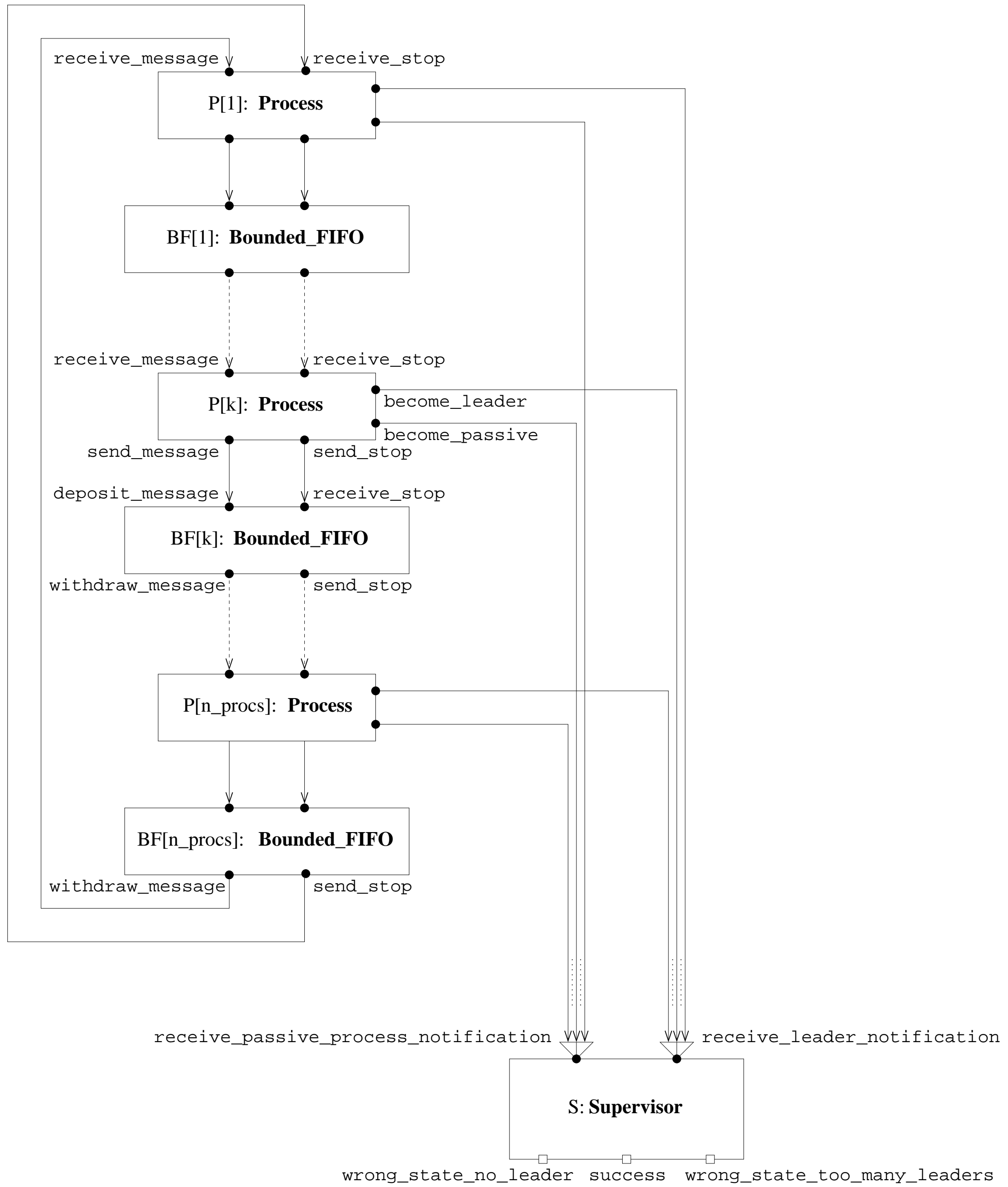

Figure 8.2. Extended flow graph of the leader election system

receive_stop and an output interaction send_stop. When a process becomes leader, it forwards (through the queue) the stop-signal to its contiguous process, which in turn forwards the signal to another process, and then terminates. The leader process terminates, instead, as soon as the stop-signal has done a whole round around the ring, coming back to the leader.

In the bottom-right part of the diagram, the Supervisor is shown that receives the signal receive_passive_process_notification from all the processes that become passive, and receives the signal receive_leader_notification from the process that becomes a leader. The interaction success succeeds if one (and only one) process communicates its leadership, while wrong_state_no_leader and wrong_state_too_many_leaders should never succeed in a correct specification of the algorithm.

The whole system, called Leader_Election_B is described in Æmilia as follows, where parameters n_ids, n_procs, and queue_capacity are respectively the number of processes, the number of different identities that the processes can take, and the maximum size of bounded FIFO queues:

```
ARCHI_TYPE Leader_Election_B(const integer n_ids          := 3,
                             const integer n_procs        := 3,
                             const integer queue_capacity := 3)

ARCHI_BEHAVIOR

        ARCHI_ELEM_TYPE Process(const integer n_ids,
                                const integer n_procs)

            BEHAVIOR

              Start(integer(0..n_ids) self_id := d_uniform(0, n_ids - 1);
                    void) =
                <send_message!(self_id, 1), inf> .
                  Active_Message_Receiving(self_id);

              Active_Message_Receiving(integer(0..n_ids) self_id;
                                       local integer(0..n_ids) id,
                                       local integer(1..n_procs) hop) =
                <receive_message?(id, hop), _> . Evaluation(self_id,
                                                            id,
                                                            hop);

              Evaluation(integer(0..n_ids) self_id,
                         integer(0..n_ids) id,
                         integer(1..n_procs) hop;
```

```
               void) =
        <count_evaluation_steps, inf> .
          choice
          {
            cond(hop = n_procs) ->
              <become_leader, inf> . Leadership(),
            cond((hop < n_procs) && (id = self_id)) ->
              <new_identity, inf> . Start(d_uniform(0, n_ids - 1)),
            cond(id > self_id) ->
              <become_passive, inf> . <send_message!(id, hop + 1), inf> .
                Passive_Message_Receiving(),
            cond(id < self_id) ->
              <purge_message, inf> .
                Active_Message_Receiving(self_id)
          };

      Leadership(void;
                 void) =
        <announce_leadership, inf> .
          <send_stop, inf> .
            <receive_stop, _> . stop;

      Passive_Message_Receiving(void;
                                local integer(0..n_ids) id,
                                local integer(1..n_procs) hop) =
        choice
        {
          <receive_message?(id, hop), _> .
            <send_message!(id, hop + 1), inf> .
              Passive_Message_Receiving(),
          <receive_stop, _> .
            <send_stop, inf> . stop
         }

    INPUT_INTERACTIONS

      UNI receive_message;
          receive_stop

    OUTPUT_INTERACTIONS

      UNI send_message;
          send_stop;
          become_leader;
          become_passive


ARCHI_ELEM_TYPE Bounded_FIFO(const integer n_ids,
                             const integer n_procs,
                             const integer capacity)
```

```
    BEHAVIOR

      Queue(array(capacity, integer(0..n_ids)) id_arr := array_cons(0, 0, 0),
            array(capacity, integer(1..n_procs)) hop_arr := array_cons(1, 1, 1),
            integer(0..capacity) head_index := 0,
            integer(0..capacity) size := 0;
            local integer(0..n_ids) rec_id,
            local integer(1..n_procs) rec_hop) =
        choice
        {
          cond(size < capacity) -> % queue is not full
            <deposit_message?(rec_id, rec_hop), _> .
              Queue(write(mod(head_index + size, capacity), rec_id, id_arr) ,
                    write(mod(head_index + size, capacity), rec_hop, hop_arr),
                    head_index,
                    size + 1),
          cond(size > 0) -> % queue is not empty
            <withdraw_message!(read(head_index, id_arr),
                               read(head_index, hop_arr)), inf> .
              Queue(id_arr,
                    hop_arr,
                    mod(head_index + 1, capacity),
                    size - 1),
         <receive_stop, _> . <send_stop, inf> . stop
        }

    INPUT_INTERACTIONS

      UNI deposit_message;
          receive_stop

    OUTPUT_INTERACTIONS

      UNI withdraw_message;
          send_stop


ARCHI_ELEM_TYPE Supervisor(const integer n_procs)

    BEHAVIOR

      Waiting_for_Leader(integer(0..n_procs) active_processes := n_procs;
                         void) =
        choice
        {
          cond(active_processes > 0) ->
            <receive_passive_process_notification, _> .
              Waiting_for_Leader(active_processes - 1),
          cond(active_processes > 0) ->
            <receive_leader_notification, _> .
              Waiting_for_Processes(active_processes - 1),
```

```
        cond(active_processes = 0) ->
          <wrong_state_no_leader, inf> . stop
      };

    Waiting_for_Processes(integer(0..n_procs) active_processes;
                          void) =
      choice
      {
        cond(active_processes > 0) ->
          <receive_passive_process_notification, _> .
            Waiting_for_Processes(active_processes - 1),
        cond(active_processes > 0) ->
          <receive_leader_notification, _> .
            <wrong_state_too_many_leaders, inf> . stop,
        cond(active_processes = 0) ->
          <success, inf> . stop
      }

  INPUT_INTERACTIONS

    OR receive_leader_notification;
       receive_passive_process_notification

  OUTPUT_INTERACTIONS

    UNI success;
        wrong_state_no_leader;
        wrong_state_too_many_leaders


ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES

    FOR_ALL i IN 1..n_procs
      P[i] : Process(n_ids,
                     n_procs);
    FOR_ALL i IN 1..n_procs
      BF[i] : Bounded_FIFO(n_ids,
                           n_procs,
                           queue_capacity);
    S : Supervisor(n_procs)


  ARCHI_INTERACTIONS

    S.success;
    S.wrong_state_no_leader;
    S.wrong_state_too_many_leaders
```

```
    ARCHI_ATTACHMENTS

      FOR_ALL i IN 1..n_procs
        FROM P[i].send_message TO BF[i].deposit_message;
      FOR_ALL i IN 1..n_procs
        FROM BF[i].withdraw_message TO P[mod(i, n_procs) + 1].receive_message;
      FOR_ALL i IN 1..n_procs
        FROM P[i].send_stop TO BF[i].receive_stop;
      FOR_ALL i IN 1..n_procs
        FROM BF[i].send_stop TO P[mod(i, n_procs) + 1].receive_stop;
      FOR_ALL i IN 1..n_procs
        FROM P[i].become_leader TO S.receive_leader_notification;
      FOR_ALL i IN 1..n_procs
        FROM P[i].become_passive TO S.receive_passive_process_notification

END
```

Note that the defining equation `Evaluation` of the AET `Process` contains the same rules specified in Sect. 8.2.1 for the Fokkink-Pang algorithm $\mathcal{B}$. Also note that `d_uniform()`, a pseudo-random number generator [45] of PADL, has been used in order to produce a random number following a discrete uniform distribution between the two arguments `0` and `n_ids - 1`.

8.2.4 Analyzing the Leader Election System

In this section we assess the correctness of the Æmilia description of the leader election system. Then, we measure the probability that the underlying Fokkink-Pang algorithm $\mathcal{B}$ terminates within a given number of transitions.

In order to assess the correctness of our description, two properties have been verified through the LTL model checker of TwoTowers. These properties refer to the architectural interactions `S.wrong_state_no_leader` and `S.wrong_state_too_many_leaders` that should never be executed – no future state should satisfy the interactions – since one and only one leader must be elected. The specification of the two properties, written in a `.ltl` file, is as follows:

```
PROPERTY at_least_one IS
  NOT(SOME_FUTURE_STATE_SAT(S.wrong_state_no_leader));

PROPERTY at_most_one IS
  NOT(SOME_FUTURE_STATE_SAT(S.wrong_state_too_many_leaders))
```

and the outcome of their verification is:

```
Validity of the properties for Leader_Election_B:

- Property "at_least_one" is satisfied.

- Property "at_most_one" is satisfied.
```

In order to verify the correct termination of the algorithm, the following specification could be used:

```
PROPERTY termination IS
  SOME_FUTURE_STATE_SAT(S.success);
```

but it does not lead to a successful result. The following counterexample is produced by the model checker:

```
- Property "termination" isn't satisfied
  as demonstrated by the following execution sequence:
...
<<loop starts here>>
...
```

This is due to the fact that the LTL model checker is not probabilistic, and it points out a situation in which all the processes always randomly select the same identity.

However, the performance evaluator of TwoTowers allows the calculation of the stationary probability distribution for the state space of the performance semantic model of the Æmilia description. When the number of processes, the number of identities, and the capacity of the bounded queue are set to 3, the following (piece of) result is given by the evaluator:

```
- ...
- Global state 81:      0
- Global state 82:      0
- Global state 83:      1
- Global state 84:      0
- Global state 85:      0
- ...
```

that means that the algorithm terminates at state 83 with probability one. On the basis of the performance semantic model produced in a readable text format by the compiler of TwoTowers, it is possible to verify that such a state, described as:

```
>> Global state 83:
        ...
        - Local state of P[1]:
                stop
        - Local state of P[2]:
                stop
        - Local state of P[3]:
                stop
        - Local state of BF[1]:
                stop
        - Local state of BF[2]:
                stop
        - Local state of BF[3]:
                stop
        - Local state of S:
                stop
        - No transitions.
```

is always reached after the action `S.success` has succeeded in some previous state. This is sufficient to confirm the correct termination of the algorithm.

In order to estimate the probability of electing a leader within a given number of transitions, the transient probability calculator of TwoTowers has been used instead. Recalled that the performance semantic model underlying our description is a discrete-time one, at different transient instants (0, 5, 10, . . . , 70) the following table can be obtained for the absorbing state 83:

```
 0 - Global state 83:     0
 5 - Global state 83:     0
10 - Global state 83:     0
15 - Global state 83:     0
20 - Global state 83:     7.13628E-05
25 - Global state 83:     0.00406503
30 - Global state 83:     0.0500913
35 - Global state 83:     0.226095
40 - Global state 83:     0.522359
45 - Global state 83:     0.787218
50 - Global state 83:     0.930883
55 - Global state 83:     0.98306
60 - Global state 83:     0.99674
65 - Global state 83:     0.999488
70 - Global state 83:     0.999932
```

which indicates the probability of electing a leader as a function of discrete-time steps, when the number of processes and of identities is 3.

Other measurements have been performed by varying the number of processes and of identities in our Æmilia description. Some results are illustrated in Fig. 8.3.

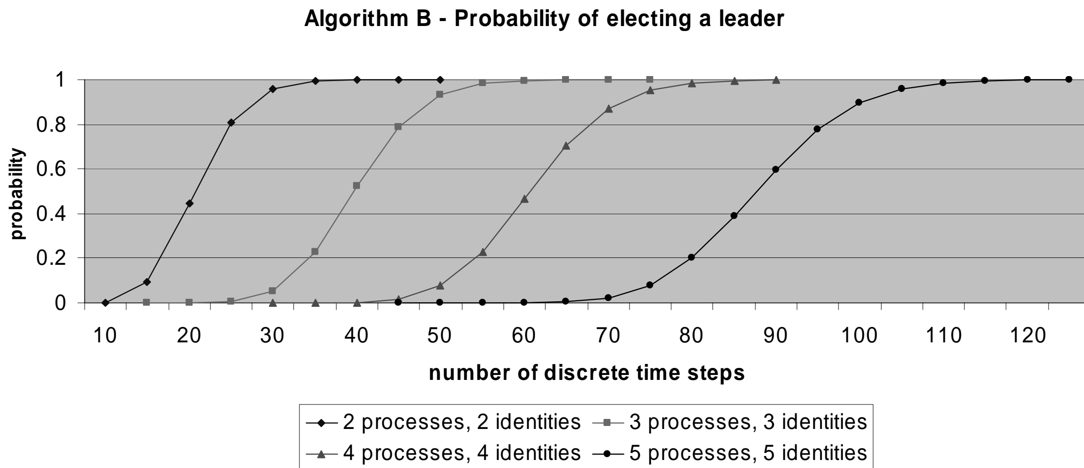

Figure 8.3. Probability of electing a leader

8.2.5 Synthesizing the Leader Election System

The translator PADL2Java has been used for generating code from the Æmilia description of the leader election system (`Leader_Election_B`). Note that the AET `Bounded_FIFO`, whose instances satisfy all the monitor constraints, is synthesized as a monitor, while all the other AETs are synthesized as threads:

```
public class Leader_Election_B implements RunnableArchi {

  //-------- DECLARING RUNNABLE ELEMENTS --------//
  Sync.util.HashArray<Process> P;
  Sync.util.HashArray<Bounded_FIFO> BF;
  Supervisor S;

  //--- DECLARING ARCHITECTURAL INTERACTIONS ----//
  public UniSyncSenderPort S_success;
  public UniSyncSenderPort S_wrong_state_no_leader;
  public UniSyncSenderPort S_wrong_state_too_many_leaders;

  //----------- DEFINING CONSTRUCTOR ------------//
  protected int n_ids;
  protected int n_procs;
  protected int queue_capacity;

  // GENERAL CONSTRUCTOR:
  Leader_Election_B(int n_ids,
                    int n_procs,
                    int queue_capacity) {
    this.n_ids = n_ids;
    this.n_procs = n_procs;
```

```
    this.queue_capacity = queue_capacity;
    buildArchiTopology();
  }

  // DEFAULT CONSTRUCTOR:
  Leader_Election_B() {
    this(3,
         3,
         3);
  }

  //----------- BUILDING ARCHITECTURE -----------//
  void buildArchiTopology() {

    //  INSTANTIATING RUNNABLE ELEMENTS:
    P = new Sync.util.HashArray<Process>();
    for (int i = 1; i <= n_procs; i++)
      P.put(i,
            new Process(n_ids,
                        n_procs));
    BF = new Sync.util.HashArray<Bounded_FIFO>();
    for (int i = 1; i <= n_procs; i++)
      BF.put(i,
             new Bounded_FIFO(n_ids,
                              n_procs,
                              queue_capacity));
    S = new Supervisor(n_procs);

    //  ASSIGNING ARCHITECTURAL INTERACTIONS:
    this.S_success = S.success;
    this.S_wrong_state_no_leader = S.wrong_state_no_leader;
    this.S_wrong_state_too_many_leaders = S.wrong_state_too_many_leaders;

    //  ATTACHING LOCAL INTERACTIONS:
    try {
      for (int i = 1; i <= n_procs; i++)
        ArchiMeth.attach(P.get(i).send_message, BF.get(i).deposit_message);
      for (int i = 1; i <= n_procs; i++)
        ArchiMeth.attach(BF.get(i).withdraw_message, P.get(i % n_procs + 1).receive_message);
      for (int i = 1; i <= n_procs; i++)
        ArchiMeth.attach(P.get(i).send_stop, BF.get(i).receive_stop);
      for (int i = 1; i <= n_procs; i++)
        ArchiMeth.attach(BF.get(i).send_stop, P.get(i % n_procs + 1).receive_stop);
      for (int i = 1; i <= n_procs; i++)
        ArchiMeth.attach(P.get(i).become_leader, S.receive_leader_notification);
      for (int i = 1; i <= n_procs; i++)
        ArchiMeth.attach(P.get(i).become_passive, S.receive_passive_process_notification);
    } catch(BadAttachmentException e) {}

  } // end of method buildArchiTopology()
```

```
  //---------- RUNNING ARCHITECTURE  -----------//
  Thread th_Leader_Election_B = null;

  public void start() {
    (th_Leader_Election_B = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Leader_Election_B.join();
  }

  public void run() {
    for (int i = 1; i <= n_procs; i++)
      P.get(i).start();
    for (int i = 1; i <= n_procs; i++)
      BF.get(i).start();
    S.start();
    try {
      for (int i = 1; i <= n_procs; i++)
        P.get(i).join();
      for (int i = 1; i <= n_procs; i++)
        BF.get(i).join();
      S.join();
    } catch(InterruptedException e) {}
  }
}


class Process implements RunnableElem {

  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }
  BehavioralEquationInterface Start,
                              Active_Message_Receiving,
                              Evaluation,
                              Leadership,
                              Passive_Message_Receiving;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
  UniSyncReceiverPort receive_message =
    new UniSyncReceiverPort(this);
  UniSyncReceiverPort receive_stop =
    new UniSyncReceiverPort(this);

  UniSyncSenderPort send_message =
    new UniSyncSenderPort(this);
  UniSyncSenderPort send_stop =
    new UniSyncSenderPort(this);
  UniSyncSenderPort become_leader =
    new UniSyncSenderPort(this);
```

```
UniSyncSenderPort become_passive =
  new UniSyncSenderPort(this);

//-------------- DECLARING STUBS --------------//
IAS_Process internal_Process;
// No EHS declaration as there are
// no architectural interactions and
// no semi-synchronous interactions.

//----------- DEFINING CONSTRUCTOR ------------//
protected int n_ids;
protected int n_procs;

Process(int n_ids,
        int n_procs) {
  this.n_ids = n_ids;
  this.n_procs = n_procs;
  defineBehavEquations();
}

//------------- DEFINING BEHAVIOR -------------//
void defineBehavEquations() {

  Start =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Start((int)actualPars[0]);
      }

      private void _Start(int self_id) {
        try {
          send_message.send(self_id,
                            1);
        } catch(SyncException e) {}
        nextBehavEq = Active_Message_Receiving;
        actualPars = new Object[] {self_id};
      }

    }; // end of behavioral equation Start

  Active_Message_Receiving =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Active_Message_Receiving((int)actualPars[0]);
      }

      private void _Active_Message_Receiving(int self_id) {
        int id;
        int hop;
```

```
      Object[] inputPars;
      try {
        inputPars = receive_message.receive();
        id = (int)Object[0];
        hop = (int)Object[1];
      } catch(SyncException e) {}
      nextBehavEq = Evaluation;
      actualPars = new Object[] {self_id,
                                 id,
                                 hop};
    }

  }; // end of behavioral equation Active_Message_Receiving

Evaluation =
  new BehavioralEquationInterface() {

    public void behavEqCall() {
      _Evaluation((int)actualPars[0],
                  (int)actualPars[1],
                  (int)actualPars[2]);
    }

    private void _Evaluation(int self_id,
                             int id,
                             int hop) {
      internal_Process.count_evaluation_steps();
      switch (
        ElemMeth.choice(
          new ChAct[] {
            new ChAct(hop == n_procs, become_leader),
            new ChAct((hop < n_procs) && (id == self_id), null),
            new ChAct(id > self_id, become_passive),
            new ChAct(id < self_id, null)
          }
        )
      ) // Choice body :
      {
        case 0:
          try {
            become_leader.send();
          } catch(SyncException e) {}
          nextBehavEq = Leadership;
          actualPars = null;
          break;
        case 1:
          internal_Process.new_identity();
          nextBehavEq = Start;
          actualPars = new Object[] {Sync.random.d_uniform(0,
                                                       n_ids - 1)};
           break;
```

```
           case 2:
            try {
              become_passive.send();
            } catch(SyncException e) {}
            try {
              send_message.send(id,
                                hop + 1);
             } catch(SyncException e) {}
            nextBehavEq = Passive_Message_Receiving;
            actualPars = null;
            break;
          case 3:
            internal_Process.purge_message();
            nextBehavEq = Active_Message_Receiving;
            actualPars = new Object[] {self_id};
            break;
          default:
            nextBehavEq = null;  // STOP
            actualPars = null;
        }
      }

    }; // end of behavioral equation Evaluation

  Leadership =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Leadership();
      }

      private void _Leadership() {
        internal_Process.announce_leadership();
        try {
          send_stop.send();
        } catch(SyncException e) {}
        try {
          receive_stop.receive();
        } catch(SyncException e) {}
        nextBehavEq = null;  // STOP
        actualPars = null;
      }

    }; // end of behavioral equation Leadership

  Passive_Message_Receiving =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Passive_Message_Receiving();
      }
```

```
      private void _Passive_Message_Receiving() {
        int id;
        int hop;
        Object[] inputPars;
        switch (
          ElemMeth.choice(
            new ChAct[] {
              new ChAct(true, receive_message),
              new ChAct(true, receive_stop)
            }
          )
        ) // Choice body :
        {
          case 0:
            try {
              inputPars = receive_message.receive();
              id = (int)inputPars[0];
              hop = (int)inputPars[1];
            } catch(SyncException e) {}
            try {
              send_message.send(id,
                                hop + 1);
            } catch(SyncException e) {}
            nextBehavEq = Passive_Message_Receiving;
            actualPars = null;
            break;
          case 1:
            try {
              receive_stop.receive();
            } catch(SyncException e) {}
            try {
              send_stop.send();
            } catch(SyncException e) {}
            nextBehavEq = null;  // STOP
            actualPars = null;
            break;
          default:
            nextBehavEq = null;  // STOP
            actualPars = null;
        }
      }

    }; // end of behavioral equation Passive_Message_Receiving

  }

  //--------- RUNNING ELEMENT [thread] ----------//
  Thread th_Process = null;

  public void start() {
```

```
    (th_Process = new Thread(this)).start();
  }

  public void join() throws InterruptedException {
    th_Process.join();
  }

  public void run() {
    internal_Process =
      new IAS_Process();
    nextBehavEq = Start;
    actualPars = new Object[] {Sync.random.d_uniform(0,
                                                    n_ids - 1)};
    while (nextBehavEq != null)
      nextBehavEq.behavEqCall();
  }

}


class Bounded_FIFO_Monitor {

  //-------------- DECLARING STUBS --------------//
  // No IAS declaration as there are
  // no internal actions

  //----------- DEFINING CONSTRUCTOR ------------//
  private boolean[] guard;
  private final static int _Queue         = 0,
                           _Split_1_Queue = 1;
  private final static int _deposit_message  = 0,
                           _receive_stop     = 1,
                           _withdraw_message = 2,
                           _send_stop        = 3;
  int[] id_arr;
  int[] hop_arr;
  int head_index;
  int size;

  int n_ids;
  int n_procs;
  int capacity;

  Bounded_FIFO_Monitor(int n_ids,
                       int n_procs,
                       int capacity) {
    this.n_ids = n_ids;
    this.n_procs = n_procs;
    this.capacity = capacity;
  }

  //------------ DEFINING BEHAVIOR ------------//
```

```
private void checkGuard(int guardIndex, boolean blocking)
               throws InterruptedException,
                      SemisyncInteractionException {
  if (blocking)
    while (!guard[guardIndex])
      wait();
  else
    if (!guard[guardIndex])
      throw new SemisyncInteractionException();
}

protected synchronized void Queue(int[] id_arr,
                                  int[] hop_arr,
                                  int head_index,
                                  int size) {
  this.id_arr = id_arr;
  this.hop_arr = hop_arr;
  this.head_index = head_index;
  this.size = size;
  guard = new boolean[] {size < capacity, true, size > 0, false};
  notifyAll();
}

protected synchronized void Split_1_Queue() {
  guard = new boolean[] {false, false, false, true};
  notifyAll();
}

public synchronized void deposit_message(boolean blocking,
                                         int rec_id,
                                         int rec_hop)
                           throws InterruptedException,
                                  SemisyncInteractionException {
  checkGuard(_deposit_message, blocking);
  this.id_arr[(head_index + size) % capacity] = rec_id;
  this.hop_arr[(head_index + size) % capacity] = rec_hop;
  Queue(this.id_arr,
        this.hop_arr,
        this.head_index,
        this.size + 1);
}

public synchronized void receive_stop(boolean blocking)
                           throws InterruptedException,
                                  SemisyncInteractionException {
  checkGuard(_receive_stop, blocking);
  Split_1_Queue();
}

public synchronized Object[] withdraw_message(boolean blocking)
                           throws InterruptedException,
```

```
                                   SemisyncInteractionException {
    int id_arr_read;
    int hop_arr_read;
    checkGuard(_withdraw_message, blocking);
    id_arr_read = this.id_arr[head_index];
    hop_arr_read = this.hop_arr[head_index];
    Queue(this.id_arr,
          this.hop_arr,
          (this.head_index + 1) % this.capacity,
          this.size - 1);
    return new Object[] {id_arr_read,
                         hop_arr_read};
  }

  public synchronized void send_stop(boolean blocking)
                           throws InterruptedException,
                                  SemisyncInteractionException {
    checkGuard(_send_stop, blocking);
    guard = new boolean[] {false, false, false, false}; // STOP
  }

  //------------- STARTING MONITOR --------------//
  public void startMonitor() {
    Queue(new int[] {0,
                     0,
                     0},
          new int[] {1,
                     1,
                     1},
          0,
          0);
  }
}


class Bounded_FIFO implements RunnableElem {

  //------------- DECLARING MONITOR -------------//
  Bounded_FIFO_Monitor core_monitor_Bounded_FIFO;

  //-------- INSTANTIATING INTERACTIONS ---------//
  public UniSyncReceiverMonitorPort deposit_message =
    new UniSyncReceiverMonitorPort(this) {
      public synchronized void send(Object[] inputPars)
                               throws InterruptedException,
                                      NotReadyPortException {
        try {
          core_monitor_Bounded_FIFO.deposit_message(isBlocking,
                                                    (int)inputPars[0],
                                                    (int)inputPars[1]);
        } catch(SemisyncInteractionException e) {
          throw new NotReadyPortException(e);
```

```
      }
    }
  };

  public UniSyncReceiverMonitorPort receive_stop =
    new UniSyncReceiverMonitorPort(this) {
      public synchronized void send()
                              throws InterruptedException,
                                     NotReadyPortException {
        try {
          core_monitor_Bounded_FIFO.receive_stop(isBlocking);
        } catch(SemisyncInteractionException e) {
          throw new NotReadyPortException(e);
        }
      }
    };

  public UniSyncSenderMonitorPort withdraw_message =
    new UniSyncSenderMonitorPort(this) {
      public synchronized Object[] receive()
                                  throws InterruptedException,
                                         NotReadyPortException {
        Object[] outputPars;
        try {
          outputPars = core_monitor_Bounded_FIFO.withdraw_message(isBlocking);
        } catch(SemisyncInteractionException e) {
          throw new NotReadyPortException(e);
        }
        return outputPars;
      }
    };

  public UniSyncSenderMonitorPort send_stop =
    new UniSyncSenderMonitorPort(this) {
      public synchronized Object[] receive()
                                  throws InterruptedException,
                                         NotReadyPortException {
        try {
          core_monitor_Bounded_FIFO.send_stop(isBlocking);
        } catch(SemisyncInteractionException e) {
          throw new NotReadyPortException(e);
        }
        return null;
      }
    };

  //---------- DEFINING CONSTRUCTOR -----------//
  Bounded_FIFO(int n_ids,
               int n_procs,
               int capacity) {
    core_monitor_Bounded_FIFO = new Bounded_FIFO_Monitor(n_ids,
```

```
                                                        n_procs,
                                                        capacity);
  }

  //--------- RUNNING ELEMENT [monitor] ---------//
  public void start() {
    run();
  }

 public void join() throws InterruptedException { }

 public void run() {
   core_monitor_Bounded_FIFO.startMonitor();
 }
}


class Supervisor implements RunnableElem {

  //- DECLARING BEHAVIORAL EQUATIONS INTERFACES -//
  interface BehavioralEquationInterface { void behavEqCall(); }
  BehavioralEquationInterface Waiting_for_Leader,
                              Waiting_for_Processes;
  BehavioralEquationInterface nextBehavEq;
  Object[] actualPars;

  //-------- INSTANTIATING INTERACTIONS ---------//
  OrSyncReceiverPort receive_leader_notification =
    new OrSyncReceiverPort(this);
  OrSyncReceiverPort receive_passive_process_notification =
    new OrSyncReceiverPort(this);

  UniSyncSenderPort success =
    new UniSyncSenderPort(this);
  UniSyncSenderPort wrong_state_no_leader =
    new UniSyncSenderPort(this);
  UniSyncSenderPort wrong_state_too_many_leaders =
    new UniSyncSenderPort(this);

  //-------------- DECLARING STUBS --------------//
  // No IAS declaration as there are
  // no internal actions.
  EHS_Supervisor exception_Supervisor;

  //----------- DEFINING CONSTRUCTOR ------------//
  protected int n_procs;

  Supervisor(int n_procs) {
    this.n_procs = n_procs;
    defineBehavEquations();
  }
```

```
//------------- DEFINING BEHAVIOR -------------//

void defineBehavEquations() {

  Waiting_for_Leader =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Waiting_for_Leader((int)actualPars[0]);
      }

      private void _Waiting_for_Leader(int active_processes) {
        switch (
          ElemMeth.choice(
            new ChAct[] {
              new ChAct(active_processes > 0, receive_passive_process_notification),
              new ChAct(active_processes > 0, receive_leader_notification),
              new ChAct(active_processes == 0, wrong_state_no_leader)
            }
          )
        ) // Choice body :
        {
          case 0:
            try {
              receive_passive_process_notification.receive();
            } catch(SyncException e) {}
            nextBehavEq = Waiting_for_Leader;
            actualPars = new Object[] {active_processes - 1};
            break;
          case 1:
            try {
              receive_leader_notification.receive();
            } catch(SyncException e) {}
            nextBehavEq = Waiting_for_Processes;
            actualPars = new Object[] {active_processes - 1};
            break;
          case 2:
            try {
              wrong_state_no_leader.send();
            } catch(UnattachedPortException e) {
              exception_Supervisor.wrong_state_no_leader();
            }
            nextBehavEq = null;  // STOP
            actualPars = null;
            break;
          default:
            nextBehavEq = null;  // STOP
            actualPars = null;
        }
      }
```

```
    }; // end of behavioral equation Waiting_for_Leader

  Waiting_for_Processes =
    new BehavioralEquationInterface() {

      public void behavEqCall() {
        _Waiting_for_Processes((int)actualPars[0]);
      }

      private void _Waiting_for_Processes(int active_processes) {
        switch (
          ElemMeth.choice(
            new ChAct[] {
              new ChAct(active_processes > 0, receive_passive_process_notification),
              new ChAct(active_processes > 0, receive_leader_notification),
              new ChAct(active_processes == 0, success)
            }
          )
        ) // Choice body :
        {
          case 0:
            try {
              receive_passive_process_notification.receive();
            } catch(SyncException e) {}
            nextBehavEq = Waiting_for_Processes;
            actualPars = new Object[] {active_processes - 1};
            break;
          case 1:
            try {
              receive_leader_notification.receive();
            } catch(SyncException e) {}
            try {
              wrong_state_too_many_leaders.send();
            } catch(UnattachedPortException e) {
              exception_Supervisor.wrong_state_too_many_leaders();
            }
            nextBehavEq = null;  // STOP
            actualPars = null;
            break;
          case 2:
            try {
              success.send();
            } catch(UnattachedPortException e) {
              exception_Supervisor.success();
            }
            nextBehavEq = null;  // STOP
            actualPars = null;
            break;
          default:
            nextBehavEq = null;  // STOP
            actualPars = null;
```

```
            }
          }

        }; // end of behavioral equation Waiting_for_Processes

    }

    //--------- RUNNING ELEMENT [thread] ----------//
    Thread th_Supervisor = null;

    public void start() {
      (th_Supervisor = new Thread(this)).start();
    }

    public void join() throws InterruptedException {
      th_Supervisor.join();
    }

    public void run() {
      exception_Supervisor =
        new EHS_Supervisor();
      nextBehavEq = Waiting_for_Leader;
      actualPars = new Object[] {n_procs};
      while (nextBehavEq != null)
        nextBehavEq.behavEqCall();
    }
  }



  class IAS_Process {

    IAS_Process() {
      // FILL IN THE CONSTRUCTOR BODY IF NEEDED
    }

    void count_evaluation_steps() {
      // FILL IN THE METHOD BODY
    }

    void new_identity() {
      // FILL IN THE METHOD BODY
    }

    void purge_message() {
      // FILL IN THE METHOD BODY
    }

    void announce_leadership() {
      // FILL IN THE METHOD BODY
    }
  }
```

```
class EHS_Supervisor {

  EHS_Supervisor() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void wrong_state_no_leader() {
    // FILL IN THE METHOD BODY
  }

  void wrong_state_too_many_leaders() {
    // FILL IN THE METHOD BODY
  }

  void success() {
    // FILL IN THE METHOD BODY
  }
}
```

Finally, note that the random number generator `d_uniform()` used in the Æmilia description of the AET `Process` is translated into a call of a function having the same name (and the same purpose, obviously), provided by the package `Sync`.

8.2.6 Results on the Implementation Side

Since the analysis of the leader election system at the process algebraic description level refers to probabilistic properties, it is not guaranteed that the code generated by the translator PADL2Java preserves the same properties correctly. In order to empirically assess the correct termination of the algorithm and to analyze the probability of electing a leader as a function of discrete-time steps, the generated code has been used for producing traces of its own execution.

For checking the correct termination of the algorithm on the implementation side, the stub `EHS_Supervisor` has been filled in as follows:

```
class EHS_Supervisor {

  EHS_Supervisor() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }

  void wrong_state_no_leader() {
    System.err.println("ERROR: no leader has been elected");
  }
```

```
  void wrong_state_too_many_leaders() {
    System.err.println("ERROR: too many leader have been elected");
  }

  void success() {
    System.err.println("SUCCESS: just one leader has been elected");
  }
}
```

while for the analysis of the probability of electing a leader, the stub `IAS_Process` has been filled in as follows:

```
class IAS_Process {

  int step_counter;

  IAS_Process() {
    step_counter = 0;
  }

  void count_evaluation_steps() {
    step_counter++;
  }

  void new_identity() {
    // FILL IN THE METHOD BODY
  }

  void purge_message() {
    // FILL IN THE METHOD BODY
  }

  void announce_leadership() {
    System.out.println("-> " + step_counter + " steps before becoming a leader");
  }
}
```

Moreover, a program has been written for running the class `Leader_Election_B` several thousand of times for different numbers of processes and of identities. The same program, then, analyzes the traces that in the previous phase have been redirected from the standard `err` and from the standard `out` into two different files.

The first trace contained:

```
SUCCESS: just one leader has been elected
...
```

```
SUCCESS: just one leader has been elected
...
SUCCESS: just one leader has been elected
```

without any error message. This empirically confirms the correct termination of the implemented leader election system.

The second trace contained several sections related to different experiments with varying size of the problem – i.e., number of processes and identities:

```
...
4 Processes and 4 Identities
-> 12 steps before becoming a leader
-> 18 steps before becoming a leader
-> 15 steps before becoming a leader
...
```

From this trace, an average value for any section has been calculated. These values, compared with the values estimated through the transient probability distribution calculator of TwoTowers, empirically confirm the trend of the probability curves illustrated in Fig. 8.3.

Chapter 9

Conclusion

9.1 Summary of Results

In this thesis we have presented an approach for automatically generating multithreaded Java programs from process algebraic architectural descriptions of concurrent software systems. The approach is divided into three phases.

In the first phase, a package called `Sync` has been developed to guarantee the correct thread coordination management according to a general communication model. The definition and the use of this package is inspired by architectural concepts.

In the second phase, the translation of the process-algebraically-specified behavior of the software units has been addressed. Due to the different level of abstraction of a programming language and an architectural description language, only a part of the translation can be automated, while for the rest – internal actions and interaction exceptions – a set of guidelines has been provided in order to assist the developer.

In the third phase, the suitability of synthesizing monitors rather than threads has been assessed. Specific constraints have been identified that must be satisfied by the process algebraic description of a software unit in order for it to be easily translated into a monitor.

Beyond the mere automatic code generation, an important result of this approach is that it guarantees – under certain conditions – that the properties proved at the architectural level are preserved at the code level. These conditions

only concern the code that has to be manually written by the software developer for implementing the internal actions and the interaction exceptions. Thus, following the guidelines provided in Sect. 5.7 completely ensures the preservation of the architectural properties.

As far as limitations of the proposed approach are concerned, it is worth reminding that the application of the third phase is subject to the constraints defined in Sect. 6.1. Since the architectural element types that do not satisfy the constraints can always be generated as threads instead of monitors, the monitor constraints are not a real limitation for the application of the whole approach. However, some of the constraints may appear quite strong. Actually, the main reason why we have provided sufficient but not (always) necessary conditions for guaranteeing a correct monitor generation, is that we have preferred to develop constraints that can be easily checked in architectural topologies using local criteria. This issue, together with the correct application of the guidelines for manually writing code at the end of the second and of the third phase, will be discussed in Sect. 9.3 as a future work.

The whole approach described in this thesis has been implemented in a translator called PADL2Java that offers three options: package generation, program generation, and applet generation. The various options and the structure of the generated code have been illustrated through the architectural description of an audio processing system. PADL2Java has then been integrated in TwoTowers, an architecture-centric verification tool. Hence, this integrated toolset can provide an effective support to bridge the gap between software modeling and software implementation. Two case studies have been presented in order to illustrate the usage of the translator PADL2Java and the advantages deriving from its integration in TwoTowers.

A further result of this thesis is that it has contributed to the enhancement of the expressiveness of the architectural description language PADL. First, the communication model provided by the language has been extended thanks to the introduction of asynchronous and semi-synchronous communications among architectural elements. Second, the generic object, i.e., `object(`◁*type id*▷`)`, has been introduced as a new data type of PADL in order to increase the effectiveness of the code generated by the translator PADL2Java.

9.2 Related Work

Concerning related work, it is worth to recall [52] where the problem of developing concurrent software systems is tackled by applying a systematic methodology based on a PA. More precisely, concurrent Java programs are modeled and designed using a simple process algebra notation called FSP. The verification tool LTSA is used for analyzing properties of the models.

The process algebra-based language LOTOS [16] is used instead by the toolbox CADP (CÆSAR/ALDEBARAN Development Package [28, 34]) for the design of communication protocols and distributed systems. The toolbox is able to produce C code for rapid prototyping and testing purposes.

Strictly concerning the automatic code/program generation from architectural descriptions, first of all we mention ArchJava [3]. This is an extension of Java aiming at the unification of software architecture with implementation, in order to ensure that the implementation conforms to the architectural specification with respect to communication integrity. According to this property, each component in the implementation may only communicate directly with the components to which it is connected in the architecture.

Our approach differs from ArchJava in several ways. First, it does not extend Java, but generates Java code from process algebraic architectural descriptions. In our approach the developer is then required to fill in some stubs to complete the code for the behavior of the threads, thus giving a certain degree of flexibility. The price to be paid is that the guidelines may be violated, whereas a similar situation is not possible in ArchJava. Second, our approach adopts a richer communication model, implemented and transparently made available through package `Sync`. This guarantees a property even stronger than communication integrity: implementation threads directly communicate only with the threads they are connected to in the architectural description in the way prescribed by the architectural description itself with respect to communication synchronicity (synchronous, semi-synchronous, and asynchronous) and communication multiplicity (uni, and, or). Third, since it keeps the architectural description language separated from the implementation language, our approach provides a higher-level support than ArchJava for the

preservation of behavioral properties. On the other hand, the strong integration between architecture and implementation endows ArchJava with useful dynamic capabilities, like run-time creation of components – although communication integrity places restrictions on the way in which their instances can be used – and connections among them.

Then we mention C2SADEL [54]. This is an architectural description language tied to the C2 style, which combines the usual architectural concepts with type theory. Type checking is used to analyze the architectural descriptions for consistency by unifying corresponding operations required and provided by different components. Moreover, Java code can be automatically generated from C2SADEL descriptions. Since type checking is a static analysis technique, while the architectural features on which we focus are behavioral and concerned with a rich communication model, we believe that our approach can guarantee the preservation of more complex properties than C2SADEL.

We also mention [38], where an approach is proposed for automatically generating Java code from SDL specifications of telecommunication systems. Although the target of this approach and ours is similar, the framework in which the two approaches are considered is quite different. In fact, while our approach deals with the correct thread coordination within a single Java program, the other approach aims at the generation of distributed Java code whose components are coordinated via CORBA.

If we restrict ourselves to monitor generation, we have [67], where a tool equipped with a model checker automatically generates Java monitor classes from monitor descriptions written in Action Language. The correctness of the synchronization and of the behavior of a generated Java monitor is guaranteed by construction, independently of the context of the monitor description. Unlike our approach, this approach requires that the monitor description conforms a priori to a specific monitor template.

Finally we have [27], where implementations of synchronization policies are generated in Java through synchronized methods and lock objects. While in the previously described approaches the generated Java monitors are obtained from formal specifications and are correct by construction, in this approach the code is generated from critical regions delimited by the developer with

high-level synchronization directives and the correctness of the implemented synchronization policies is verified at the code level via model checking.

9.3 Future Research

Concerning future work, on the theoretical side we plan to conduct further investigations on the monitor constraints, in particular with respect to specific contexts. For instance, one constraint requires that a candidate monitor has no other monitor attached to it, which is a sufficient condition for avoiding monitor interferences. However, monitors that invoke methods of other monitors are often employed in correct concurrent programs. Thus, our constraint does not result in a necessary condition. If we knew additional information about the components interacting with the monitor, we could obtain a set of necessary conditions that would make it possible to weaken the constraints.

On the methodological side we plan to investigate the combination of our approach with some of those discussed in Sect. 9.2. In particular, we would like to investigate the applicability of our approach to C2SADEL, in order to take advantage of both type checking and behavioral analysis from the architectural level to the code level. Similarly, we would like to experiment our approach with ArchJava – by generating ArchJava code instead of Java code – in order to exploit the complementary strengths of the two approaches.

Finally, on the applicative side we would like to further integrate our toolset with software model checking tools, like Bandera [24], and to define specific rules for static analysis tools, like Eclipse TPTP [39]. The reason is that the preservation at the code level of the properties proved at the architectural level is guaranteed only if (the underlying platform is correct and) the designer follows the guidelines provided in Sect. 5.7 when filling in the stubs for internal actions and interaction exceptions. Having a software model checker available within TwoTowers would strengthen our approach as it would permit the verification of the overall system after the intervention of the software designer, when customized static analysis tools will have been suitably applied for guiding the intervention.

References

[1] A. Aldini and M. Bernardo, *"On the Usability of Process Algebra: An Architectural View"*, in Theoretical Computer Science 335:281-329, 2005.

[2] A. Aldini and M. Bernardo, *"Mixing Logics and Rewards for the Component-Oriented Specification of Performance Measures"*, in Theoretical Computer Science 382:3-23, 2007.

[3] J. Aldrich, C. Chambers, and D. Notkin, *"ArchJava: Connecting Software Architecture to Implementation"*, in Proc. of the *24th Int. Conf. on Software Engineering (ICSE 2002)*, IEEE-CS Press, pp. 187-197, Orlando (Florida, USA), 2002.

[4] R. Allen and D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213-249, 1997.

[5] D. Angluin, *"Local and Global Properties in Networks of Processors"*, in Proc. of the *12th ACM Int. Symp. on Theory of Computing*, ACM Press, pp. 82-93, Los Angeles (California, USA), 1980.

[6] S. Balsamo, M. Bernardo, and M. Simeoni, *"Performance Evaluation at the Software Architecture Level"*, in [15]:207-258.

[7] J.A. Bergstra, A. Ponse, and S.A. Smolka (eds.), *"Handbook of Process Algebra"*, Elsevier, 2001.

[8] M. Bernardo, *"Symbolic Semantic Rules for Producing Compact STGLA from Value Passing Process Descriptions"*, in ACM Trans. on Computational Logic 5:436-469, 2004.

[9] M. Bernardo, *"TwoTowers 5.1 User Manual"*, `http://www.sti.uniurb.it/bernardo/twotowers`, 2006.

[10] M. Bernardo and E. Bontà, *"Generating Well-Synchronized Multithreaded Programs from Software Architecture Descriptions"*, in Proc. of the *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA 2004)*, IEEE-CS Press, pp. 167-176, Oslo (Norway), 2004.

[11] M. Bernardo and E. Bontà, *"Preserving Architectural Properties in Multithreaded Code Generation"*, in Proc. of the *7th Int. Conf. on Coordination Models and Languages (COORDINATION 2005)*, LNCS 3454:188-203, Namur (Belgium), 2005.

[12] M. Bernardo and E. Bontà, *"Non-Synchronous Communications in Process Algebraic Architectural Description Languages"*, submitted for publication, 2008.

[13] M. Bernardo, P. Ciancarini, and L. Donatiello, *"Architecting Families of Software Systems with Process Algebras"*, in ACM Trans. on Software Engineering and Methodology 11:386-426, 2002.

[14] M. Bernardo, L. Donatiello, and P. Ciancarini, *"Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language"*, in *Performance Evaluation of Complex Systems: Techniques and Tools*, LNCS 2459:236-260, 2002.

[15] M. Bernardo and P. Inverardi (eds.), *"Formal Methods for Software Architectures"*, LNCS 2804, 2003.

[16] T. Bolognesi and E. Brinksma, *"Introduction to the ISO Specification Language LOTOS"*, in Computer Networks and ISDN Systems 14:25-59, 1987.

[17] E. Bontà, M. Bernardo, J. Magee, and J. Kramer, *"Synthesizing Concurrency Control Components from Process Algebraic Specifications"*, in Proc. of the *8th Int. Conf. on Coordination Models and Languages (COORDINATION 2006)*, LNCS 4038:28-43, Bologna (Italy), 2006.

[18] F.P. Brooks, *"The Mythical Man-Month"*, Addison Wesley, 1975.

[19] F. Budinsky, M. Finnie, P. Yu, and J. Vlissides, *"Automatic Code Generation from Design Patterns"*, IBM Systems Journal 35:151-171, 1996.

[20] C. Canal, E. Pimentel, and J.M. Troya, *"Compatibility and Inheritance in Software Architectures"*, in Science of Computer Programming 41:105-138, 2001.

[21] R. Cavada, A. Cimatti, E. Olivetti, M. Pistore, and M. Roveri, *"NuSMV 2.1 User Manual"*, `http://nusmv.irst.itc.it/`, 2002.

[22] E.M. Clarke, O. Grumberg, and D.A. Peled, *"Model Checking"*, MIT Press, 1999.

[23] W.R. Cleaveland and O. Sokolsky, *"Equivalence and Preorder Checking for Finite-State Systems"*, in [7]:391-424.

[24] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Păsăreanu, Robby, and H. Zheng, *"Bandera: Extracting Finite-state Models from Java Source Code"*, in Proc. of the *22nd Int. Conf. on Software Engineering (ICSE 2000)*, IEEE-CS Press, pp. 439-448, Limerick (Ireland), 2000.

[25] K. Czarnecki and S. Helsen, *"Classification of Model Transformation Approaches"*, in Proc. of the *2nd OOSPLA Workshop on Generative Techniques in the Context of MDA (OOSPLA 2003)*, Anaheim (California, USA), 2003.

[26] K. Czarnecki and S. Helsen, *"Feature-Based Survey of Model Transformation Approaches"*, IBM Systems Journal 45:621-645, 2006.

[27] X. Deng, M.B. Dwyer, J. Hatcliff, and M. Mizuno, *"Invariant-Based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs"*, in Proc. of the *24th Int. Conf. on Software Engineering (ICSE 2002)*, ACM Press, pp. 442-452, Orlando (Florida, USA), 2002.

[28] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis, *"A Toolbox for the Verification of LOTOS Programs"*, in Proc. of the *14th Int. Conf. on Software Engineering (ICSE 1992)*, IEEE-CS Press, pp.246-259, Melbourne (Australia), 1992.

[29] M. Fowler, *"UML Distilled: A Brief Guide to the Standard Object Modeling Language"*, Addison-Wesley, 2003.

[30] R.B. France, S. Ghosh, and T. Dinh-Trong, *"Model-Driven Development Using UML 2.0: Promises and Pitfalls"*, in [42]:59-66.

[31] R. Focardi and R. Gorrieri, *"A Classification of Security Properties"*, in Journal of Computer Security 3:5-33, 1995.

[32] W. Fokkink and J. Pang, *"Simplifying Itai-Rodeh Leader Election for Anonymous Rings"*, in Proc. of the *4th Workshop on Automated Verification of Critical Systems (AVoCS 2004)*, London (UK), ENTCS 128:53-68, 2005.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley, 1995.

[34] H. Garavel, *"OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing"*, in Proc. of the *1st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*, LNCS 1384:68-84, Lisbon (Portugal), 1998.

[35] D. Garlan, *"Formal Modeling and Analysis of Software Architectures: Components, Connectors, and Events"*, in [15]:1-24.

[36] D. Garlan, R. Allen, and J. Ockerbloom, *"Exploiting Style in Architectural Design Environment"*, in Proc. of the *2nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE 1994)*, ACM Press, pp. 175-188, New Orleans (Lousiana, USA),1994.

[37] R.J. van Glabbeek, *"The Linear Time - Branching Time Spectrum I"*, in [7]:3-99.

[38] R. Guimarães and W. Borelli, *"An Automatic Java Code Generation Tool for Telecom Distributed Systems"*, in Proc. of the *Int. Conf. on Software, Telecommunications and Computer Networks (SOFTCOM 2002)*, IEEE-CS Press, pp. 1-6, Split (Croatia), 2002.

[39] S. Gütz and O. Marquez, *"TPTP Static Analysis Tutorial"*, `http://www.eclipse.org/tptp/`, 2006.

[40] C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985.

[41] R.A. Howard, *"Dynamic Probabilistic Systems"*, John Wiley & Sons, 1971.

[42] IEEE Computer Magazine 39(2), 2006.

[43] P. Inverardi, A.L. Wolf, and D. Yankelevich, *"Static Checking of System Behaviors Using Derived Component Assumptions"*, in ACM Trans. on Software Engineering and Methodology 9:239-272, 2000.

[44] A. Itai and M. Rodeh, *"Symmetry Breaking in Distributed Networks"*, in Information and Computation 88:60-87, 1990.

[45] R. Jain, *"The Art of Computer Systems Performance Analysis"*, John Wiley & Sons, 1991.

[46] JavaCC, *"Java Compiler Compiler (tm) - The Java Parser Generator"*, `https://javacc.dev.java.net/`, 2007.

[47] F. Jouault and I. Kurtev, *"On the Architectural Alignment of ATL and QVT"*, in Proc. of the *21st ACM Symp. on Applied Computing (SAC 2006)*, ACM Press, pp. 1188-1195, Dijon (France), 2006.

[48] L. Kleinrock, *"Queueing Systems"*, Wiley, 1975.

[49] M.Z. Kwiatkowska, G. Norman, and D. Parker, *"PRISM: Probabilistic Symbolic Model Checker"*, in Proc. of the *12th Int. Conf. on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance Tools 2002)*, LNCS 2324:200-204, London (UK), 2002.

[50] G. Lenz and C. Wienands, *"Practical Software Factories in .NET"*, Apress L.P., 2006.

[51] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *"Specifying Distributed Software Architectures"*, in Proc. of the *5th European Software Engineering Conf. (ESEC 1995)*, LNCS 989:137-153, Barcelona (Spain), 1995.

[52] J. Magee and J. Kramer, *"Concurrency: State Models & Java Programs"*, Wiley, 1999.

[53] MDA, *"OMG – Model-Driven Architecture"*, `http://www.omg.org/mda/`, 2007.

[54] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, *"A Language and Environment for Architecture-Based Software Development and Evolution"*, in Proc. of the *21st Int. Conf. on Software Engineering (ICSE 1999)*, IEEE-CS Press, pp. 44-53, Los Angeles (California, USA), 1999.

[55] N. Medvidovic and R.N. Taylor, *"A Classification and Comparison Framework for Software Architecture Description Languages"*, in IEEE Trans. on Software Engineering 26:70-93, 2000.

[56] R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989.

[57] OMG, *"The Object Management Group (OMG)"*, `http://www.omg.org`, 2007.

[58] A. Poggi and G. Rimassa, *"An Efficient and Flexible C++ Library for Concurrent Programming"*, in Software Practice and Experience 28:1437-1463, 1998.

[59] QVT, *"OMG – MOF QVT Final Adopted Specification"*, `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

[60] D.C. Schmidt, *"Model-Driven Engineering"*, in [42]:25-31.

[61] M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.

[62] SmartQVT, *"SmartQVT – A QVT implementation"*, `http://smartqvt.elibel.tm.fr/`, 2007.

[63] W.J. Stewart, *"Introduction to the Numerical Solution of Markov Chains"*, Princeton University Press, 1994.

[64] UML, *"OMG – Unified Modeling Language"*, `http://www.uml.org/`, 2007.

[65] D. Varro, G. Varro, and A. Pataricza, *"Designing the Automatic Transformation of Visual Languages"*, Science of Computer Programming 44:205-227, 2002.

[66] P.D. Welch, *"The Statistical Analysis of Simulation Results"*, in *"Computer Performance Modeling Handbook"*, Academic Press, pp. 267-329, 1983.

[67] T. Yavuz-Kahveci and T. Bultan, *"Specification, Verification, and Synthesis of Concurrency Control Components"*, in Proc. of the *9th ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA 2002)*, ACM Press, pp. 169-179, Rome (Italy), 2002.