ALMA MATER STUDIORUM — UNIVERSITÀ DI BOLOGNA

DEIS - Department of Electronics, Computer Science and Systems
PhD Course in Electronics, Computer Science and Telecommunications

Cycle XXV
Concourse Sector: 09/H1
Disciplinary Sector: ING-INF/05

ENGINEERING AGENT-ORIENTED TECHNOLOGIES AND PROGRAMMING LANGUAGES FOR COMPUTER PROGRAMMING AND SOFTWARE DEVELOPMENT

Candidate

Dott. ANDREA SANTI

Ph.D Coordinator:

Chiar.mo Prof. Ing. ALESSANDRO VANELLI CORALLI

Supervisor:

Chiar.mo Prof. Ing. ALESSANDRO RICCI

Tutor:

Chiar.mo Prof. Ing. ANTONIO NATALI

FINAL EXAMINATION YEAR 2013

Contents

Abstract

Mainstream hardware is becoming parallel, heterogeneous, and distributed on every desk, every home and in every pocket. As a consequence, in the last years software is having a fundamental and epochal turn toward concurrency, decentralization, distribution, interaction which is pushed by the evolution of hardware architectures and the growing of network availability. This is having a strong impact on everyday programming: concurrent and distributed programming – which are challenging – are no longer a matter of specific application domains, but are becoming more and more issues to take into the account in mainstream programming and related languages.

This calls for introducing further abstraction layers on top of those provided by classical mainstream programming paradigms, to tackle more effectively the new complexities that developers have to face in everyday programming.

Given the big tide on concurrency and the well-known difficulties and problems that affect multi-threaded programming, a convergence it is recognizable in the mainstream toward the adoption of the actor model and actor-related approaches as a mean to unite object-oriented programming and concurrency. Nevertheless, we argue that the actor paradigm can only be considered a good starting point to provide a more comprehensive response to such a fundamental and radical change in software development.

Accordingly, the main objective of this thesis is to propose *Agent-Oriented Programming* (AOP) as a high-level general purpose programming paradigm, natural evolution of actors and finally objects, introducing a further level of human-inspired concepts for programming software systems, meant to simplify the design and programming of concurrent, distributed, reactive and interactive programs.

To this end, in the dissertation first we construct the required background by studying the state-of-the-art of both actor-oriented and agent-oriented programming, and then we focus on the engineering of *integrated* programming technologies for developing agent-based systems in their classical application domains: artificial intelligence (AI) and distributed artificial intelligence (DAI).

Then, we shift the perspective moving from the development of intelligent software systems, toward general purpose computing and software development. Using the knowledge and expertise maturated during the phase of background construction, we introduce a general-purpose programming language, named simpAL, which founds its roots on general principles and practices of software development, and at the same time provides an agent-oriented level of abstraction for the engineering of general purpose software systems. Practical experience with relevant case studies suggests that the proposed programming approach is effective for tackling some of the main complexities that modern software development introduces.

Keywords: *concurrent programming*, *distributed programming*, *event-driven programming*, *programming languages*, *agents*, *actors*.

1 Introduction

The fundamental turn of software toward concurrency, decentralization, distribution, interaction that we are witnessing in recent years – pushed by the evolution of hardware architectures (e.g., multi-core, many-core, mobile platforms) and network availability – is having a strong impact on everyday programming. Mainstream hardware is becoming parallel, heterogeneous, and distributed on every desk, every home and in every pocket. 2011 in particular was special: it's when we completed the transition to parallel computing in all mainstream form factors, with the arrival of multi-core tablets and smart-phones. These changes are permanent, and so will permanently affect the way we have to write programs. There is no going back [Suta].

As stated by Herb Sutter, *"The free lunch is over. Now welcome to the jungle."* [Sutb, Suta][1]: concurrent and distributed programming – which are challenging – are no more a matter of only specific application domains (e.g., high-performance computing), but are more and more issues to take into the account in mainstream programming and related languages. This caused a big tide on concurrency, and the consequent development of libraries, frameworks and fine-grained mechanisms on top of existing languages specifically tailored to harness the power of multi-core, many-core and cloud-core architectures in our programs, sometimes also hiding concurrency, parallelism and decentralization as much as possible (e.g., automatic parallelization). So, most of the research nowadays is focused on how to get a performance boost from the exploitation of such powerful hardware, while sticking with current well-established paradigms—i.e., Object-Oriented Programming (OOP) in particular. This is evident by briefly analyzing the contributions appeared in the last years in the main tracks of reference research conferences on programming languages such as OOPSLA/SPLASH [LD12, LF11].

Besides this important viewpoint, we argue that *the free lunch is over* is also a matter of conceptual modeling and abstraction, not only performance, that is to think about introducing and experimenting in everyday programming novel programming paradigms, modeling and implementing in a more natural and effective way than mainstream approaches concurrent, reactive, adaptive and distributed software systems. Or, by using Sutter and Larus' words in [SL05],

[1] The first part of the quote *"The free lunch is over"* refers to a famous 2005 article appeared on Dr Dobbs by Herb Sutter, which led to a similar articles on the theme [SL05]. The *"Now welcome to the jungle"* part has been added later on in a second online article by Sutter – which is the conceptual continuation of the first one – named *"Welcome to the Jungle"* [Sutb].

approaches introducing effective programming abstractions that would *"help build concurrent programs, just as object-oriented abstractions help build large component-based programs"*.

This calls for going back to research, e.g., on Object-Oriented Concurrent Programming developed in the eighties and nineties in particular [SW87, BGL98], when we were still *free lunching* so to say, and the context of such research was not really mainstream programming but high-performance computing and parallel programming for super-computers. The *actor computing model* [KA11, AMST97] and similar approaches (e.g., active objects [YT87]) were among the main results and target of investigations, along with the development of several new programming languages and frameworks based on the actor idea. Given the big tide on concurrency and the well-known difficulties and problems that affect multi-threaded programming, actors seem to get a momentum today, as far as one considers their injections in terms of libraries and frameworks developed on top of existing programming languages [KSA09] (e.g., Scala Actors[HS12], akka [Gup12], ActorFoundry [KA]), or directly supported by new languages (e.g., DART [Incb], with isolates). Indeed, the injection is not without problems given some basic fundamental incompatibilities at the conceptual level among the programming models [HO08] that become particularly evident when trying to integrate different concurrency models and programming styles—i.e., thread-based, event-based, actor-based.

Besides these integration problems, actor programming can be conceived nowadays as an extension or evolution of OOP with decentralization and encapsulation of control and asynchronous message passing [KA11, AMST97]. So, it is apparent that actors do not introduce a huge change in the main abstractions featured by OOP, which is strongly based too on the concept of message passing, as often reminded by Alan Kay [Kay96, Kay69]. Conversely, one can argue that important features that are nowadays a well-defined part of OOP (e.g. inheritance, sub-typing, polymorphism, etc.) have not found a corresponding easy settlement in the context of actors [MY93]. So, getting back to the free lunch call, we argue that actors can be seen as a strong foundation layer, but finally giving solely asynchronous message passing as unique concept [Mit02], thus not significantly enhancing the set of abstractions that we can use to simplify – in general – the development of software systems.

In that perspective, in this dissertation we propose *Agent-Oriented Programming* (AOP) as a high-level general purpose programming paradigm, natural evolution of actors and finally objects, specifically introducing a set of first-class abstractions that are meant to simplify the design and programming of concurrent, distributed, reactive and interactive programs [RS11a].

Actually, the idea of Agent-Oriented Programming is not new. The first paper about AOP is dated 1993 [Sho93], and since then many Agent Programming Languages (APLs) and languages for programming Multi-Agent Systems (MASs) have been proposed in literature [BDDEFS11, BDDFS05a, BDEFSD09]. The original objective of AOP as introduced in [Sho93] was the definition of a post object-oriented programming paradigm for developing complex applications, providing higher-level features compared to existing paradigms. In spite of this objective, it is apparent that agent-oriented programming has not had a significant impact on mainstream research in programming languages and software development, so far. We believe that this depends on the fact that – despite few exceptions – most of the efforts and

emphasis have been put on the study of architectures, theories and languages to program agents and agent-based programs taking Artificial Intelligence (AI) and Distributed AI (DAI) as the reference contexts.

Given the (D)AI background, current APLs in the state-of-the-art focus on features that are especially important in *that* context – e.g., programming using mentalistic notions, basic reasoning capabilities, etc. – and are essentially exploited to develop intelligent software systems, exhibiting some kind of individual or collective intelligent behavior [Jen01, Woo09, RN09]. So, current APLs have not been intentionally designed with the aim of being adopted as general-purpose programming approaches in mainstream software development—this is clear also by checking the number of publications about agent-oriented programming in journals or conferences about programming languages and software development. This is exactly the broad objective of this thesis work, so as to explore and develop agent-oriented programming targeting general-purpose computing, taking as a reference context existing programming paradigms and, in particular, the general principles and practices of software development. In particular we are interested in:

- Identifying the essential concepts and features of the paradigm, and investigating how such features could be effective in particular for tackling complexities of modern software development.

- Investigating how well-known features and mechanisms that have been introduced and developed in modern programming languages to support programming in the large and good programing (but in the sequential case), could be injected and eventually re-framed by adopting an agent-oriented level of abstraction. Main examples are typing, reuse, inheritance and polymorphism.

- Investigating if and how the new abstraction level raised by agent-oriented programming impacts on the design of *tools* supporting the development and deployment process, from front-ends to debuggers and runtime infrastructures.

The method we chose to explore these points is the design and development of a new agent-oriented programming language called simpAL, and its related ecosystems, composed by an Integrated Development Environment (IDE) and a distributed runtime infrastructure.

For the achievement of the aforementioned research goals, a key point has been the construction of a solid background in the context of the classical research on agent and multi-agent systems, with the aim to take inspiration from the main models, abstractions and programming languages that have been introduced in the state-of-the-art. In particular, this background construction has purposefully been an *active process*—i.e., we have actively took part in the study, improvement and engineering of *integrated* agent-oriented technologies and programming frameworks for the development of intelligent software systems.

1.1 Contributions

This thesis work has produced several contributions that can be roughly divided in two main macro-groups. The first macro-group of contributions is related to the achievements obtained in the engineering of agent-oriented technologies for the (D)AI context. In detail these contributions are:

- The definition of an effective action and perception model (Section 4.1) specifically conceived to make BDI-based agents – i.e., agents rooted on the Belief Desire Intention model [RG$^+$95] (see Section 3.2.1) – work in endogenous environments (Section 3.3.2), shifting from the classical models available in the state-of-the-art. Besides being implemented and adopted in every agent-oriented technology that we contributed to realize during the background construction of this thesis (see below), the model is general, i.e. it can be possibly used in any BDI-based APL that needs to interact with endogenous environments.

- The realization of both the JaCa programming approach and development platform (Chapter 4), for developing MASs by synergistically exploiting both the agent and the environment programming dimensions.

- The realization of several extensions of the JaCa platform for exploiting its programming model in some of the most modern and relevant application domains. In details these platforms are: JaCa-Android, for programming smart mobile applications on top of the Android platform (Section 4.4); JaCa-Web, for programming Rich (client) Internet Applications (RIAs) (Section 4.6); and finally JaCa-WS (Section 4.5) for programming applications rooted on the Service Oriented Architecture (SOA).

- The concrete evaluation of both the JaCa platform and its various extensions through the development of interesting case studies, with the objective to investigate both the benefits and the limitations of the proposed programming approach (Section 4.4.2, Section 4.5.2, Section 4.6.2).

- We collaborated to the realization of JaCaMo (Chapter 5), which provides both a concrete programming approach and a related development platform/infrastructure for developing multi-agent systems, taking into the account three different agent-oriented programming dimensions in a synergistic manner, namely the agent, environment and the organization.

- We collaborated to the evaluation of the JaCaMo platform by studying its application in some real-world projects (Section 5.3). As in the case of the JaCa platform, the main objective of this concrete evaluation has been the investigation of the benefits and possible shortcomings of the JaCaMo agent-oriented programming approach.

The second macro-group of contributions instead is strictly related to the results obtained through the engineering and evaluation of simpAL and its ecosystem. In detail these contributions are:

- The simpAL language itself (Chapter 7), which founds its roots on general principles and practices of software development, and at the same time provides an agent-oriented level of abstraction for the engineering of general purpose software systems, possibly concurrent and distributed.
- The introduction of specific programming abstractions that ease the integration of autonomous and reactive behaviors when programming active entities that need to exhibit both thread-based and event-based forms of computations. This is a relevant problem in the context of concurrent programming, and we argue that simpAL gives the opportunity to solve this issue at the foundation level, in a quite seamless manner. This aspect is discussed in details in Section 7.3.1.
- The definition of a notion of type for the main first-class abstractions of the simpAL programming model. We argue that this is quite an important contribution, in particular for what concerns the definition of an explicit and dedicated notion of type for active entities – i.e., the agents in our case – which is still missing in programming languages and frameworks rooted on actors and active objects. In general, the introduction of typing, first enables compile-time error checking, greatly reducing the cost of errors detection—from both a temporal and economic point of view. Second, it provides developers a conceptual tool for modeling generalization/specialization relationships among concepts and abstractions, eventually specializing existing ones through the definition of proper sub-types and making it possible to fully exploit the principle of substitutability [WZ88] for supporting a safe extension and reuse in programming. Accordingly, we argue that it is fundamental to introduce an explicit notion of type for active entities in order to provide developers, besides static error checking controls, the adequate abstraction tools to model and characterize the different active parts of a software systems.
- The introduction of types has given us also the opportunity to investigate and define some basic forms of polymorphism in simpAL programs (Section 7.3.3).
- We concretely evaluated both the simpAL programming approach and its related development platform through the implementation of relevant programming examples, which have been purposely developed to stress the programming abstractions supporting the integration of autonomous and event-driven behaviors.
- We engineered a suitable ecosystem around the simpAL programming language composed by an integrated development environment and a distributed runtime infrastructure. The final objective that has driven the development of such ecosystem was trying to provide developers the best means for coding, deploying, executing and inspecting concurrent and distributed simpAL applications.

1.2 Outline of the Dissertation

Besides this introductory chapter, this dissertation is organized in five macro-parts. In detail, the remainder of this thesis is organized as follows.

- ***Part I: Setting The Stage*** — the role of this first part is introduce the background context of the thesis, along with the presentation of the main related work. Accordingly, Chapter 2 provides the required background on the actor model that, on the basis of current trends experienced in the context of mainstream software development (Section 2.3), is taken here as the reference forthcoming state-of-the-art programming paradigm for the engineering of complex, concurrent and distributed applications. Chapter 3 provides instead the fundamental background on agent and multi-agent systems, and in particular on state-of-the-art programming languages and technologies that have been introduced for their programming and development. This chapter is structured along the four main programming dimensions that have been introduced for the practical development of MASs, namely: the agent, environment, interaction and organization dimensions.

- ***Part II: Engineering Agent-Oriented Technologies for Programming Multi-Agent Systems*** — in this part of the thesis we describe the result of our efforts in the engineering of *integrated* – i.e., concerning the synergistic use of multiple programming dimensions – programming approaches and concrete technologies supporting the development of multi-agent systems, in the (distributed) artificial intelligence context. In particular in Chapter 4 is presented the JaCa platform – along with its main extensions – integrating both the agent and environment dimensions; in Chapter 5 instead is described the JaCaMo platform, which provides a synergistic integration of the agent, environment and organization programming dimensions. A concrete evaluation of both platforms, through the development of real world applications, is provided in the dedicated chapters. Finally, Chapter 6 sets the stage for the next part of the dissertation, by providing a critical discussion about the good points and the current limitations of both JaCa and JaCaMo – taken in the chapter as reference examples of state-of-the-art technologies for developing MASs – when moving from the distributed artificial intelligence context to general-purpose computer programming and software development—the reference context for this thesis work.

- ***Part III: The simpAL Project*** — in this part is described the core contribution of the dissertation, simpAL an agent-oriented programming language which has its foundations rooted on the key concepts and features introduced and developed in the history of modern mainstream programming languages (e.g., typing, polymorphism, etc.), properly revised for an agent-oriented language. In Chapter 7 we first provide a general overview of the simpAL language, then is given a description of the simpAL programming model along with its main features, also making comparisons (when possible) with similar features in object-oriented and actor-oriented approaches. Finally, the chapter is concluded by a concrete evaluation of the language through the implementation of relevant programming examples and by a discussion addressing performance tests and current limitations.

- ***Part IV: Conclusion*** — this part concludes the dissertation by presenting final remarks, and discussing possible future work and research directions.
- ***Part V: Appendix*** — this appendix contains: *(i)* the EBNF (Extended BackusNaur Form) grammar of the simpAL language, and *(ii)* additional sources related to performance tests and applicative examples.

Part I

Setting The Stage

2
Background on the Actor Model

This chapter provides the required background on the actor model and on actor-oriented programming. The chapter is structured as follows:

- Initially in Section 2.1 is given a general overview of the actor model, along with its main features.
- Then in Section 2.2 are presented the main programming abstractions that have been introduced in the state-of-the-art to facilitate the programming of actor-based systems.
- Finally, the chapter is concluded by presenting a discussion concerning both the growing diffusion/adoption of the actor paradigm in mainstream programming (Section 2.3), and current shortcomings and limitations of actors as a reference model of concurrent and distributed computation (Section 2.4).

2.1 Overview of The Actor Model

The actor model has been firstly introduced by Hewitt in the 1973 [HBS73][1], and since then further extensions an contributions to the model have been provided by Agha [Agh86, AMST97], Yonezawa [YBS86, Yon90], and others [HB77, Cli81]. From the seventies, the actor concept evolved as a result of more than twenty years of subsequent efforts, influencing a range of concurrent object-oriented systems [YT87]. As a result, there is no one language or framework that embodies all of the concepts that have been developed in the literature [Mit02]. Following what stated in [KA11], for the sake of the construction of this thesis' background we will consider the most commonly used definition of actors today, which follows the work of Agha [Agh86].

The actor model is a model of concurrent computation for developing parallel, distributed and mobile systems [KA11]. An actor system (Figure 2.1) consists of a collection of actors communicating by exchanging asynchronous messages, some of whom may send messages to, or receive messages from, actors outside the system. The data exchanged in messages can

[1] Even if [HBS73] is generally referred as the first work introducing the actor computational model, the earliest use of the term *actors* was in Hewitt's early work on PLANNER [Hew69]. Here the term was coined for referring to rule-based active entities which search a knowledge base for patterns to match, and in response, trigger actions.

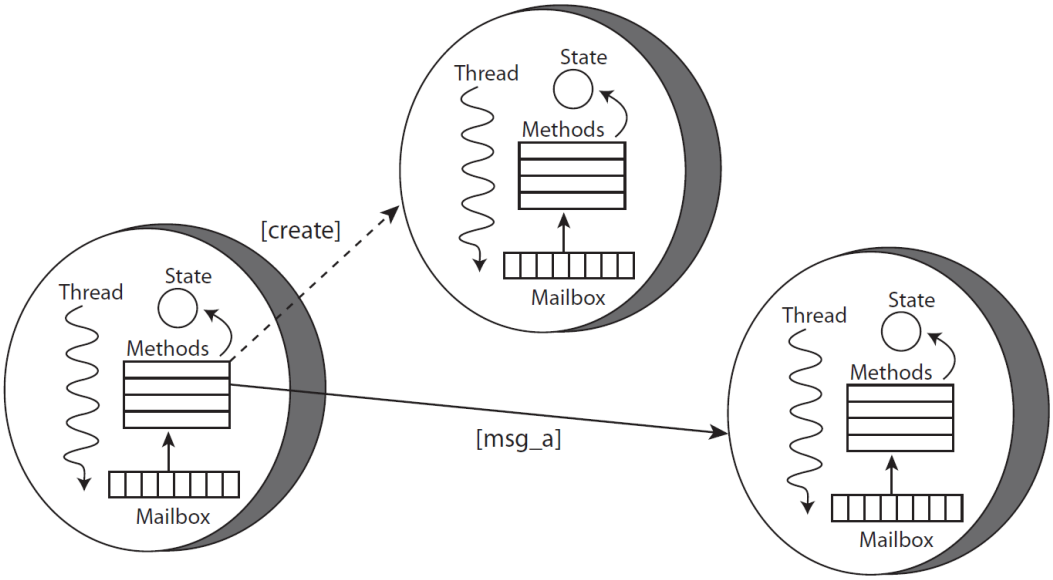


Figure 2.1: Abstract view of an actor system [KA11].

range from primitive data types, to complex ad-hoc structured informations. Each actor is an autonomous process that operates in a *concurrent* and *asynchronous* manner, receiving and sending messages to other actors, creating new actors, and updating its own *local* state. Each actor has:

- Its own mutable local state—actors do not share this local state with other actors, each actor is responsible for updating its own local state. An actor may affect the state of another actor only by sending the second actor a message.
- A mailbox in which incoming messages are stored. Messages can arrive in the mailbox at any time and will be held there until the recipient is ready to process them. Due to the different paths a message may take and unexpected network delays, the order of message delivery is *indeterminate*. As a consequence, the order in which messages are processed by an actor can not be determined in advance.
- A unique, immutable name which is required to send a message to that actor. An actor name can not be guessed, but may be communicated to other actors.

The basic primitives available for programming the behavior of an actor are:

- `create:` to create a new actor starting from its behavior description and a set of input parameters, possibly including the reference to existing actors.

- `send`: to *asynchronously* send a message to an actor.
- `become`: to designate the behavior (local state) to be used for the next message the actor receives—i.e., the mechanism used by an actor to replace its own behavior with a new one. This gives actors a history-sensitive behavior necessary for shared, mutable data objects.

Several programming languages rooted on the actor model exist. Among these, Erlang [Arm10] is arguably the best known. It was developed to program telecommunication switches at Ericsson about 20 years ago. Other languages worth to mention are: SALSA [VA01] (mainly targeting the Internet domain), ActorNet [KSMA06] (developed in the context of sensor networks), E [MB], Axum [Micb] and Ptolemy [BHLM02].

Despite the development of a number of Actor languages, there continue to be efforts to develop Actor frameworks based on familiar languages such as C/C++ (Act++ [Kaf90], Thal [Kim97]), Smalltalk (Actalk [Bri89]), Python (Stackless Python [Chr], Parley [Jac]), Ruby (Stage [Sil08]), .NET (Retlang [Mikb]) and Java (Scala Actors library [HS12, Typb], akka [Gup12, Typa], Kilim [SM08], Jetlang [Mika], ActorFoundry [KA], AmbientTalk [DVCM+06], Actors Guild [Tim]).

Depending on the concrete actor-oriented programming language or development framework considered, the basic actor programming primitives described above can be directly available among the set of programming constructs provided by the language/framework or not. As a concrete example, akka provides an explicit support for the `become` primitive, while Erlang does not—in Erlang the `become` is implemented manually by programmers, invoking a function that encapsulates the new actor behavior.

Algorithm 1 Actor Event Loop

```
1: while true do
2:     msg ← PICKMSGFROMMSGQUEUE()
3:     method ← SELECTHANDLER(msg)
4:     EXECUTE(method)
5: end while
```

From a behavioral point of view, actors – like objects[2] – are based on reactivity and the *reactivity principle* [Agh90, BGL98, Kay69]. Actors are *reactive* in the sense that they *react to an event*, i.e. the receipt of a message. The only way to activate an actor is by sending a message. So finally, actors can be conceived as processes continuously executing the above loop, called *event loop*[3] [MTS05].

[2] As often reminded by Alan Kay [Kay96], OOP was initially deeply rooted on the concept of message passing: *"The big idea is messaging – that is what the kernel of Smalltalk/Squeak is all about..."* (this is a quote from a Smalltalk discussion group [Kay98].

[3] Event loops are widely recognized to be part of the actor model by the research community working on actors but Carl Hewitt: *"[...] In any case, 'event loop' is confusing terminology because there can*

All the extensions introduced in literature – presented in the next section – such as making it explicit the `receive` primitive or providing a way to order the messages to receive, can be finally translated into this basic loop [AMST97]. Two aspects are worth to be emphasized, in particular for the background of this dissertation: *(i)* if no messages are available in the queue, the loop is *blocked* (this is consistent with the reactivity principle); *(ii)* the method selected for handling a message must be executed until completion, *atomically, before fetching the next message*—this is called *macro-step semantics* [AMST97].

Some other important semantic properties of the pure Actor model are *fairness* and *location transparency*. The former ensures that every actor makes progress if it has some computation to do, and that every message is eventually delivered to the destination actor, unless it is permanently disabled (i.e., it has terminated its execution). It has been shown that this property is particularly useful for simplifying the reasoning about liveness properties in actor programs [AMST97]. The latter instead makes communications among actors independent from their actual physical location. Actors communicate by exchanging messages with other actors, which could be on the same core, on the same CPU, or on another node in the network. Location transparent naming also facilitates runtime migration of actors to different nodes. In turn, migration can enable runtime optimizations for load-balancing and fault-tolerance.

Different languages and frameworks support the main properties of the actor model at different levels, in different ways. E.g., fault-tolerance is one of the key features of Erlang and akka. For a more detailed comparison among some of the most relevant actor frameworks in the state-of-the-art the interested reader can refer to [KSA09].

2.2 Programming Abstractions

Asynchronous communications are the only ones supported by the basic actor model. So, by strictly following the basic model, complex forms of communication must be programmed by hand, by defining the appropriate exchange of asynchronous messages between actors. It is clear that such a basic approach could be problematic when developing real-world software systems: most of the time not all the interactions among the different parts of an application can be easily modeled as asynchronous communications.

Accordingly, over the years, several programming abstractions have been introduced to ease the definition of complex forms of communication and synchronization in actor programs. In the remainder of this section we introduce some of the mains ones.

2.2.1 Request-Reply Messaging Pattern

Synchronous communications rooted on the request-reply messaging pattern are one of the most common used in the development of software systems—e.g., a client that requests a certain

be "holes in the cheese." [HA79], which means code is just nested expressions (i.e., no loop)..." `http://lambda-the-ultimate.org/node/4453`

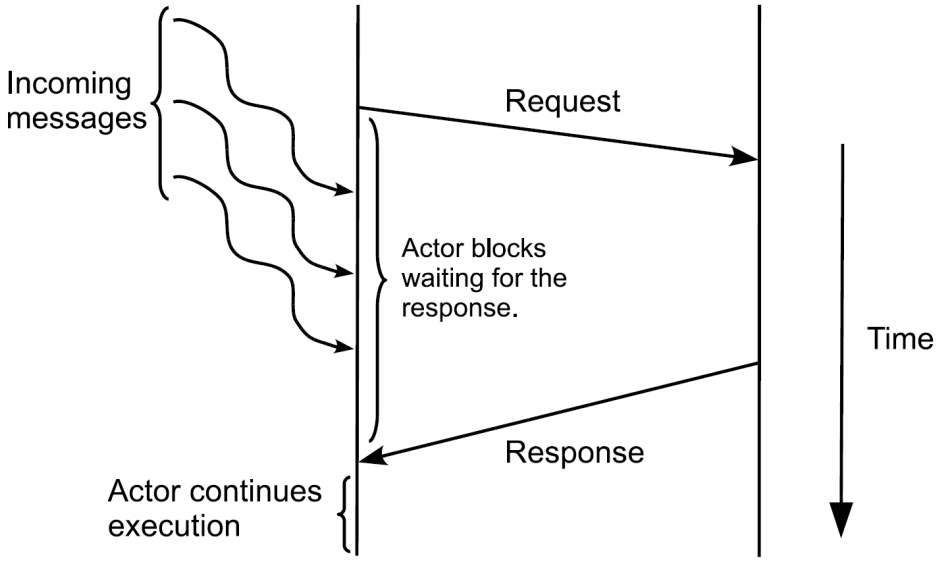


Figure 2.2: Request-reply messaging pattern blocks the sender of a request until it receives the reply. All other incoming messages during this period are deferred for later processing [KA11].

functionality to a server and needs to inspect the server's response message in order to decide what are the next actions to do. In this pattern, the sender of a message blocks, waiting for the reply to arrive before it can proceed. The actor model does not explicitly support synchronous communications. So, without a high-level language abstraction programmers have to explicitly implement the behavior of their client programs as follows: *(i)* send the request message; *(ii)* wait for incoming messages; *(iii)* when a new message arrives it must be checked whether the message is a reply to the request, or it is another message that happened to arrive between the request and the reply, *(iv)* if the incoming message corresponds to the waited reply, take the appropriate actions, otherwise, the message has to be buffered for further processing and the client must continue to check messages for the reply.

To avoid the burden of programming by hands all these steps, ther request-reply messaging pattern (Figure 2.2) is almost universally supported in actor languages and frameworks. For example, is available as a basic language primitive in Scala Actors, akka, SALSA, ActorFoundry, etc.

2.2.2 Local Synchronization Constraints

As mentioned in Section 2.1, the order in which messages are processed by an actor is nondeterministic. However, sometimes it is needed to process messages in a specific order. A classical example of this need is given by a bounded buffer actor which needs to defer the pro-

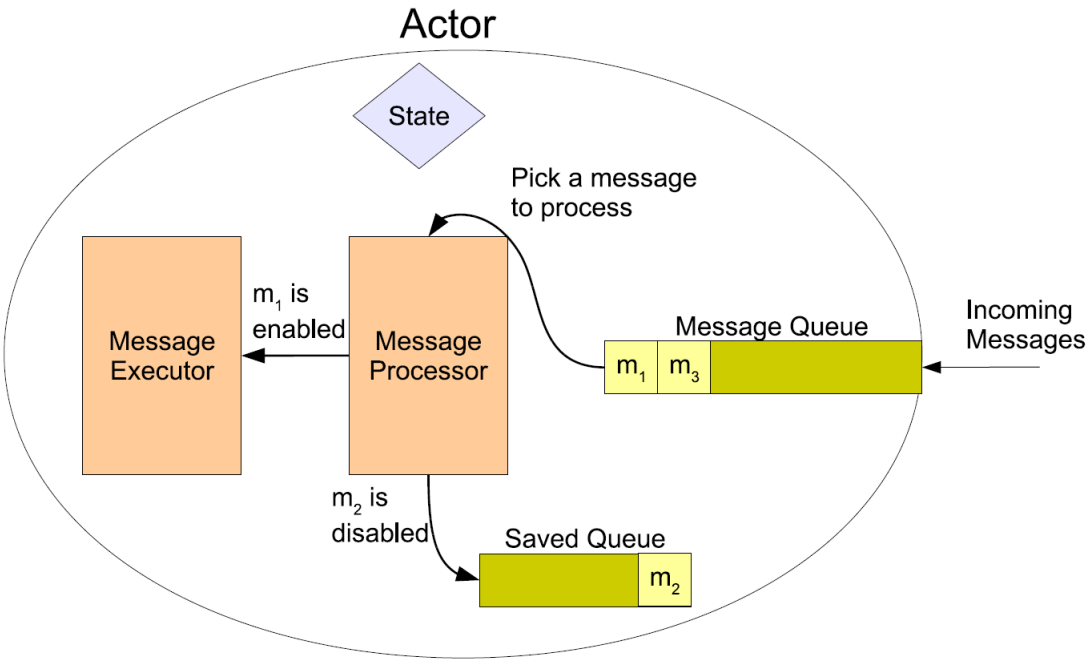


Figure 2.3: Implementation semantics of local synchronization constraints in Actor-Foundry [KA11].

cessing of `put` and `get` messages on the basis of the current state of the buffer—i.e., `put` messages can be processed only when the buffer is not full, `get` messages when the buffer is not empty. To this end, the order in which messages are processed must be properly *restricted* or *regulated*. Synchronization constraints simplify the task of programming such restrictions on the order in which messages are processed. For example, they can allow the programmer to declare that an actor postpones the processing of a message until it receives some sequence of messages, or until a condition on the actor's state is satisfied.

In the case of languages and frameworks with an explicit `receive` (e.g., Erlang and Scala Actors) this can be obtained simply by doing pattern matching on incoming messages. In the case of other languages instead, explicit first-class language constructs must be provided to this end. The following code shows an example of synchronization constraint written in Actor-Foundry.

```
1  public Boolean disablePut(Integer x) {
2    if (bufferReady) {
3      return (tail == bufferSize);
4    } else {
5      return true;
6  }}
```

Here, the Java annotation framework has been used to define a synchronization constraint for the message `put`, whose condition is expressed by the method `disablePut`. In the above code,

the constraint returns `true` if the `put` message can not be processed in the actor's current state. At runtime, a message is processed if it is not disabled i.e., no constraint returns `true` for the message. When a disabled message is received, it is placed in a queue called *save queue* for later processing (see Figure 2.3). Whenever a message is processed successfully by an actor, it is possible that a previously disabled message (a message in save queue) is no longer disabled. This is because the state of the actor may change after processing a message, and this change of state may enable other messages.

2.2.3 Continuations and Promises

In this sub-section we present two other programming constructs that have been introduced in the state-of-the-art in oder to ease the exchange of messages among actors: *(i)* continuations (*join* [AH87] and *token-passing continuations* [VA01]), and *promises* [MTS05] (often also called *futures*). The first mechanism allows to define the continuation of actors' behavior in response to the reception of a set of user-defined messages. In particular, via join continuations it is possible to define the set of messages that need to be received – in any sequence – before the actor can execute a certain message handler—i.e., the actor continuation.

Token-passing continuations allow instead to write chained sequences of asynchronous statements which can be connected as needed. The order of the asynchronous invocations is determined by the passing of a token, which enables the sending of the next message. As a concrete example we consider now an snippet of code written in SALSA:

```
1 a1 <- m1() @ a2 <- m2(token);
```

In the above code `@` represents the token, while the notation `msg_recipient <- msg()` means that message `msg` is sent asynchronously to the `msg_recipient` actor. In the example `m2` will be sent to actor `a2` only when message *m1* has been processed by actor `a1`: indeed, the argument to `m2` is the token that proves that `a1` has finished the handling of `m1`. We consider now an example, in SALSA, that involves both join and token-passing continuations:

```
1 join{ a1<- m1(); a2 <- m2(); } @ a <- m();
```

The above code will send messages `m1` and `m2` concurrently to actors `a1` and `a2`. Then, message `m` will be sent to actor `a` only when both `m1` and `m2` have been received and processed by the respective recipients.

Finally, the last programming construct that we consider here are promises (or futures). Generally speaking, in computer science futures/promises refer to a programming construct that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete [Wik]. Besides actor-oriented languages, this construct is also available in many object-oriented ones such as Java [SUNa] and C# [Mica].

In an actor program, promises are useful in particular in the case of request-reply message exchanges. Indeed, in some cases, we want the calling actor to return immediately after sending a message, but we may also need access to the target actor's reply at a later time. Using a

promise the calling actor can send the request message and then continue with its computation immediately. Later on, when it needs the result, it can check the promise that was specified at the time of the request transmission. If the result has been stored in the promise object, it can be used. Given a promise reference, the calling actor can both: *(i)* check whether or not the result is available, and *(ii)* if needed, block until the response is retrieved.

In addition, in some actor-oriented languages such as E [MB] promises can be also *pipelined* [MTS05]. Promise pipelining brings two main benefits. On the one side, it enables the definition of ad-hoc workflows concerning asynchronous message exchanges. This can be particularly useful for handling and managing the reception of multiple futures, possibly in an arbitrary order. The following code shows an example of use of promise pipelining in E:

```
1 def r3 := x <- a() <- c (y <-b())
```

Like in the case of SALSA, the notation `x <- a()` means sending the message `a` to actor `x` asynchronously. Without using promise pipelining, one would have been forced to deal manually with the resolution of each promise, constructing step by step the previous workflow.

On the other side promise pipelining is also useful to reduce the number of round trips involved in message exchanges, and in turn the overall latency of the system [MTS05]. Indeed, when multiple messages are sent to the same actors, or even to multiple actors that however reside on the same machine, such messages can be streamed together and sent in just one round trip.

2.3 Actor-Oriented Programming: The Next Big Thing is Already Here

As discussed in the introduction Chapter 1, the actor paradigm is becoming more and more one of the reference programming paradigms for the development of modern software systems—i.e., concurrent and distributed systems. It is recognizable in the mainstream a convergence toward the adoption of the actor model and actor-related approaches as unification of object-oriented programming and concurrency.

Moreover, in the last years there has been a proliferation of actor-oriented languages and frameworks which are increasingly becoming more robust and mature, and are starting to be applied outside the usual domain boundaries of massive parallel programming and related specific application contexts. Besides Erlang that mainly targets the specific domain of telecommunications, there are several well known, recognized and adopted actor-oriented frameworks targetting general purpose computing: Scala actors, akka, and ActorFoundry are main examples. Eventually, these languages and frameworks are starting to be applied for programming in the large, and particularly in some of the hottest and most relevant application domains such as the web and the mobile context. E.g., it is well known that a core part of the current implementation of Twitter have been written exploiting the Scala actor library [Eri10]. Moreover, DART [Incb], a recent programming language by Google for the development of fore-coming

web applications, comes along with an actor-based library [Inca]. For what concerns the mobile context instead, it is possible to see that inside the Android SDK [Gooa], even if the term actor is not explicitly mentioned, the Looper [Good], one of the key components of the programming model, has been engineered strictly mimicking actors' behavior.

Summing up, given the current trends in mainstream software development, we argue that actor-oriented programming is fated to be the *short-term* answer for the free lunch is over call in the near future.

2.4 Main Shortcomings and Limitations

It is undeniable that actors represent a step forward w.r.t. mainstream object-oriented programming languages for the engineering of concurrent and distributed software systems as an ensemble of autonomous computing entities communicating through asynchronous message passing. Moreover, important features such as location transparency, fault tolerance etc. are directly part of the actor model itself (Section 2.1).

Nevertheless, we argue that the actor paradigm can be only considered a good starting point to provide a fully comprehensive response to the fundamental and radical challenges introduced by the free lunch is over call. Accordingly, we describe here what are – in our opinion – the main flaws and limitations that motivate such a strong assumption.

Firstly, one can argue that important features such as inheritance, sub-typing, polymorphism, etc., which are nowadays a well-defined part of the object-oriented paradigm, and in turn have become important aspects of the software engineering process, are either completely missing, or have not found a corresponding easy settlement in the context of actors. A main example is given by the support for inheritance, which is still flawed by the inheritance anomaly problem [MY93] more than ten years after its first discovery [MS04]. A second example is given by the absence of a well defined notion of type and related sub-typing relations for actors. To the best of our knowledge, there is not strong or mature work that consider this issue. An exception is given by the akka framework, which introduces the notion of typed actors [Gup12, Typa]. A typed actor in akka is given by a public interface – written in Java or Scala – and a concrete actor implementation. The interface is used to define the set of messages that an actor implementing such an interface is able to understand. From the one side this is useful to check programming errors related to message exchanges at compile time. On the other side this is clearly a programming trick just to enable error checking, which does not address the issue of typing and sub-typing at the conceptual and foundation levels. Questions such as *"what does it mean to define an actor sub-type?"*, or *"which kind of messages an actor sub-type is able to understand?"*, etc. still do not have a rigorous answer.

Secondly, from a conceptual viewpoint, being based on a pure *reactivity principle* [BGL98, Kay69], actors – as objects as well – do not provide native means to effectively model and structure the autonomous behavior of active entities, and in turn of an entire application. An actor does something in response to the reception of a certain message. So, one is forced to

program the autonomous behavior of an active entity on a pure event-driven basis—i.e., on the basis of the set of messages the active entity must be able to deal with.

Another issue, which is strictly related to the previous one, concerns the integration of autonomous and reactive behaviors, which is in general a relevant problem in the context of concurrent programming. A couple of proposals to solve this issue have been introduced in literature in the context of actor-oriented programming [VCMDM09, HO08, HO09]. However, we argue that they are not fully adequate to solve the problem. The same also applies for all the programming abstractions and constructs described in Section 2.2. Indeed, the final objective of these extensions is simply the management of complex chain of messages, thus, improving the programming of the reactive part. A broader and more in depth discussion related to the integration of autonomous and reactive behaviors will be described in detail in Section 7.3.1.

Finally, as often happens in computer science, the actor paradigm strives to provide a single abstraction to model every component of a system: everything is an actor, i.e. an active element. This has the merit of providing uniformity and simplicity. At the same time, the perspective in which everything is an active, autonomous entity is not really always effective, at least from an abstraction point of view. For instance – even if possible – it is not really natural to model as active entities either a shared bounded-buffer in producers/consumers architectures, a blackboard, or a simple shared counter in concurrent programs. In traditional thread-based systems such entities are designed as monitors, which are passive. More generally, we argue that in the context of concurrent programs, the availability of first-class abstractions to effectively model shared-memory avoiding interferences and related synchronization issues is still important today for many problems like it was when monitors were devised.

To conclude this discussion, as well summarized by Mitchell in [Mit02]: *"although the simplicity of the actor model is appealing, the problems with message order, message delivery, and coordination between sequences of concurrent actions also help us appreciate the programming value of more complex concurrent languages"*. So, getting back to the free lunch call we argue that actors can be seen as a strong foundation layer, but giving finally solely asynchronous message passing as unique concept, and so, not significantly enhancing the set of abstractions that we can use to simplify the design and engineering of concurrent and distributed systems.

3
Background on Programming Multi-Agent Systems

This chapter provides the fundamental background on agent and multi-agent systems, and particularly on state-of-the-art programming languages and technologies that have been introduced for their programming and development. The overall structure of the chapter is organized on the basis of the four main programming dimensions that have been introduced for the practical development of MASs. In detail, this chapter is structured as follows:

- In Section 3.1 a brief overview of agent and multi-agent systems is provided.
- Agent models, architectures, theories, programming languages and frameworks focused on the agent programming dimension are described in Section 3.2.
- The programming of the environment dimension, along with the main environment models and computational infrastructures introduced in the-state-of-the-art are presented in Section 3.3.
- The programming of the interaction and organization dimensions and related communication and organizational models are discussed in Section 3.4.

3.1 Introduction

Multi-Agent Systems (MASs) are systems composed of multiple interacting computing elements, known as *agents*. An agent is a computer system that is *situated* in some *environment*, and that is capable of *autonomous actions* in this environment in order to meet its design objectives [Woo97]. The main features of an agent are:

- *Autonomy*: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state [Cas94].
- *Situatedness*: as already mentioned, agents are entities *situated* in some *environment*. Agents are capable of sensing their environment (via *sensors*), and have a repertoire of

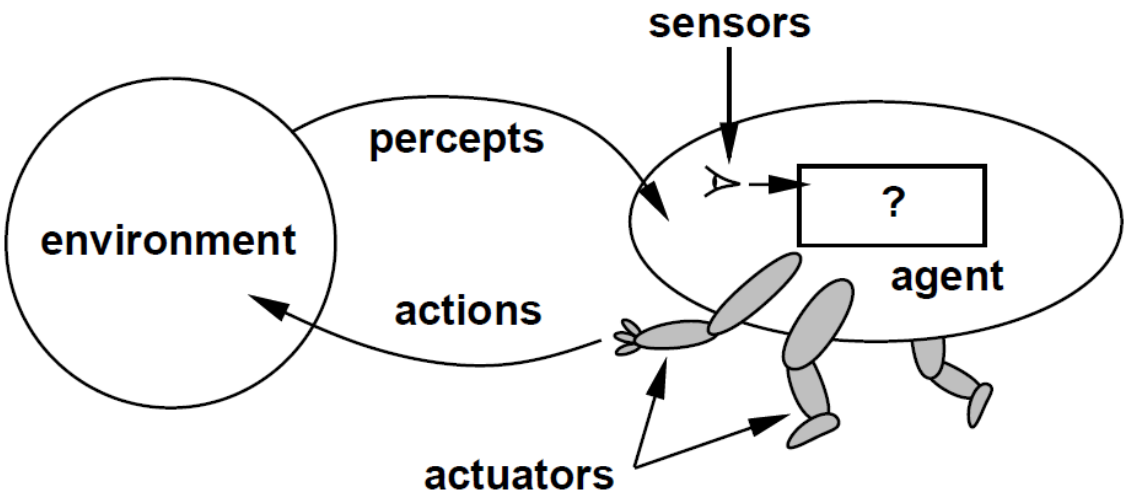


Figure 3.1: An agent in its environment [RN09]. The agent perceives sensory information from the environment through its sensors – generating, if needed, internal percepts – and is able to interact/inspect/affect the environment itself via a set of proper external/environment actions.

possible actions that they can perform (via *effectors* or *actuators*) in order to modify it (see Figure 3.1).

- *Pro-activeness*: agents do not simply act in response to stimuli coming from their environment or other agents. They are able to decide for themselves what they need to do in order to satisfy their design objectives.

- *Social ability*: agents interact with other agents (and possibly humans) via some kind of agent-communication language [GK94].

- *Reactivity*: agents are able to react to both events coming from the environment and from communications received from other agents, and respond in a timely fashion to them.

Multi-agent systems have been studied as a field in their own since about 1980, and the field gained widespread recognition in the mids nineties. The research on MASs is not tied to a single domain, since they seem to find currency in a host of different research domains [Woo09]. Some of the most important ones are: *(i)* (distributed) Artificial Intelligence (D)AI and Agent-Oriented Software Engineering (AOSE), where theories, agent models/architectures and related agent-oriented programming languages (APLs) and frameworks are defined with the purpose of designing *intelligent* software entities, and finally intelligent software systems; *(ii)* social science, where MASs are used as a tool to model and investigate the functioning of human

societies [GD94, Eps11]; *(iii)* game theory and economics, where agent-based theories, techniques and tools are developed in order to study and predict forms of interaction and negotiation among different parties, which may have different objectives [PW02, RZ94, NRTV07]; *(iv)* agent-based simulation, where models, theories and adequate software tools are studied and developed in order to simulate the most disparate things [SCG98, AT07, BGJ11, VSMS12].

For what concerns this dissertation, we are mainly interested in the exploitation of agent and multi-agent system as a programming paradigm for the development of intelligent software systems. The idea of agent-oriented programming (AOP) was coined by Y. Shoham [Sho93] as a new programming paradigm combining the use of *mentalistic* notions (such as belief) for programming (individual) autonomous agents in a societal view of computation. In the nineties, most of the work centered on a few agent models, architectures and languages that had significant theoretical work but limited use in practice, and mostly centered on developing individual agents whether for a multi-agent system context or not [Tho95, Sho93, BFG+90, Fis94, LRL+97, DGLL00]. The work in this area in the two thousands changed substantially this picture by producing many different APLs based on varied underlying formalisms and inspired by various other programming paradigms. Furthermore, many such languages were developed into serious programming approaches, with working platforms and development tools (see Section 3.2). This led to some of these languages having now growing user bases and being used in many AI and multi-agent systems university courses.

The important contribution of agent-oriented programming as a new paradigm was to provide ways to help programmers to develop autonomous systems. For example, agent programming languages typically have high-level programming constructs which *facilitate* (compared to traditional programming languages) the development of systems that are continuously running and reacting to events that characterize changes in the dynamic environments where such autonomous systems usually operate. Not only agents need to take on new opportunities or revise planned courses of action because of changes in the environment, agent programming also facilitates programming agent behavior that is not only *reactive* but also *pro-active* in attempting to achieve long-term goals. The features of agent programming also make it easier, again compared to other paradigms, for programmers to ensure the agents behave in a way that in the agents literature is referred to as *"rational"*. For example, if a course of action is taken in order to achieve a particular goal (typically explicitly represented in the agent state), if the agent realizes that the goal has not been achieved we would not expect the agent not to take further actions to achieve that goal on behalf of its human designer, unless there is sufficient evidence that the goal can no longer be achieved or is no longer needed.

Multi-agent systems are normally used to develop very complex systems, where not only many autonomous entities are present, but they need to interact in complex ways and need to have social structures and norms to regulate the overall social behavior that is expected of them and, equally important, a shared environment can be a relevant and efficient source of coordination means for autonomous agents. Accordingly, other researchers focused their efforts in the study of social structures, interactions and environmental aspects of the development of multi-agent systems. This has lead to the proliferation of what could appear to

be other (separate) programming paradigms, which are focused in one specific programming dimension of MASs. These programming paradigms are: Organization-Oriented Programming (OOP) [BHS07, PTCC99] to program the organization dimension, Interaction-Oriented Programming (IOP) [Huh01] to program the interaction dimension, and finally Environment-Oriented Programming (EOP) [RPV11, RPVO09] to program the environment dimension.

In the remainder of this background chapter, for each programming dimension, we present the main models, programming languages and frameworks that have been developed in the state-of-the-art.

3.2 Programming the Agent Dimension

Several agent models and architectures have been proposed over the years, many of them rooted on logical theories of rational agency. Besides the firsts that have been introduced – firsts, from a temporal point of view – and which have been already mentioned in the previous section, some of the main ones are: the Belief Desire Intention (BDI) model [RG$^+$95], the Knowledge Goals and Plan (KGP) model [KMS$^+$04], Minerva [LAP01], the Knowledge Abilities Results and Opportunity (KARO) logic and theory [HLM94], and the Belief Obligations Intentions Desires (BOID) architecture [BDH$^+$01].

Among these, the BDI agent model is nowadays becoming more and more the reference one used in state-of-the-art APLs for the development of intelligent agents. Accordingly, in the remainder of this section first we provide a presentation of this model, which represents a key part of the background for this dissertation—indeed: *(i)* it is the agent model used in both JaCa (Chapter 4) and JaCaMo (Chapter 5), and *(ii)* it has been the starting point for the development of the simpAL agent model Section 7.1.2. Then, we present some of the most relevant APLs in the state-of-the-art, some of which not necessarily rooted on the BDI model.

3.2.1 The Belief Desire Intention Agent Model

The BDI model originated from studies on human behavior in the context of the Rational Agency project at the Stanford Research Institute in the mid-eighties. The origins of the model lie in the theory of human practical reasoning developed by the philosopher Michael Bratman [Bra87], which focuses particularly on the role of intentions in practical reasoning. The conceptual framework of the BDI model was first presented in [BIP07], and then further improvements and contributions has been made by Rao and Georgeff [RG98, RG$^+$95, GPP$^+$99].

Following the BDI model, an agent is characterized by a *mental state* which is defined – not surprisingly – on the basis of *beliefs*, *desires* and *intentions*. Roughly, these concepts can be described as follows [BHW07]:

- Beliefs — are information the agent has about the world and other agents part of the MAS. They can be considered *analogous* to the variables available in the context of classical programing languages.

- Desires — are all the possible states of affairs that the agent might like to accomplish. Having a desire, however, does not imply that an agent acts upon it: it is a potential influencer of the agent's actions. It is worth noting that it is perfectly reasonable for a rational agent to have desires that are mutually incompatible with one another—i.e., desires can be considered as sort of *options* for an agent.
- Intentions — are the states of affairs that the agent *has decided to bring about*. Intentions may be goals that are delegated to the agent, or may result from considering options. Options that are selected become intentions. Therefore, we can imagine our agent starting with some delegated goal, and then considering the possible options that are compatible with this delegated goal; the options that it chooses are then intentions, which the agent is committed to.

Practical Reasoning

How does an agent with beliefs, desires and intentions go from these to its actions? The answer to this question is given by the *practical reasoning*. Practical reasoning is the decisional process on top of which the functioning of a BDI agent is rooted. It can be defined as a reasoning process *directed toward actions*.

The practical reasoning process is characterized by two distinct phases or activities: the *deliberation* phase and the *means-ends* reasoning phase. During the deliberation phase the agent decides from a set of available options which are the intentions it wants to achieve, on the basis of its current representation of the world (expressed in terms of current beliefs, desires and intentions). During the means-ends reasoning phase the agent decides how to achieve an end (i.e., a chosen intention) using the available means (i.e., the actions it can perform in the environment or the communicative actions that can be used to interact with other agents).

So, finally a BDI agent can be considered like a *reactive planning system*, whose computational behavior follows the practical reasoning, on the basis of the agent control loop described in Figure 3.2: A brief description of the control loop follows. As it easy to suppose variables B, D and I are introduced to represent respectively the agent's beliefs, desires and intentions. Firstly the agent gets new percepts either from the environment or from other agents through its sensors (line 4). Then, the new obtained perceptual information is used, in conjunction with the current beliefs, to obtain the new beliefs of the agent via the belief revision function (`brf`, line 5). Now the agent updates its own desires and intentions by exploiting the functions `options` and `filter` (lines 6-7), then, using the `plan` function, (line 8) it searches for plans to achieve the selected intentions. If all goes well, then the agent simply picks off each action in turn from its plan and executes it, until the plan π is *empty*—i.e., all the actions in the plan have been executed with success and the related intention achieved.

It is worth noting that, after executing an action from the plan (line 11), the agent: observes again the environment (line 13), invokes the `brf` function to updates its beliefs (line 14), and, if needed, it can also update the set of its desires and intentions (lines 16-17). So, after the execution of each action and on the basis of its actual knowledge of the world, the agent has

```
1.  B ← B0;          /* B0 are initial beliefs */
2.  I ← I0;          /* I0 are initial intentions */
3.  while true do
4.    get next percept ρ via sensors;
5.    B ← brf(B, ρ);
6.    D ← options(B, I);
7.    I ← filter(B, D, I);
8.    π ← plan(B, I, Ac); /* Ac is the set of actions */
9.    while not (empty(π) or succeeded(I, B) or impossible(I, B)) do
10.     α ← first element of π;
11.     execute(α);
12.     π ← tail of π;
13.     observe environment to get next percept ρ;
14.     B ← brf(B, ρ);
15.     if reconsider(I, B) then
16.       D ← options(B, I);
17.       I ← filter(B, D, I);
18.     end-if
19.     if not sound(π, I, B) then
20.       π ← plan(B, I, Ac)
21.     end-if
22.   end-while
23. end-while
```

Figure 3.2: The BDI Agent Control Loop [BHW07].

the opportunity to decide if continue to purse its current intentions or not. Such decision can also lead to the premature termination of a plan and its related intention, which are no more considered feasible (this check is performed by the function *sound*, line 19).

3.2.2 Agent-Oriented Programming Languages and Frameworks

In this sub-section we provide a brief survey of the main agent-oriented programming languages and frameworks that have been introduced over the years in the state-of-

the-art [BBD+06]. Among the languages we consider: ***Jason*** [BHW07, BH06, Raf], Agent Factory [RJOC11, Rem], CLAIM along with its distributed platform SyMPA [FSS05], GOAL [Hin09], JACK [Age] and 2APL [DvRM05] / 3APL [Das08].

Jason [BHW07, BH06, Raf] — is a platform for the development of multi-agent systems that incorporates a BDI-based agent-oriented programming language and a related Integrated Development Environment (IDE). The language comes from an extended version of AgentSpeak(L), a logic-based APL introduced by Rao in [Rao96], which has been later much extended in a series of publications by Bordini, Hübner, and colleagues, so as to make it suitable as a practical agent programming language.

Jason is available open-source under GNU LGPL from the ***Jason*** official website [Raf]. The interpreter has been developed in order to be modular and easily customizable. Indeed, it is quite straightforward for MAS developers to implement their own customizations – e.g., custom belief base, custom belief-update and belief-retrieval functions, modifications to the core functioning of the agent reasoning cycle, etc. – on top of an extensible architecture.

The ***Jason*** IDE provides a graphical interface for editing a multi-agent system configuration file, as well as AgentSpeak code for the individual agents. Through the IDE, it is possible to run and control the execution of multi-agent systems. The IDE also provides another tool, called *"Mind Inspector"*, a sort of agent-oriented debugger which allows the user to inspect agents' internal states when the system is running in debugging mode.

The Agent Factory Framework [RJOC11, Rem] — is a cohesive framework for the development and deployment of multi-agent systems. The development of the framework started in the mid-nineties, but has gone through a significant redevelopment whereby several new extensions, revisions, and enhancements have been made. The first version of the Agent Factory Agent Programming Language (AFAPL) was originally based on Agent-Oriented Programming as first put forward by Shoham [Sho93], but was later revised and extended with BDI concepts, such as beliefs and plans.

Today Agent Factory is an open framework for building agent-based systems, and as such, does not enforce a single flavor of agent upon the developer. Instead, developers are free to either use a pre-existing agent interpreter / architecture, or develop a custom solution that is more suited to their specific needs. The framework provides a set of pre-written components for building agent interpreters, together with a set of tools that can be easily adapted to different APLs. Through this framework has been possible to rapidly prototype a range of different APLs: *(i)* AFAPL2 a reimplementation of the original Agent Factory agent programming language that is based on commitment rules; *(ii)* AF-AgentSpeak, an implementation of the AgentSpeak language based on ***Jason***; *(iii)* AF-TeleoReactive, an implementation of Nilsson's teleo-reactive programming model [Nil94]; *(iv)* AF-ASTR, an implementation of the integration of AgentSpeak(L) with Teleo-Reactive Programs, known as AgentSpeak(TR).

The framework is distributed with an open-source license and provides a practical and

efficient approach to the development of intentional agent-oriented applications. This is combined with a methodology, integrated development environment support, and a suite of tools that aid the agent development process.

Computational Language for Autonomous, Intelligent and Mobile Agents (CLAIM) and SyMPA [Sun05, FSS05, FSS03] — CLAIM is a high-level declarative agent-oriented programming language. It is part of an unified framework called Himalaya [EFSS05] (Hierarchical Intelligent Mobile Agents for building Large-scale and Adaptive sYstems based on Ambients). It combines the main advantages of agent-oriented programming languages, for representing cognitive aspects and reasoning, with those of concurrent languages based on process algebra, for representing concurrency and agent mobility [BBD+06].

CLAIM is inspired by ambient calculus [CG98] and agents are hierarchically organized, thus supporting the design of Mobile Multi Agent Systems (MMAS) – a set of connected hierarchies of agents - to be deployed on a network of computers. Every agent (i.e., a node of a hierarchy) contains cognitive elements (e.g., knowledge, goals, capabilities), processes, and sub-agents and is also mobile as it can move within its hierarchy or to a remote one.

CLAIM comes along with a well defined operational semantics describing the multi-agent system's behavior, which has been presented in [SFS07]. As an MMAS within Himalaya is deployed on a set of connected computers, the language CLAIM is supported by a distributed platform called SyMPA [SFS04], which offers all the necessary mechanisms for the management of agents, communication, mobility, security, fault-tolerance, and load balancing.

Goal-Oriented Agent Language (GOAL) [Hin09, Koe] — is a high-level language to program rational agents that derive their choice of actions from their beliefs and goals. The language provides a set of programming constructs that allow and facilitate the manipulation of an agent's beliefs and goals and to structure its decision-making. The beliefs and goals of a GOAL agent are called its mental state. Various constraints are placed on the mental state of an agent, which roughly correspond to constraints on their common sense counterparts. On top of the mental attitudes a GOAL agent also has so-called action rules to guide the action selection mechanism.

The programming constructs available in the language have a well defined formal semantics, which has been described in [Hin09]. The language comes with an integrated development environment, which allows to edit and debug MASs written in GOAL.

JACK Intelligent Agents [Age] — is a commercial framework for developing multi-agent systems which has been developed by a company called Agent Oriented Software. JACK is based on ideas of reactive planning systems resulting from the work on the BDI agent architecture and is, in this respect, similar to hybrid languages such as ***Jason***, 3APL / 2APL, and Jadex. However, instead of providing a logic-based language, JACK is an extension of Java, implementing some features of logic languages such as logical variable. A number of syntactic constructs are added to Java, allowing programmers to create plans and belief bases, all in a graphical manner

as JACK has a sophisticated IDE which provides a tool for such purpose. In JACK, plans can be composed of reasoning methods and grouped into capabilities which, together, compose a specific ability an agent is supposed to have, thus supporting a good degree of modularization.

Although JACK has no formal semantics, as a commercial platform, it has extensive documentation and supporting tools. It has been used in a variety of industrial applications as well as for research. For evaluation purposes, a free trial license can be obtained; more information is available here [Age].

An Abstract Agent Programming Language (triple-a-p-l, 3APL) [DvRM05, Mehb] — is a BDI-based programming language for implementing cognitive agents. The first version of 3APL was designed by Hindriks et al. at Utrecht University [HDBVdHM99]. Since its initial design, the 3APL programming language has been subject to continuous development [DvRM05]. One of the main features of 3APL consists of programming constructs to implement an agent's mental attitudes as well as the deliberation process that manipulates them. In particular, 3APL allows direct specification of mental attitudes such as beliefs, goals, plans, actions and reasoning rules.

A dedicated IDE allows developers to load/edit 3APL programs that implement individual agents, execute one or more agent programs in either a step-by-step or continuous fashion, monitor the internal state of individual agents and monitor the exchange of messages through the sniffer tool. The 3APL development environment, its user guide, and further documentation can be found here [Mehb].

A Practical Agent Programming Language (double-a-p-l, 2APL) [Das08, Meha] — is a BDI-based agent programming language, successor of 3APL. 2APL extends and modifies the original version of 3APL in many different ways. While the original version of 3APL is basically a programming language for single agents, 2APL is designed to develop multi-agent systems. 2APL includes all the 3APL programming constructs and adds new ones to implement a set of agents, a set of external environments, the access relation between agents and environments, and a variety of different action types such as external actions, goal related actions, and communication action.

The language comes along with two different IDEs. The first one is a standalone IDE, the second instead is a basic Eclipse-based IDE built using the Xtext framework [Ite]. Both provide the means to load, execute, and debug 2APL programs using different execution modes and several debugging/observation tools.

Among the set of existing agent-oriented programming frameworks in the state-of-the-art, in this background chapter we consider: JADE [BCG07, BPR99] and Jadex [PBL05].

Java Agent DEvelopment Framework (JADE) [BCG07, BPR99, Tel] — is a Java framework

for the development of distributed multi-agent systems. It represents an agent-oriented middleware providing a set of available services and several graphical tools for debugging and testing. One of the main objectives of the platform is to support interoperability by strictly adhering to the FIPA [Foue] specifications concerning the platform architecture [Foub, Foud] as well as the communication infrastructure [Fouc]. Moreover, JADE can be adapted to be used on devices with limited resources such as Personal Digital Assistants (PDAs) and mobile phones. JADE has been widely used over the last years by many academic and industrial organizations, ranging from tutorials for teaching support in agent-related university courses to industrial prototyping.

The JADE platform is released as open-source software, distributed by TILAB (Telecom Italia LABoratories) under the terms of the LGPL license and can be obtained here [Tel].

Jadex [PBL05, Aleb] — is a software framework for the creation of goal-oriented agents following the BDI model. The framework is realized as a rational agent layer that sits on top of the JADE middleware agent infrastructure, and supports agent development with well established technologies such as Java and XML. The Jadex reasoning engine introducing new concepts such as explicit goals and goal deliberation mechanisms (see e.g. [BPML05]), making results from goal-oriented analysis and design methods more easily transferable to the implementation phase.

Jadex has been used to build applications in different domains such as simulation, scheduling, and mobile computing. The Jadex platform, developed at the Distributed Systems and Information Systems group at the University of Hamburg, is freely available under the LGPL license and can be downloaded from here [Aleb]. Besides the framework and additional development tools, the distribution contains an introductory tutorial, a user guide, and several illustrative application examples with source code.

3.3 Programming the Environment Dimension

As introduced in Section 3.1, the notion of environment is a primary concept in the agent literature, being the place - either virtual or physical - where agents are situated, which agents are capable of sensing through some kind of sensors, and modifying through a repertoire of actions provided by some kind of effectors (Figure 3.1).

Actually two main different perspectives can be adopted when defining the concept of environment in MASs: a classical one born in an artificial intelligence context, and a more recent one grown in the context of agent-oriented software engineering [OVDPFB03].

In the classical AI view [RN09], the notion of environment is used to identify the *external world* (with respect to the system, being a single agent or a set of agents) that is perceived and acted upon by the agents so as to fulfill their tasks. A canonical representation of this perspective is shown in Figure 3.3 (adapted from [Jen01]). There, the environment is depicted as the context shared by multiple agents, each one having some kind of sphere of influence on it, i.e. that portion that they are able to (partially) control, and that could overlap with

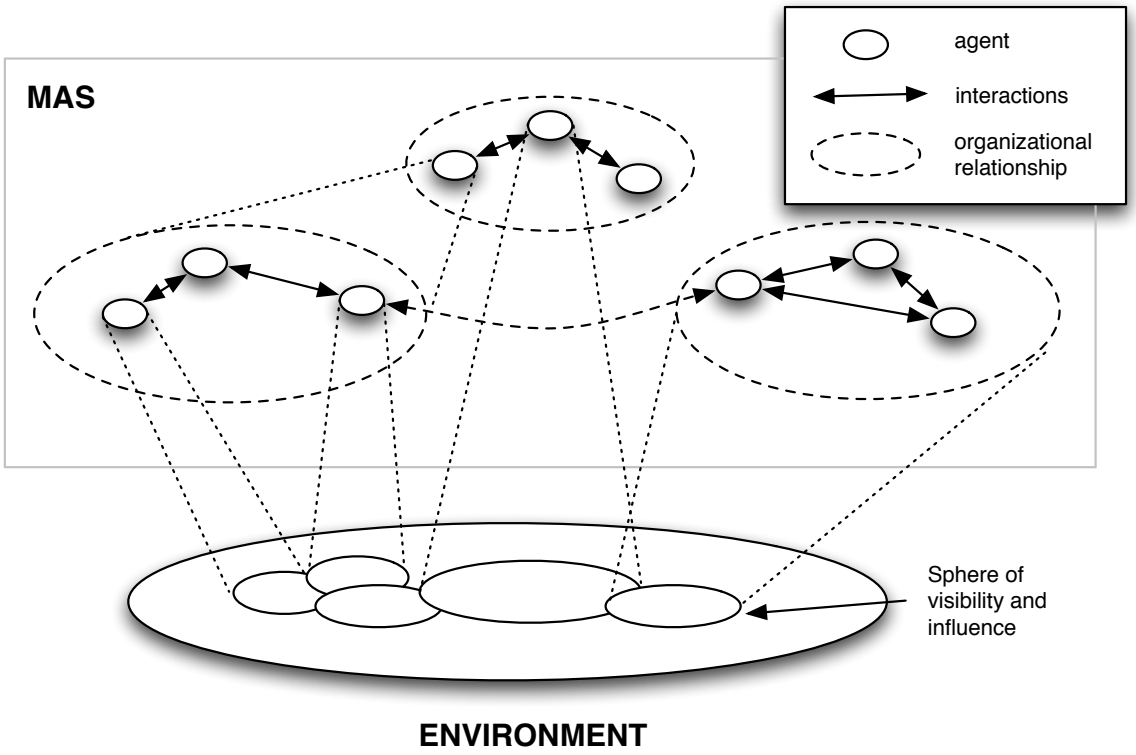


Figure 3.3: The canonical view of MAS as defined in [Jen01].

other agents sphere of influencemeaning that the environment is shared and could be jointly controlled. This first perspective is reflected by existing agent programming languages and platforms for programming MASs which typically provide some kind of API to define agent actions and perceptions implementing the interaction with some kind of external system. Quite frequently, the API also includes some kind of support for defining the structure and behavior of the environment - besides the interface - so as to set up simulations of the overall system. So, in this canonical view the environment is basically conceived as a black box, defining the set of the possible agent moves and generating perceptions accordingly.

Besides this perspective, a new one has been introduced by more recent works that investigated the important role that the environment can play in MAS engineering [WOO07]: by being the enabler and mediator of agent interactions, the environment is no longer just the target of agent actions and the container and generator of agent percepts, but a part of the MAS that can be suitably designed so as to improve the overall development of the system. So, the environment should be conceived as something that can be designed to be a good place for agents to live and work in. In other words, the idea is to *design worlds in the agents' world aimed at the agents use*. We refer to such a kind of world as *work environment* [RPVO09]. By referring to the MAS representation previously seen, work environments could be represented

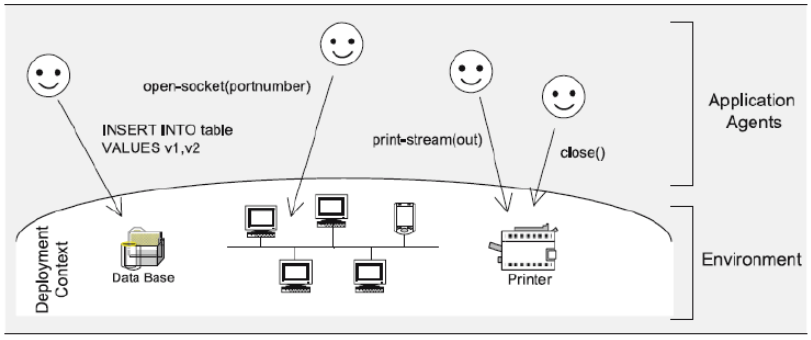


Figure 3.4: First level of support: the environment enables agents to access the deployment context.

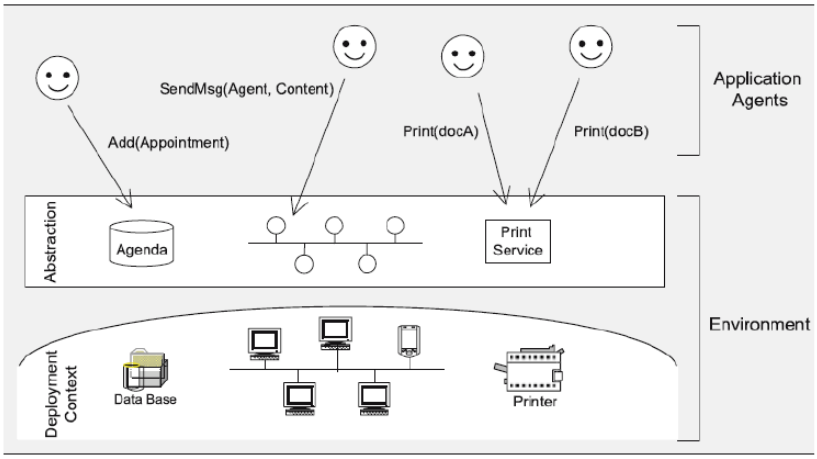


Figure 3.5: Second level of support: the environment bridges the conceptual gap between the agent abstraction and low-level details of the deployment context.

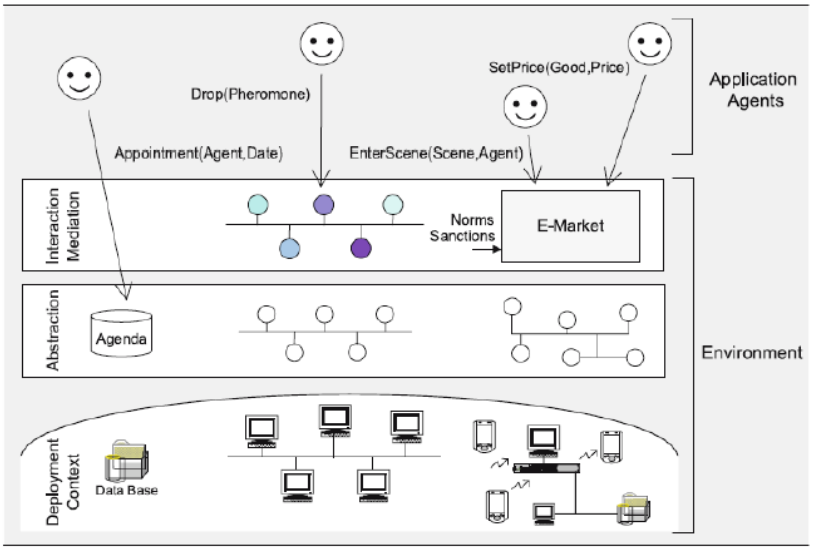


Figure 3.6: Third level of support, the environment : *(i)* regulates the access to shared resources and *(ii)* mediates interaction between agents.

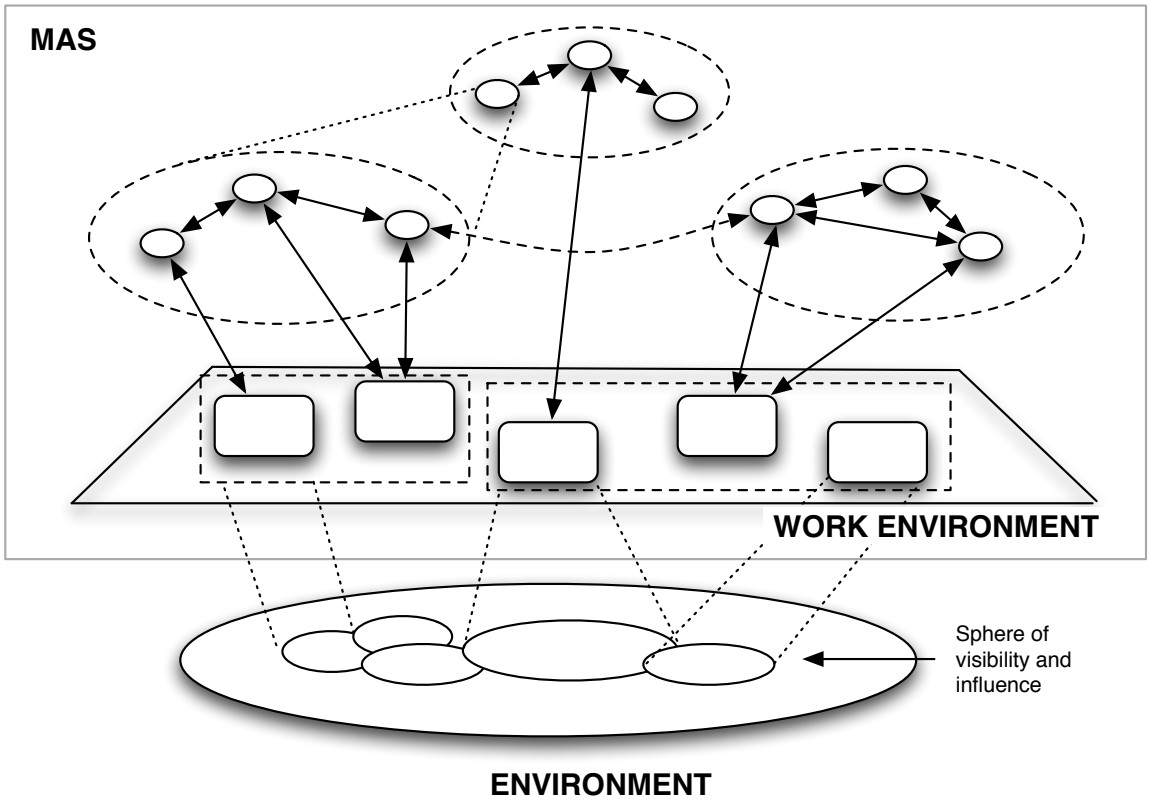


Figure 3.7: A MAS view enriched with a work environment layer.

as an extra computational layer within the MAS, conceptually placed between agents and the external environment, and mediating agent activities and interactions with the external environment (see Figure 3.7). So, the environment can be defined from the MAS and MAS engineers point of view as *endogenous*, being part of the software system to be designed—in contrast to classic AI environments which can be defined, dually, as *exogenous*.

The responsibilities and functionalities of endogenous environments can be summarized by the following three different levels of support, identified in [WOO07]: *(i)* a basic level (Figure 3.4), where the environment is exploited to simply enable agents to access the deployment context, i.e. the given external hardware/software resources which the MAS interacts with (sensors and actuators, a printer, a network, a database, a Web service, etc.); *(ii)* abstraction level (Figure 3.5), exploiting an environment abstraction layer to bridge the conceptual gap between the agent abstraction and low level details of the deployment context, hiding such low level aspects to the agent programmer; and *(iii)* interaction-mediation level (Figure 3.6), where the environment is exploited to both regulate the access to shared resources, and mediate the interaction between agents. These levels represent different degrees of functionality that agents can use to achieve their goals.

3.3.1 Programming the Environment Taking the AI Perspective

As stated in the previous section, by adopting the classical AI perspective, the environment is seen as a sort of black box defining the set of sensory information agents can perceive and the actions they can use to inspect/affect it. So, in this case programming a MAS environment basically means define the black box's interface in terms of actions and perceptions. This seems the most natural choice when dealing with either physically or simulated environment that are outside the programmers control (Figure 3.8).

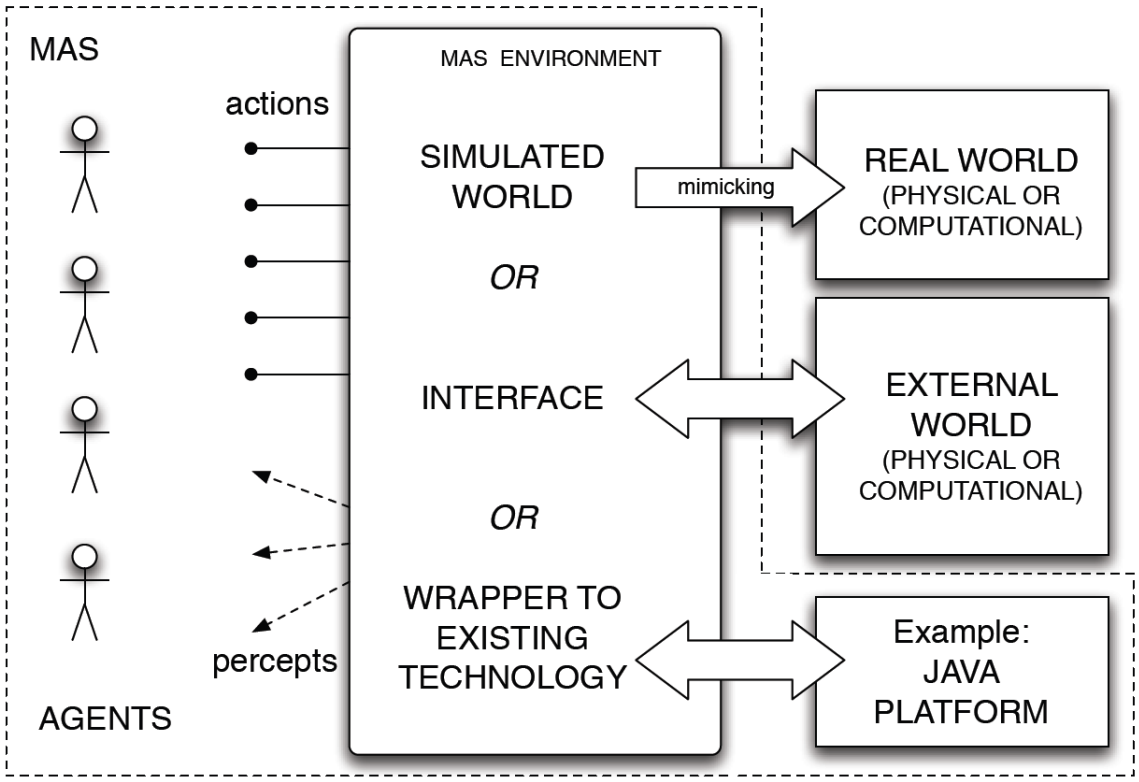


Figure 3.8: Abstract representation of the agents' interaction with exogenous environment.

Then, when developing a MAS, developers also program the environment API by defining: *(i)* the set of agents' external actions along with their computational behavior, and *(ii)* the logic behind the generation of perceptual information. Typically, such API is realized as a monolithic block by extending a predefined *"environment"* class, which is programmed in some well known mainstream object-oriented programming (OOP) language such as Java: *(i)* methods are used to implement agents' external actions—at runtime method invocation is meant to express the computational behavior associated to the external actions executed by the agents, *(ii)* class fields are used for storing the environment's state, and finally *(iii)* other specific methods are

```
public class MarsEnv extends Environment {
  private MarsModel model;
  private MarsView  view;

  public void init(String[] args) {
        model = new MarsModel();
        view  = new MarsView(model);
        model.setView(view);
        updatePercepts();
  }

  public boolean executeAction(String ag, Structure action) {
    String func = action.getFunctor();
    if (func.equals("next")) {
      model.nextSlot();
    } else if (func.equals("move_towards")) {
      int x = (int)((NumberTerm)action.getTerm(0)).solve();
      int y = (int)((NumberTerm)action.getTerm(1)).solve();
      model.moveTowards(x,y);
    } else if (func.equals("pick")) {
      model.pickGarb();
    } else if (func.equals("drop")) {
      model.dropGarb();
    } else if (func.equals("burn")) {
      model.burnGarb();
    } else {
      return false;
    }

    updatePercepts();
    return true;
  }
  ...
```

```
  ...

    /* creates the agents perception
     * based on the MarsModel */
  void updatePercepts() {

    clearPercepts();

    Location r1Loc = model.getAgPos(0);
    Location r2Loc = model.getAgPos(1);

    Literal pos1 =  Literal.parseLiteral
        ("pos(r1," + r1Loc.x + "," + r1Loc.y + ")");
    Literal pos2 = Literal.parseLiteral
        ("pos(r2," + r2Loc.x + "," + r2Loc.y + ")");

    addPercept(pos1);
    addPercept(pos2);

    if (model.hasGarbage(r1Loc)) {
      addPercept(Literal.parseLiteral("garbage(r1)"));
    }

    if (model.hasGarbage(r2Loc)) {
     addPercept(Literal.parseLiteral("garbage(r2)"));
    }
  }

  class MarsModel extends GridWorldModel { ... }

  class MarsView extends GridWorldView { ... }
}
```

Figure 3.9: Environment definition in ***Jason*** for the *"agents on MARS"* example.

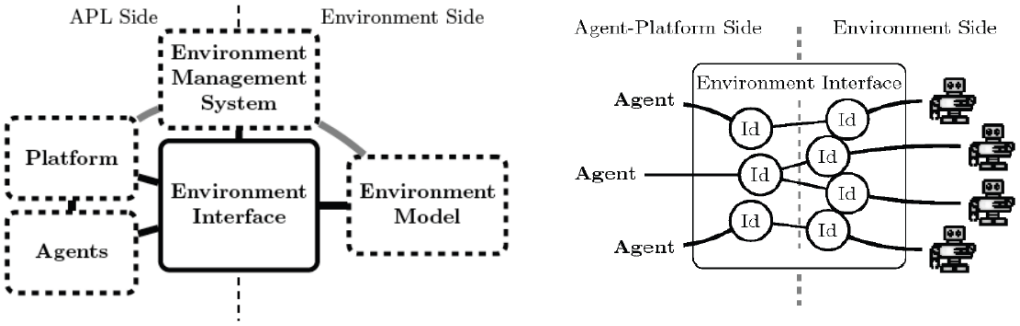


Figure 3.10: Abstract architecture of the Environment Interface Standard (EIS) in action, taken from [BHD11].

used for generating percepts. This is the standard approach adopted by default by a big majority of state-of-the-art APLs such as ***Jason***, GOAL, 2APL, etc.

To make a concrete example, Figure 3.9 shows the definition of the environment for the *"agents on MARS"* scenario, which is one of the basic examples available in the standard ***Jason***

distribution. In ***Jason***, to define a custom MAS environment, one needs to override three basic methods provided by the base `jason.environment.Environment` class:

- `init` — this is the method that is invoked when the MAS is booted in order to create the initial configuration of the environment.
- `executeAction` — this is method that defines the computational behavior of all the agents' external actions. It's here that developers have to introduce the implementation of the external actions they want to make available to the agents. In the example five different external actions are provide by the MARS environment: `next`, `move_towards`, `pick`, `drop` and `burn`. External actions execution works as follow: the ***Jason*** interpreter stores in an internal buffer the requests related to the execution of external actions and then, at each cycle, the method `executeAction` is invoked by providing in input the request on top of the buffer.
- `updatePercepts` — this is the method that defines how the environment state evolves cycle by cycle. It is possible to see how in Figure 3.9 different percepts are generated on the basis of the current position of the agents inside the simulated environment.

For what concerns the programming of environments rooted on the AI perspective, it is worth mentioning the EIS (Environment Interface Standard) initiative [BHD11], whose aim is to define a standard interface to allow agents developed using different programming languages to share the same environment, independently of the specific model and technology adopted for realizing it (Figure 3.10). So, agent platforms that support the interface can connect to any environment that implements it. This significantly reduces the efforts required by agent and environment programmers, as the environment code needed to implement the interface needs to be written only once [BHD11]. EIS works as a bridge mediating the interaction among the agents and the environment. To this end, the bridge: *(i)* introduces the notion of agent body on the agent side, *(ii)* introduces the notion of controllable entity on the environment side, and finally *(iii)* makes it possible to control environment entities by the agents via the interface.

3.3.2 Programming the Environment Taking a Software Engineering Perspective

By taking the software engineering perspective the environment is no longer some external component or black-box completely out of the control of MAS programmers (AI case/perspective), it becomes instead *endogenous*, part of the MAS itself. This perspective has led to the birth of a new research field in the MAS community, whose results have been published as proceeding of a workshop series named E4MAS [WPM05, WPM06, WPM07]. In these workshops several themes and issues were raised. The main ones were related to: theories, models, mechanisms, architectures and infrastructures for endogenous environments. The interested reader can find in [VHR+07] a good survey of all these works. Instead, for what concerns the construction of

this thesis's background we will only consider those works focusing on the study of the role of the environment abstraction for MASs programming.

Considering endogenous MAS environments, the basic idea behind the programming of a multi-agent system can now be summarized by the following equation:

programming MAS = programming Agents + programming Environment

where implicitly we refer to software MASs and endogenous environments. In this view, the environment becomes a new programming dimension, orthogonal to – but strongly integrated with – the agent one. Orthogonality means *separation of concerns*: on the one side, agents are the basic abstraction to design and program the autonomous parts of the software system, i.e. those parts that are designed to fulfill some goal/task – either individual or collective – encapsulating the logic and the control of their action. On the other side, the environment can be used to design and program the computational part of the system that is *functional* to agents' work, i.e. that agents can dynamically access and *use* to exploit some kind of functionality, and possibly *adapt* to better fit their actual needs. As a simple example, we consider the implementation inside a multi-agent program of a blackboard as a mechanism to enable uncoupled communication among agents. Without the above-mentioned separation of concerns, a blackboard must be implemented as an agent, creating then a mismatch between the design and implementation, since a blackboard is typically not designed to fulfill pro-actively and autonomously some goal, but rather to be used by other agents to communicate and coordinate. By adopting environment programming, the blackboard is implemented as an environment resource, accessed by agents in terms of actions and percepts. The example can be generalized, considering any possible computational entity properly designed to help agent work and interaction.

In the remainder of this section we provide a brief overview of the main frameworks, platforms and infrastructures that have been introduced in the state-of-the-art for programming endogenous environments. A first main example is given by CArtAgO [RPV11, RPVO09, RVO07]. CArtAgO (Common ARtifact infrastructure for AGent Open environments) is a framework and infrastructure for programming and executing endogenous artifact-based environments for MAS. Due to its relevance for this dissertation – CArtAgO is the environment framework exploited both in JaCa (Chapter 4) and JaCaMo (Chapter 5) – CArtAgO and its underlying conceptual model will be presented in the next two sub-sections. The other environment frameworks and platforms that we describe here are: GOLEM [BS08], MadKit [GF01], AGRE / AGREEN / MASQ [FMBB04, SFT09].

GOLEM [BS08] — is a logic-based framework that allows for representing an agent environment declaratively, as a composite structure that evolves over time, including two main classes of entities – *agents* and *objects* – organized in containers. Interactions between these entities inside a container are specified in term of events whose occurrence is governed by a set of *physical laws* specifying the possible evolutions of the agents' environment, including how these evolutions are perceived by agents and affect objects and other agents in the environment.

GOLEM uses a particular architecture for objects, which are described in terms of the perceived affordances—i.e., perceivable attributes of the object, and possible actions available to interact with that particular object [BS08].

The framework allows the construction of distributed environments by spreading different GOLEM containers over a network. To allow a container's affordances to be discovered within a distributed environment, the environment's object representations are translated in WSMO [RKL+05] ontologies and concepts.

MadKit [GF01] — has been one of the first general-purpose Java-based framework for developing multi-agent systems, implementing the *influence-reaction* model introduced by Ferber and Müller [FM96]. Even if not explicitly introducing a computational and programming model for the environment, in practice the framework allows for programming the environment in terms of objects embedding some computational behavior. Actually, this programming support has been exploited in particular for defining the behavior of the environment in MAS-based simulations, implemented on top of MadKit.

MadKit supports the realization of distributed MASs by manually distributing and installing the MadKit core kernel in the interested network nodes. Then, the platform provides services such as distributed message passing, agent migration and agent monitoring, which are implemented via dedicated platform agent.

AGRE / AGREEN / MASQ [FMBB04, SFT09] — these three platforms are strongly related. The first to be introduced has been AGRE. It proposes the integrating of the AGR (Agent-Group-Role) organizational model with an explicit notion of environment, which is used to represent both the physical and social part of agents interactions. Then, with the introduction of AGREEN and MASQ, AGRE has been extended toward a unified representation for physical, social and institutional environments based on the MadKit platform. The integrated platform is rooted on a model that defines four perspectives over an agent-based interaction according to two axes: internal/external and individual/collective which govern the interaction among agents, environments, organizations and institutions [SFT09].

In MASQ, distributed MASs can be developed by exploiting the functionalities of the underlying MadKit platform.

The A&A Meta-Model

Agents & Artifacts (A&A) is a conceptual (or meta) model introducing the notions of *artifact* and *workspace*, along with agents, as first-class abstractions for modeling and engineering multi-agent systems [ORV08, RVO08]. Its conceptual foundations lay upon theories and results coming from computational sciences as well as from organizational and cognitive sciences [CC95], psychology, Computer Supported Cooperative Work (CSCW) [Nar96], anthropology [PRTG00, WHSA05], and ethology [Hew93].

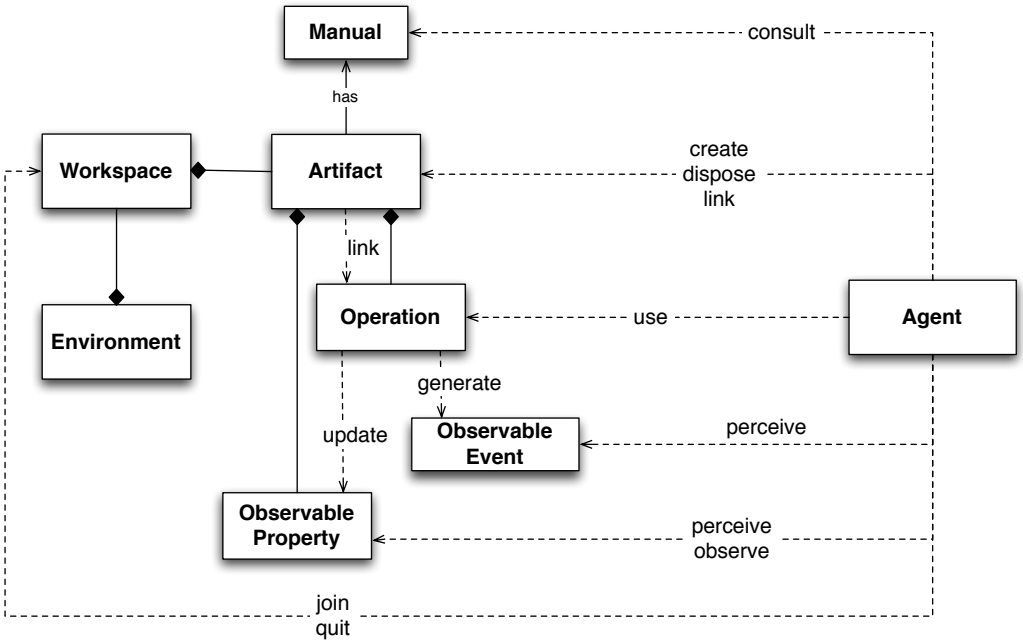


Figure 3.11: Main concepts of the A&A meta-model, here expressed in UML-like notation.

However, the main inspiration of A&A comes from Activity Theory [Nar96] – a psycho-sociological conceptual approach born in the Soviet Union at the beginning of the 20th century, further developed in northern Europe in particular – today, a main reference for HCI (Human Computer Interaction) and CSCW contexts. One of the main concepts put forward by Activity Theory – along with Distributed Cognition and other movements within cognitive science – is that, in human societies, properly designed artifacts and tools play a fundamental (mediation) role in coping with the scaling up of complexity in human activities, in particular when social activities are concerned, by simplifying the execution of tasks, improving problem-solving capabilities, and enabling the efficient coordination and cooperation in social contexts [Nor91]. In Activity Theory, the concept of tool is broad enough to embrace both technical tools, intended to manipulate physical objects (e.g., a hammer), and psychological tools, used by humans to influence other people or even themselves (e.g., the multiplication table or a calendar).

The A&A conceptual framework brings these ideas in the context of multi-agent systems, in particular for designing and programming complex software systems based on MAS [ORV08]. According to this, a MAS is conceived, designed and developed in terms of an ensemble of agents that play together within a common *working environment* not only by communicating through some high-level Agent Communication Language (ACL) (see Section 3.4), but also co-constructing and co-using using different kinds of *artifacts* organized in *workspaces* (see Figure 3.12 for an abstract representation of an A&A working environment, Figure 3.11 for an overview of the main concepts that characterize it). The environment is conceived as a dynamic

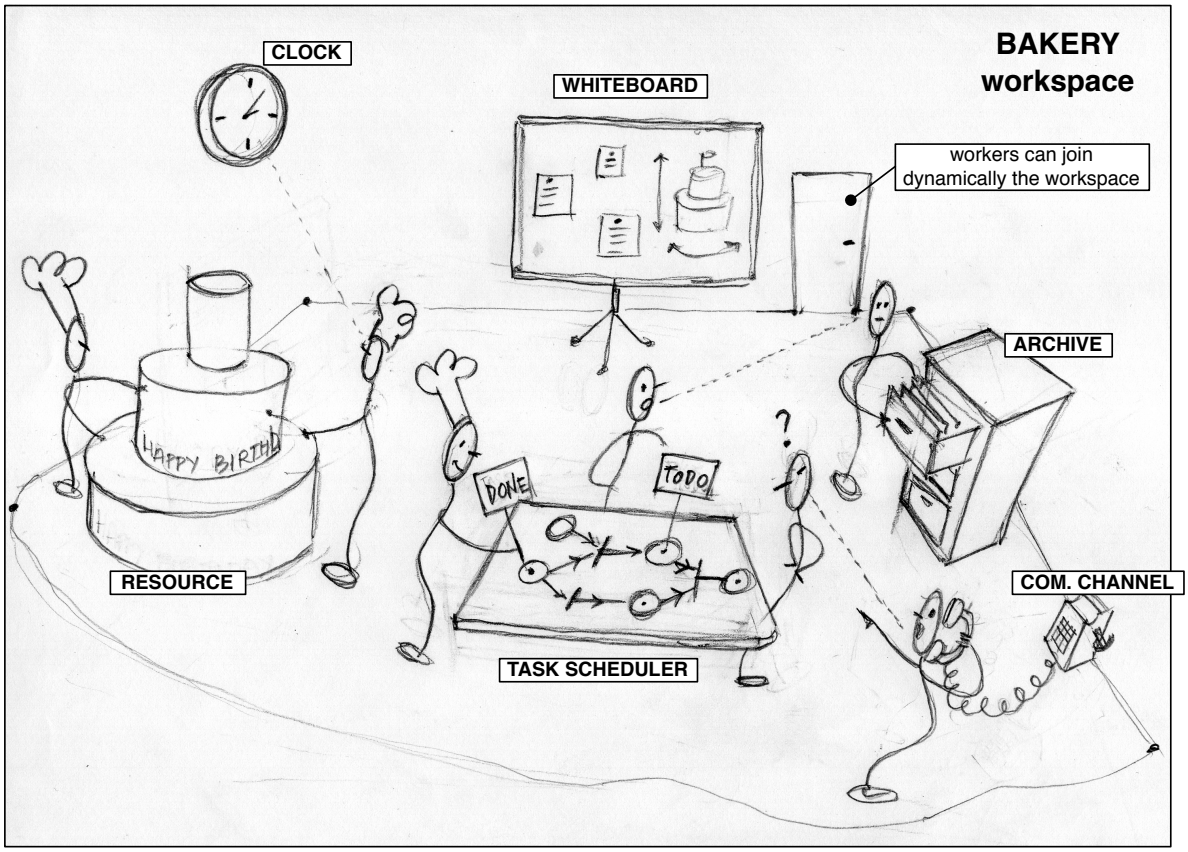


Figure 3.12: A metaphorical representation of a MAS according to the A&A meta-model.

set of computational entities called *artifacts*, representing in general resources and tools that agents working in the same environment can share and exploit. The overall set of artifacts can be organized in one or multiple *workspaces*, possibly distributed in different network nodes. A workspace represents a place, the locus of one or multiple activities involving a set of agents and artifacts.

From the MAS designer viewpoint, the notion of artifact is a *first-class abstraction*, the *basic* module to structure and organize the environment, providing a general-purpose programming and computational model to shape the functionalities available to agents. From the agent viewpoint, artifacts are the *first-class entities* structuring, from a functional point of view, the computational world where they are situated and that they can create, share, use, observe at runtime.

To make its functionalities available and exploitable by agents, an artifact provides a *usage interface* composed by a set of *operations* and *observable properties* (see Figure 3.13). Operations represent computational processes – possibly long-term – executed inside artifacts, that can be triggered by agents or other artifacts. Observable properties represent state variables

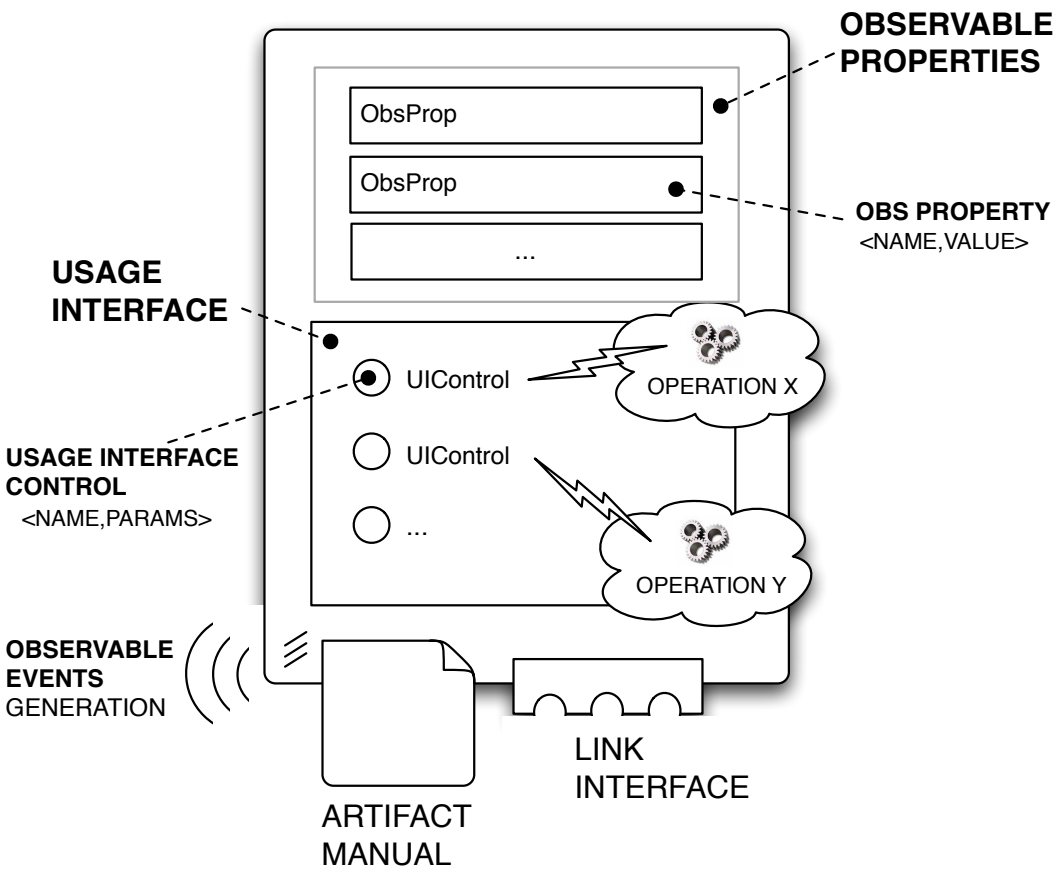


Figure 3.13: The abstract representation of an artifact in the A&A meta-model. In evidence the usage interface – i.e., artifact's operations and observable properties – the link interface and the manual.

whose value can be perceived by agents that are observing the artifact (see Figure 3.13); the value of an observable property can change dynamically, as result of operation execution. The execution of an operation can generate also *signals*, to be perceived by agents as well: differently from observable properties, signals are useful to represent non-persistent observable *events* occurred inside the artifact, carrying some kind of information. Besides the observable state, artifacts can have also an hidden state, which can be necessary to implement artifact functionalities.

From an agent viewpoint, artifact operations represent external actions provided to agents by the environment: this is a main aspect of the model. So in artifact-based environments the repertoire of external actions available to an agent – besides those related to direct communication – is defined by the set of artifacts that populate the environment. This implies that the actions repertoire can be dynamic, since the set of artifacts can be changed dynamically

by agents themselves, instantiating new artifacts or disposing existing artifacts. Observable properties and events constitute instead agent percepts.

As a principle of composition, artifacts can be linked together so as to enable one artifact to trigger the execution of operations over another – possibly distributed – linked artifact. To this purpose, an artifact can expose a link interface which, analogously to the usage interface for agents, includes the set of operations that can be executed by other artifacts, once the artifacts have been linked together by agents.

Finally, an artifact can be equipped with a manual, a machine-readable document to be consulted by agents, containing a description of the functionalities provided by the artifact and how to exploit such functionalities (that is, artifact operating instructions [VRO06]). Such a feature has been conceived in particular for open systems composed by intelligent agents that dynamically decide which artifacts to use according to their goals and dynamically discover how to use them. Actually, the notion of manual can be extended from artifacts to workspaces [RPV09]: in that case manuals may contain the description of usage protocols that can involve multiple kinds and instances of artifacts.

CArtAgO Technology

CArtAgO (Common ARtifact infrastructure for AGent Open environments) is a framework and infrastructure for programming and executing artifact-based environments for MASs (Figure 3.14), implementing the A&A conceptual model described in the previous section. It has been conceived so as to be orthogonal to the agent programming dimension, so it is – potentially – integrable with any existing agent programming language and platform. In detail, the platform includes:

- A Java-based API for programming artifacts, for defining (programming) new types of artifacts following the A&A programming model.
- An agent API on the agent side for integrating agent-oriented programming languages to CArtAgO work environments. This API provides the means to exploit all the actions required for creating and interacting with artifacts, managing and joining both local and remote workspaces, etc.
- A runtime environment and related tools, supporting the distribution and execution of work environments, managing workspace and artifact life cycles.

CArtAgO is an open-source technology implemented on top of the Java platform and released under GNU LGPL license. The full sources are freely available for download and modifications from the CArtAgO official website [Alea].

Full details about the programming of endogenous environment in CArtAgO are presented in Section 4.2.2.

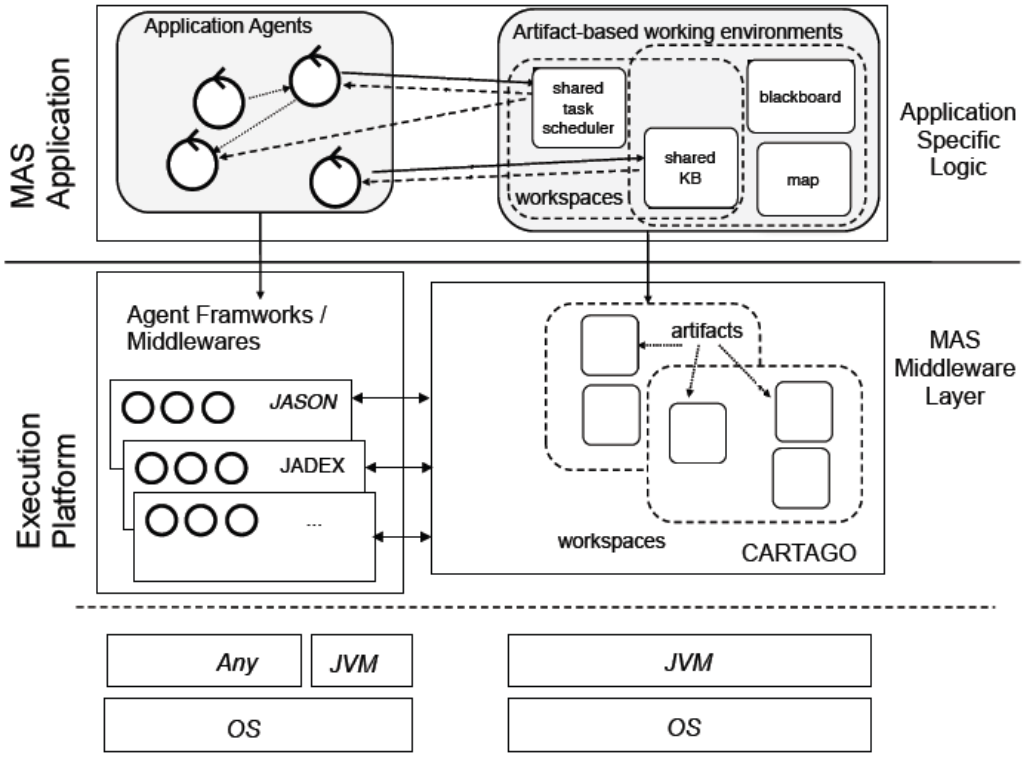


Figure 3.14: A layered representation of a MAS in execution on top of CArtAgO.

3.4 Programming the Organization and Interaction Dimensions

As introduced in Section 3.1, a key feature of agents is their *social ability*—i.e., their capability to communicate with other agents in order to exchange information, delegate the achievement of goals and tasks, take part in negotiation and interaction protocols, etc. Proper models and programming abstractions are needed in order to effectively program MASs in which agents need to interact in many complex ways, both via direct communication and possibly exploiting different social structures – i.e., organizations, electronic institutions, etc. – and norms to regulate the overall social behavior.

Accordingly, over the years there has been a proliferation of research work aiming at providing answers to these issues. In the remainder of this section we present a survey of the most relevant work in the state-of-the-art that address the aforementioned themes, separating them in work that address the programming of either the organization dimension or the interaction dimension of MASs.

3.4.1 Programming the Interaction Dimension

In multi-agent systems communication between agents is typically managed by using Agent Communication Languages (ACLs) [KSN00, LFP99a], which are based on speech-act theory, in particular the work of Austin [AUS75] and Searle [Sea69]. Speech-act theory starts from the principle that language is action—i.e., a rational agent makes an utterance in an attempt to change the state of the world, in the same way that an agent performs physical actions to change the state of the world. What distinguishes speech acts from other (non-speech) actions is that the *domain* of a speech act – i.e., the part of the world that the agent wishes to modify through the performance of the act – is typically limited to the mental state(s) of the hearer(s) of the utterance.

Speech acts are generally classified according to their *illocutionary force* – the *type* of the utterance (e.g., inform/tell, ask, achieve etc.). In natural language, illocutionary forces are associated with utterances (or locutionary acts). The *perlocutionary force* represents instead what the speaker of the utterance is attempting to obtain by performing the act. In making a statement such as *"open the door"*, the perlocutionary force will generally be the state of affairs that the speaker hopes to bring about by making the utterance; of course, the actual effects of an utterance will be beyond the control of the speaker.

In the remainder of this sub-section are briefly described the two main ACLs in the state-of-the-art: KQML [FFMM94] and FIPA ACL [Fouc].

Knowledge Query and Manipulation Language (KQML) [FFMM94] — was the first attempt to define a practical agent communication language that included speech act-based communication as considered in the distributed artificial intelligence literature. Conceptually, it is possible to identify three layers in a KQML message: *content*, *communication*, and *message*. The content layer bears the actual content of the message in the program's own representation language (KQML can carry any custom representation language). The communication layer encodes a set of features to the message that describe the lower-level communication parameters, such as the identity of the sender and recipient, and a unique identifier associated with the communication. Finally the message layer is the core of KQML: it encodes a message that one agent would like to transmit to another. This layer determines the kinds of interactions one can have with a KQML-speaking agent, by defining a set of available *performatives*.

The performative indicates whether the message sent by a sender S to a receiver R is an assertion, a query, a command, or any other of a set of known performatives. Some of the most important performatives are : *(i)* `tell`, used by S with the intention of changing (adding/updating) a belief to R's belief base; *(ii)* `achieve`, used by S with the intention of delegating to R the achievement of a new goal; *(iii)* `ask-one`, used by S to ask R the current value of a certain belief. A concrete example of KQML message follows:

```
1 (ask-one
2   :content (PRICE Google ?price)
3   :receiver stock-server
```

```
4     :language LProlog
5     :ontology NYSE-TICKS
6   )
```

With this message the sender is asking about the current price of Google's stock using the `ask-one` performative. The various components of the message represent its attributes: *(i)* `:content` specifies the message content—in this case, the content simply asks for the price of Google's stock; *(ii)* `:receiver` specifies the intended recipient of the message; *(iii)* `:language` specifies that the language in which the content is expressed is called LProlog (the recipient is assumed to `understand` LProlog), and finally *(iv)* `:ontology` defines the terminology used in the message.

KQML has been used as the communication language in several agent-oriented technologies—***Jason*** is a main example.

FIPA Agent Communication Language (FIPA ACL) [Fouc] — the Foundation for Intelligent Physical Agents (FIPA) [Foue] is an international organization that aims to develop a set of generic agent standards with the contribution of several parties involved in agent technology. In particular, the FIPA standard for ACL attempts to identify the practical components of inter-agent communication and cooperation and define a concise formal semantics and supporting communication protocols. FIPA ACL, like KQML, is based on speech act theory: messages are actions or communicative acts, as they are intended to perform some action by virtue of being sent. The FIPA ACL specification consists of a set of message types and the description of their pragmatics—i.e., the effects on the mental attitudes of the sender and receiver agents.

The specification describes every communicative act with both a narrative form and a formal semantics based on modal logic. It also provides the normative description of a set of high-level interaction protocols, including requesting an action, contract net, and several kinds of auctions. Some of the most important communicative actions available are: *(i)* `request`, to request a receiver to perform some action; *(ii)* `inform`, to inform a receiver that a given proposition is true; *(iii)* `cfp`, to make a cal for proposals to perform a given action; *(iv)* `accept-proposal`, to accept a previously submitted proposal.

The FIPA ACL specification document claims that FIPA ACL (like KQML) does not make any commitment to a particular content language. This claim holds true for most primitives. However, to understand and process some FIPA ACL primitives, receiving agents must have some understanding of Semantic Language, or SL.

Several APLs and agent-oriented frameworks use FIPA ACL as the reference communication language, by exploiting under the hood the JADE platform, which provides the reference FIPA ACL compliant implementation. In detail these languages and frameworks are: GOAL, 3APL/2APL, Jadex.

Over the years, some limitations have been advocated to agent communication languages [Sin98]—e.g., too focused on the single agent perspective, lack of a clear and effect-

ive formal semantics, poor support for interoperability among agents from different vendors, etc. Accordingly, researchers focused their efforts to push for a conceptual shift in the study of agent interactions, moving from the *individual* agent perspective to a more *social* one. This has lead to the definition of advanced forms of interaction protocols, also able to take into the account: the social context in which a certain communication takes place, the role an agent is playing inside a protocol, the set of commitments or obligations associated to a particular role, and many other aspects. For further details, the interested reader can refer to these works [Huh01, Sin96, YS02, BBM10, CS11, Sin11].

3.4.2 Programming the Organization Dimension

The organization of a multi-agent system is the collection of roles, relationships, and authority structures which govern its behavior. All multi-agent systems possess some or all of these characteristics and therefore all have some form of organization, although it may be implicit and informal. Just as with human organizations, such agent organizations guide how the members of the population interact with one another, not necessarily on a moment-by-moment basis, but over the potentially long-term course of a particular goal or set of goals [HL04]. This guidance might influence authority relationships, data flow, resource allocation, coordination patterns or any number of other system characteristics.

An organization can help groups of simple agents exhibit complex behaviors and help sophisticated agents reduce the complexity of their reasoning. Implicit in this concept is the assumption that the organization serves some purpose—i.e., that the shape, size and characteristics of the organizational structure can affect the behavior of the system. Organizations can be used to limit the scope of interactions, provide strength in numbers, reduce or manage uncertainty, reduce or explicitly increase redundancy or formalize high-level goals which no single agent may be aware of. At the same time, organizations can also adversely affect computational or communication overhead, reduce overall flexibility or reactivity, and add an additional layer of complexity to the system. So, the space of organizational options must be mapped – e.g., the different ways to redistribute agents' goal, constraint agent communications, etc. – and their relative benefits and costs understood [HL04].

A wide range of organizational models and related programming frameworks and infrastructures have emerged, each with different strengths and weaknesses. Here we provide a brief survey of the most relevant ones that have been introduced in the state-of-the-art. In particular we will focus on $\mathscr{M}$OISE – i.e., the organizational model adopted in JaCaMo (Chapter 5) – which will be described in the next sub-section. Moreover, in the remainder of this sub-section we consider two of the main other organizational models and infrastructures that have been proposed: OperA [DDM04] and 2OPL [DGMT09].

Organization Oriented Programming Language (double-o-p-l, 2OPL) [DGMT09] — is a norm-based language that allows the programming of multi-agent organizations in terms of social concepts such as norms and sanctions, monitor the actions performed by individual agents,

evaluate their effects, and impose sanctions if necessary. 2OPL is meant to be exploited in synergy with agent programming languages—2APL in particular. So far, the work has been given solid theoretical foundations but lacks a clear description of how the approach integrates with the agent level from a practical programming point of view [BBH+11].

2OPL makes distinction between the facts by which the environment is described and facts by which institutional concepts such as a violation of norms are described. More specifically, the environment where agents perform their actions is described with *brute facts*, whereas the institutional concepts such as captured violations are described with *institutional facts*. A 2OPL organization is defined by: *(i) facts*, referring to the brute facts; *(ii) effects*, which describe environment's modifications related to the actions performed by the agents; *(iii) counts-as rules*, which ascribe *institutional facts* (e.g., a violation has occurred); and *(iv) sanctions rules*, which determine which brute facts will be brought about by the system as a consequence of the violation of some normative fact. The institutional facts instead are derived by the organizational rules from the brute facts when the organization is running. They are never manually programmed beforehand by the organization programmer. Therefore, 2OPL organization model gathers all the organizational and environmental information under five components, and every component is either a collection of facts or a collection of rules. This information is used in a control cycle to evaluate and manage every agent action.

In 2OPL, norm violations can be handled in two ways. Regimentation is making the violation of norms impossible for agents. It means blocking the action that causes a regimented violation completely. Enforcement is allowing the violation of norms first and then sanctioning the actors of the violation. While regimentation makes agents less autonomous, enforcement allows agents to get outside the boundaries of norms on their own will. It is not always possible to use one instead of the other. They must both be employed carefully in order to preserve the autonomy of agents and make them work together at the same time.

OperA [DDM04] — is a model for organizations that supports individual initiative and collaboration while prescribing a formal structure for organizational processes. The OperA model integrates a top-down specification of society objectives and global structure, with a dynamic fulfillment of roles and interactions by participants. The model separates the description of the structure and global behavior of the domain from the specification of the individual entities that populate the domain. On the one hand, coordination and interaction in multi-agent systems are described in the context of the actions and mental states of individual agents. On the other hand, society models designed from an organizational perspective reflect the desired behavior of an agent society, as determined by the *society owners*. Once agents populate the society, their own goals and behavior will affect the overall society behavior, that is, such social order as envisioned by the society designer is in reality no more than conceptual; abstract behavior seldom is realized exactly in practice [DDM04].

The OperA framework consists of three interrelated models. The organizational structure of the society, as intended by the organizational stakeholders, is described in the organizational model (OM). The way interaction occurs in a society depends on the aims and characteristics

of the application, and determines the way in which roles are related with each other, and how role's goals and norms are *"passed"* between related roles. The agent population of an OM is specified in the social model (SM) in terms of social contracts that make explicit the commitments regulating the enactment of roles by individual agents. Social contracts describe the capabilities and responsibilities of an agent within the society, that is the desired way that an agent will fulfill its role(s). Finally, given an agent population for a society, the interaction model (IM) describes possible interactions between agents.

In order to support developers in designing and maintaining organization models, an organization-oriented development environment named OperettA [OD08] is provided.

Programming Organizations in Moise

M OISE provides both an organizational model [HSB07], and a concrete framework/infrastructure for programming and executing MASs organizations [HBKR10]. The *M* OISE organizational model explicitly decomposes the specification of a multi-agent system organization into *structural*, *functional*, and *deontic* dimensions (Figure 3.15). The structural dimension specifies the *roles*, *groups*, and *links* of the organization. The definition of roles states that when an agent decides to play some role in a group, it is accepting some behavioral constraints related to this role. The functional dimension specifies how the *global collective goals* should be achieved, i.e. how these goals are decomposed (in global *plans*), grouped in coherent sets (by *missions*) to be distributed to the agents. The decomposition of global goals results in a goal-tree, called *scheme*, where the leaf-goals can be achieved individually by the agents. The deontic dimension is added in order to bind the structural dimension with the functional one by the specification of the roles' *permissions* and *obligations* for missions.

As an illustrative example of an organization specified using *M* OISE, we consider agents that aim at writing a paper and therefore have an organizational specification to help them collaborate. The structure of this organization has only one group (`wpgroup`) with two roles (`editor` and `writer`) that inherit all the properties defined for the role author. The cardinalities and links of this group are specified using the *M* OISE notation [HSB07] in Figure 3.16 (*a*): the group `wpgroup` can have from one to five agents playing `writer` and exactly one playing `editor`; the `editor` has authority over `writer` and every agent playing `author` (and by inheritance everyone playing `writer` or `editor`) has the possibility to communicate with every agent playing `author` (there is a communication link from `author` to `author`). In this example, the `editor` and the `author` roles are not compatible. To be compatible, a compatibility relation must be explicitly added in the specification.

To coordinate the achievement of the goal of writing a paper, a scheme is defined in the functional specification of the organization (Figure 3.16 (*b*)). In this scheme, a draft version of the paper has to be initially defined (identified by the goal `fdv` in Figure 3.16 (*b*)). This goal is decomposed into three sub-goals: write a title (`wtitle`), an abstract (`wabstract`), and the section titles (`wsectitle`). The agents then have to *"fill"* the paper's sections to get a submis-

sion version of the paper (identified by the goal `sv`). The goals of this scheme are distributed in three missions which have specific cardinalities (Figure 3.16 (*c*)): the mission `mMan` is for the general management of the process (one and only one agent can commit to it), mission `mCol` is for the collaboration in writing the paper's content (from one to five agents can commit to it), and mission `mBib` is for getting the references for the paper (one and only one agent can commit to it). A mission defines all goals an agent commits to when participating in the execution of a scheme, for example, a commitment to the mission `mMan` is indeed a commitment to achieve four goals of the scheme. Goals without an assigned mission are satisfied by the achievement of their subgoals.

The deontic relation from roles to missions is specified in Figure 3.16 (*d*). For example, any agent playing the role `editor` is permitted to commit to the mission `mMan`, while instead agents playing the role `writer` are obliged to commit to mission `mMan` and `mBib`, following the designated cardinalities.

The specification of an organization is written in a suitable language [Hüb03], that the agents are supposed to interpret. The runtime execution, functioning and evolution of the organization is managed by a dedicate organizational infrastructure [HBKR10].

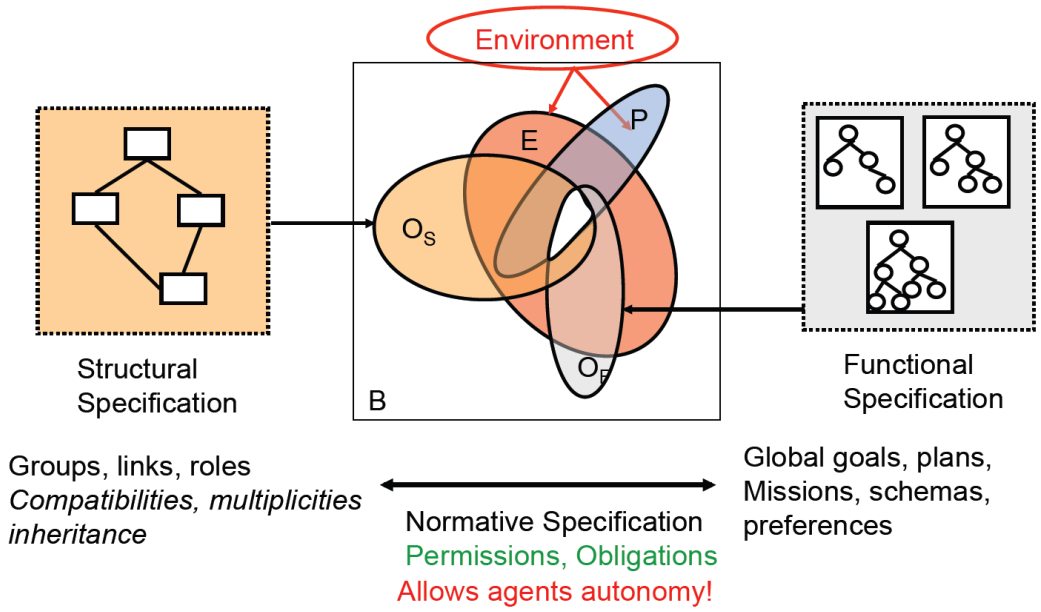
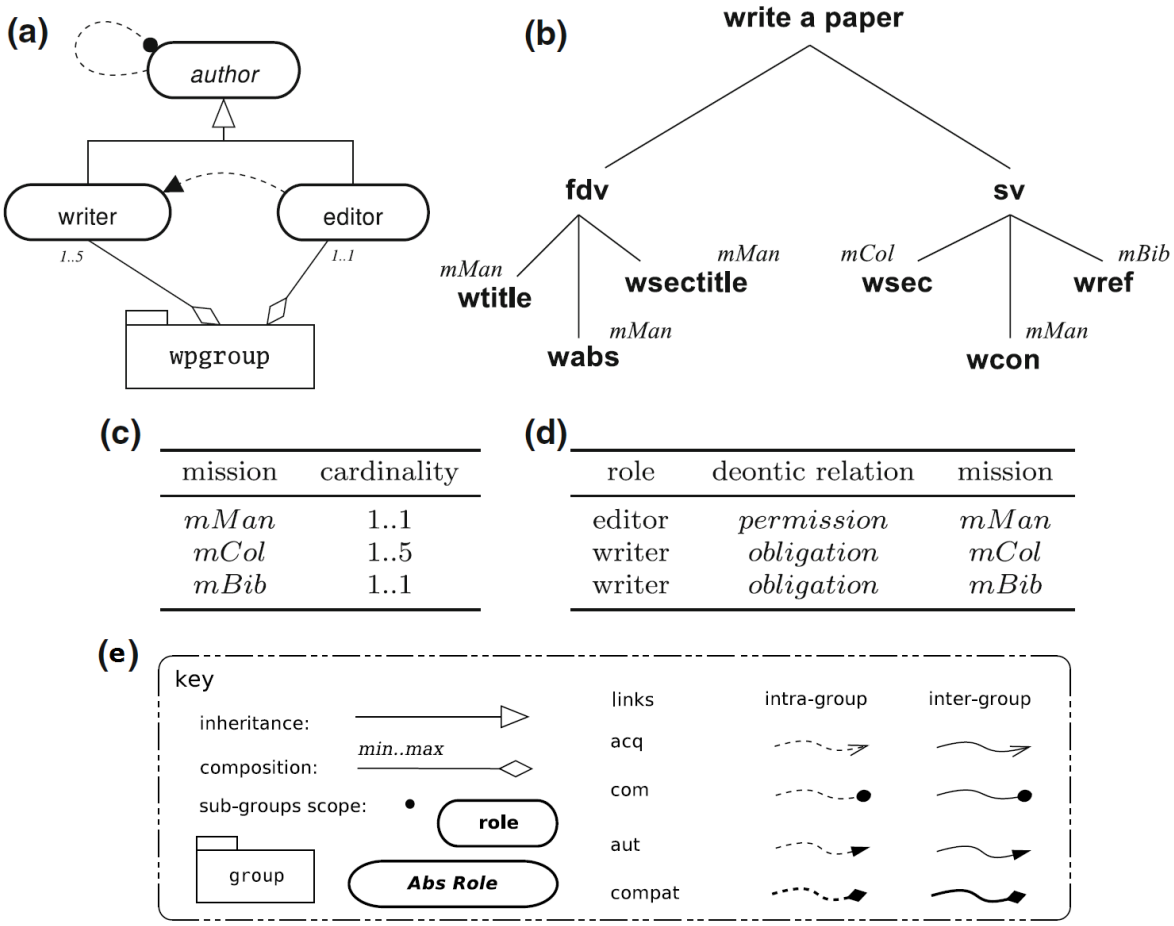


Figure 3.15: An abstract representation of a $\mathscr{M}$OISE organization with in evidence the functional, structural and deontic dimensions [HSB07].



mission	cardinality
$mMan$	1..1
$mCol$	1..5
$mBib$	1..1

role	deontic relation	mission
editor	*permission*	$mMan$
writer	*obligation*	$mCol$
writer	*obligation*	$mBib$

Figure 3.16: Graphical representation of the organizational specification for the writing paper example: (*a*) structural specification, (*b*) functional specification, (*c*) mission cardinalities (part of functional specification), (*d*) deontic specification, (*e*) legend for the symbols used in the structural specification.

3.5 Concluding Remarks

As introduced in Section 3.1, at least four separate communities within the multi-agent research community have been dealing with specific dimensions of MASs development, namely the research communities interested in: agent-oriented programming languages [Sho93, BDDFS05b, BDDFS09], interaction languages and protocols [Sin98, LFP99b, com00], environment architectures, frameworks and infrastructures [VHR+07], and multi-agent organization management systems [GF01, HL04, ERARA04, HBKR10, DDM04, DGMT09].

The results produced by these communities have clearly demonstrated the importance of

each dimension for the development of multi-agent systems. Nevertheless, perhaps a bit surprisingly, currently the engineering of such systems is still hampered by the usage of programming approaches and related development platforms that are mainly focused in just *one* specific dimension, fully orthogonal w.r.t. other ones. From the one side, providing full orthogonality gives an undeniable advantage: once realized, a particular model or technology becomes *potentially* exploitable or integrable with any other one. Moreover, as an indirect consequence, its diffusion and adoption can be increased by giving other researchers the opportunity to exploit it in combination with their own work.

On the other side, full orthogonality might severely reduce the opportunity to exploit the synergies that can emerge from the integration of the different dimensions when programming the MAS as a whole. In particular, this means that developers currently have only ad-hoc programming solutions for the engineering of complex multi-agent systems – typically rooted on low-level implementation mechanisms to integrate the different dimensions – without suitable high-level abstractions and tools. We argue that such a kind of an ad-hoc integration could be bothersome from an implementation – e.g., dealing with different implementation styles or architectures – conceptual – e.g., dealing with abstractions clashes/mappings – and programming point of views—e.g., need to take into the account integration issues when coding, hence making MAS programming more difficult.

The benefits of adopting a comprehensive approach – integrating different dimensions – have been recognized in the context of agent-oriented software engineering and MAS modeling communities quite a long time ago [FG98, Dem95, SFT09]. Accordingly, in the next part of this dissertation, we bring such a perspective down to the programming level, investigating concrete programming models and platforms preserving a strong separation of concerns but, at the same time, exploiting MASs programming dimensions in a synergistic way. As mentioned in the introduction, this is one of the two main contributions of this thesis work.

Part II

Engineering Agent-Oriented Technologies for Programming Multi-Agent Systems

4
The JaCa Platform

In this chapter we present JaCa, a general-purpose programming platform that can be used for developing MASs in general, by exploiting both the agent and the environment programming dimensions in synergy. Actually, JaCa is not an entirely new platform, but the integration of two existing agent programming technologies that have been already introduced in Chapter 3: *(i)* the ***Jason*** agent programming language – for programming BDI agents – and *(ii)* the CArtAgO framework—for programming and running the endogenous environments where agents work. The main novelty of the platform – and also its main feature – is the synergistic integration of the two agent technologies considered, and in particular the assumptions and simplifications that this makes possible – both from the agent and environment side – when programming a MAS. This chapter is organized as follows:

- First in Section 4.1 is presented the keystone on top of which the JaCa platform is rooted: the new action and perception model [RSP12] that has specifically conceived to make BDI agents work in endogenous environments, shifting from the classical models applied in state-of-the-art APLs. The definition of this model, is part of the contributions of this thesis work.
- Then are described both the JaCa platform and its underlying programming model (Section 4.2 and Section 4.3).
- Finally, the chapter is concluded by discussing the concrete use of JaCa for programming MASs in some of the most relevant application domains (Section 4.4, Section 4.5 and Section 4.6).

4.1 An Effective Action and Perception Model for BDI-based APLs Working with Endogenous Environments

In this section is described the new action and perception model that we have conceived for making BDI agents work with endogenous environments, which has been presented in [RSP12]. To better motivate and clarify the needs and the benefits related to its adoption in the case of

endogenous environments, in the description we also consider a comparison with the action and perception models adopted in current BDI-based APLs, which are instead devised for dealing with exogenous environments. The models adopted in ***Jason***, 2APL and GOAL are taken as reference case studies for the comparison.

By referring to existing formalizations, these languages follow the abstract reference architecture for intelligent agents and the practical reasoning agent cycle reported by Wooldrige in [Woo09], and which has been described in Section 3.2.1. Essentially such a control loop can be summarized as a *sense-plan-act* cycle where the agent repeatedly: *(i)* observes the environment and updates its beliefs (sense stage), *(ii)* uses practical reasoning to deliberate what intention achieve and how (plan stage), and *(iii)* executes a proper plan for fulfilling the selected intention (act stage). The environment – software or hardware – here is fully exogenous. Actually, moving from formal models to concrete architectures and implementations, current APLs adopt richer approaches and semantics which are explicitly oriented toward the integration with forms of endogenous environments.

A comprehensive survey of the environment interface models adopted by mainstream APLs and the APIs for interacting with them can be found in the EIS initiative report [BHD11]. In the following we focus on the semantics underlying the action and perception model.

4.1.1 The Action Model

The action model concerns how agents affect the state of the environment – hence, the very notion of external action – and includes what kind of semantics is adopted for defining action success/failure and which action execution model is used. In the abstract agent architecture, the action chosen by the agent action selection function is dispatched to effectors which will eventually execute it (act stage) and the control cycle can start again (sense stage). Actions are considered as options in agents repertoire which can be translated by moves enabled by the environment. The success or failure of the action executed by the agent effectors is meant to be determined by the agent itself, by analyzing the percepts that will eventually be observed from the environment. From the execution model point of view, action execution is modeled then as an *atomic event*, which corresponds to dispatching the action to effectors. This semantics is the basic one adopted by a big majority of APLs formal models, concrete examples are ***Jason*** and GOAL[1].

We argue that the action model just presented is not the most effective one when considering endogenous environments because it leads to two main drawbacks. The first one is that, by treating actions as events, it is not possible to implement *concurrent actions* – i.e., actions overlapping in time, performed by different agents – which are an effective way to realize synchronization and coordination mechanisms. E.g., a tuple space providing `in` and `out` coordination primitives as defined by the Linda language [Gel85]. In an agent perspective, we would model such primitives directly as actions that the environment provides to agents to coordinate,

[1] Actually, the concrete implementations of these APLs adopt a bit more complex solutions than the one presented (however, the intended semantics is not modified), for full details see [RSP12].

in which the execution of an `in` suspends the current agent plan – not the agent execution itself, which must go on with the execution cycle – until, for instance, a tuple is inserted by another agent executing an `out`. By treating actions as events, this is not possible and one is forced to change the semantics of the `in` action, for instance assuming that the successful execution of the action represents just the act of making the request, not the fact that a tuple has been removed.

The second related drawback concerns, more generally, the success/failure semantics of actions. Having that an action has been correctly accepted by the environment *does not imply anything about its actual completion and effects*, hence, the programmer is forced to check manually in agents code percepts coming from the environment in order to determine if an external action has been completed with success or not. This is burdensome, both from a programming point of view and for the performances of the agent program.

2APL apparently adopts a stronger semantics for action success and failure, not only at the implementation level but also in the formal model. Following [BHD11], executing an action-method in 2APL can have two outcomes: either a return-value (an object) indicating success is returned, that might be non-trivial (e.g., list of percepts in the case of a sense-action) or terminate with an exception indicating action-failure. In this case success means that the action completed with success—and so the effects of the action took place, so a stronger semantics with respect to ***Jason*** and GOAL. However, analogously to ***Jason*** and GOAL formal models, action execution is modeled and finally implemented as an atomic transition (so an event) coupling the agent and the environment. This means that by executing an action, the agent cycle is blocked until the completion of the action with success or failure occurs, and this can have drawbacks on agent reactivity.

In the case of endogenous environments it is possible to define and exploit a richer action model. The success (or failure) of an action on the agent side can be directly related to the successful (or failed) completion of an operation executed on the environment side – which has been designed by the MAS engineers – as a consequence of the agent action request. So, differently from exogenous environments, in endogenous environments action success/failure can be represented by an explicit *action completion event* generated by the environment, thereby an explicit information related to the completion of operation execution (with success or failure). Accordingly, from the APLs point of view the execution of an action does not mean that the action has been simply accepted or recognized by the environment, but that *the related environment operation has been executed up to completion*. This means seeing the set of actions as a sort of *contract* provided by the environment, including both the effects that can be assumed with action completion and the *action feedbacks*, including further information related to action success or failure. Action feedbacks are an effective way, in particular, to represent results computed by the action—so information which are not suitably modeled as effects over the environment. A simple example is given by actions performing just pure calculation.

By assuming this semantics, agent programs become – generally speaking – more terse and efficient. Action completion events are meant to be automatically processed by the agent architecture, in order to – for instance – reactivate the suspended plan where the action was

executed, without the burden for the agent programmer to manage such percepts by hand. Also, the contract makes it simpler for agents to reason about the state of the environment: agents can appraise step by step their course of actions and by completing an action, an agent is sure that the effects possibly specified for the action in its specification took place.

From the action execution model point of view, our action model promotes an *action-as-a-process* semantics, where actions are not modeled as single atomic events but as processes that can be long-term, whose completion is notified by action completion events. When adopted in APLs, this semantics have two main benefits. First, it makes it possible to effectively program agents that execute (long-term) actions without hampering their reactivity: the action-as-process semantics makes it possible for an agent to start the execution of an action whose completion occurs asynchronously – w.r.t. agent execution – by the reception of an explicit action completion event. Second, the action-as-process semantics makes it possible to implement efficient *coordination mechanisms* simply based on actions synchronization, designing environments which provide operations for that purpose. This because the action completion event of an action performed by a certain agent can be generated as a consequence of the execution of the action(s) of other agents in the same environment. Recalling the example of tuple spaces and Linda, blocking actions like `in` or `read` can be implemented quite straightforwardly by adopting an action-as-process semantics. In particular, an `in` action can start its execution before the execution of an `out` action and then complete after the `out` completion.

4.1.2 The Perception Model

The perception model concerns how the environment can be perceived by agents, the definition of the stimuli generated by the environment and the corresponding agent percepts as result of the perception process. Along with actions, these can be considered part of the contract provided by the environment as well. In the abstract intelligent agent architecture [Woo09], agent perception is modeled by a *see* function: $E \rightarrow Per$. This function encapsulates the agent ability to obtain information from the environment E in which it is situated. The output produced by this function is a percept *Per*, that is elaborated by the agent through appropriate belief-update/revision functions, to keep its mental state consistent with the actual state of the environment.

Essentially, in literature two basic semantics can be adopted when defining the perception model in exogenous environments, that we refer here as *state-based* and *event-based*. In the former case, stimuli are information about the *actual state* of the environment and are generated when the agent is engaging the perception stage of its execution cycle. In the latter, stimuli are information about *changes* occurred in the environment, dispatched to agents when such changes occur, independently from agents execution state. Referring to concrete agent programming languages and their formal operational semantics, a state-based approach is adopted – for instance – in ***Jason*** and GOAL, while instead a concrete example of APL treating percepts directly as events is 2APL.

Like in the case of the action model the chosen semantics for percepts and perception can have a strong impact on the dynamics of MASs execution. In particular the state-based approach

suffers of some problems that are particularly important when working with endogenous environments. A first problem concerns the possibility for an agent of *losing* – not perceiving – environment states that could be relevant for agent reasoning and course of action. This can occur due to environment dynamics, related to internal processes and also actions performed by other agents, changing asynchronously the environment multiple times between two subsequent perceive stages. A second problem is that retrieving perceptual information at each cycle regarding the current observable state of the environment can be a task that requires high computational complexity, in particular when considering non-naive environments, being either centralized or, worst, distributed.

In endogenous environments, modeling percepts as events – carrying on information about changes occurred in the environment – is more effective and efficient than modeling percepts as facts about the actual state of the environment itself. At first it makes it possible – in principle – to avoid the first problem described above. Referring to the abstract agent architecture, the set of percepts returned by the *see* function represent a list of changes occurred in the environment during the last agent execution cycle and which are relevant for the observing agent. Accordingly, the current internal state of the agent can be updated with respect to the *whole* set of changes occurred inside the environment, thus eventually reconstructing all the intermediate states that the environment assumed between a couple of *see* activities.

However, with this approach, in state-of-the-art APLs – e.g., 2APL and also in GOAL which however adopts a state-based approach – in order to keep track of the observable state of the environment, the programmer is forced to explicitly define the rules that specify how to change the agent's internal environment representation when a new percept is detected. This is a indeed very important capability when dealing with exogenous environments; when adopting endogenous environments – in particular complex ones – this can become burdensome. Actually, being specifically designed by MAS engineers endogenous environments allow for a stronger assumption on the relationships between percepts generated by the environment and the related agent internal representation. Such assumption can be used then to define a mapping between percepts and beliefs, *to be applied by default by the agent architecture*, so, on the one side avoiding the burden to the agent programmers of necessarily specifying the percept rules, and on the other side automatically keeping consistency between the actual state of the environment and the belief base. Then, to support the automated reconstruction of such states it is useful to identify the basic set of possible kinds of event that can occur inside an endogenous environment: *(i)* an observable part of the environment has changed; *(ii)* an observable part of the environment has been added or removed; *(iii)* a signal has been generated to acknowledge agents with some information. In the latter case, signals are meant as information explicitly generated by the environment – as designed by environment programmers – to carry on some data which can be purposefully processed by agents situated in the environment and focusing that part which is the source of the signals. By explicitly defining a model to represent environment observable parts – e.g., observable properties in the case of artifact-based environments Section 3.3.2 – it is possible then at the agent architecture level to automatically reconstruct a consistent snapshot of the current observable state of the environment by processing a list of events updating the

previous snapshot. In the APLs considered here this means introducing in the basic architecture a support for: *(i)* representing the observable part of the endogenous environments as beliefs, and *(ii)* automatically updating such beliefs as soon as such events are processed. In this perspective, there is no more the need for an agent programmer to explicitly define belief update functions (such as in 2APL) or percept rules and post-condition in actions specification (such as in GOAL): the belief base is automatically updated reflecting the perceived/reconstructed state of the observed environment.

Recalling the second problem described above, due to concurrency and distribution, the correct and efficient reconstruction of the observable state of the environment from the individual agent architecture perspective is an issue – both from the theoretical and practical point of view – even when adopting an event-based perception model. First, by working with multi-agent systems, we must assume that multiple agents can concurrently work in the same environment and then events generated concern concurrent processes. Second, environments can be distributed, which means that it is not feasible to consider a priori the availability of a unique notion of time – either physical or logical – and then a total order among events. In order to cope with these two aspects, first it is useful to conceive a distributed endogenous environment as a set of non-distributed sub-environments, eventually connected, and assume that each sub-environment defines a spatial-temporal locality. For each sub-environment it is feasible then to assume that: *(i)* a local logical notion of time can be defined, and *(ii)* observable events occurring the in the sub-environment can be totally ordered using logical timestamps, even if they are generated by concurrent processes. Given this assumption, agents perceive chains of events, which are totally ordered if the source is a single sub-environment, partially ordered if more sub-environments are involved. Then, some modularization strategy should be considered for structuring individual sub-environments, so as to *(i)* allow multiple agents to work concurrently in different parts of the overall structure, promoting as far as possible decentralization and parallelism; *(ii)* make it possible to easily change structure at runtime, eventually changing/extending dynamically the set of sub-environments available, so to better support openness, adaptation, etc.

4.2 Programming Multi-Agent Systems in JaCa

The JaCa programming model is rooted on the synergistic integration – supported by the action and perception model presented in the previous section – of concepts defined in the A&A conceptual model (Section 3.3.2) and the BDI agent model (Section 3.2.1). Accordingly, a JaCa program is conceived as a dynamic set of autonomous BDI ***Jason*** agents working cooperatively inside a shared CArtAgO working environment, whose topology is structured in terms of – possibly distributed – workspaces. The environment is composed by a dynamic set of artifacts, as computational entities that agents can dynamically create and dispose, besides using and observing them. Programming the application means then programming the ***Jason*** agents on the one side, encapsulating the logic of control of the tasks that must be executed, and the CArtAgO working environment on the other side, as a first-class abstraction providing the actions and

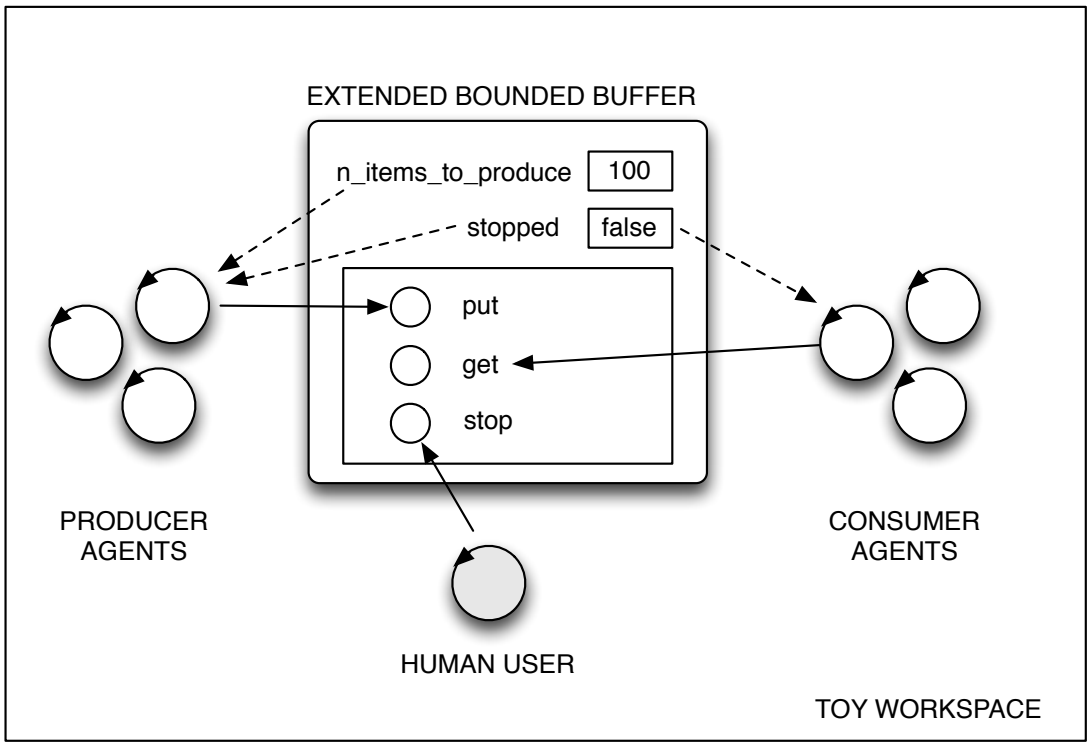


Figure 4.1: An abstract representation of the architecture of the producer-consumer example. In evidence producer and consumer agents interacting by means of the ExtBBuffer artifact.

functionalities exploited by the agents to do their tasks. Agents cooperate and interact by means of both direct verbal communication through the KQML agent communication language (Section 3.4.1), and coordination artifacts available in the environment [ORV+04, ROD03] (e.g., tuple spaces, bound buffers, etc.).

In the remainder of this section we describe how a generic MAS can be developed in JaCa using a concrete example, which is about the implementation of a slightly revised producer-consumer architecture. Like in any usual producer-consumer scenario, we have a set of producer agents that continuously and concurrently produce data items which must be consumed by consumer agents (Figure 4.1). In addition, to make the example more interesting, we have some further requirements: *(i)* the number of items to be produced is fixed, but the time for producing each item (by the different producers) is not known a priori; *(ii)* the overall process can be interrupted by the user anytime.

The task of producing items is divided upon multiple producer agents, acting concurrently—the same holds for consumer agents. To interact and coordinate the work, agents share and use an artifact, the ExtBBuffer (Extended Bounded Buffer) artifact, which functions both as a bounded buffer to collect items inserted by producers and to be removed by consumers and

as a tool to control the overall process by a human user. The artifact provides on the one side operations (actions for the agent) to insert (`put`), remove (`get`) items and to stop the overall activities (`stop`); on the other side, observable properties `n_items_to_produce` and `stopped`, keeping track of, respectively, the number of items still to be produced (which starts from an initial value and is decremented by the artifact each time a new item is inserted) and the stop flag (initially `false` and set to `true` when the stop operation is executed).

In the following, first we give some glances about agent programming in JaCa by discussing the implementation of a ***Jason*** producer agent (Section 4.2.1), which must exhibit a pro-active behavior - performing cooperatively the production of items, up to the specified number - but also a reactive behavior: if the user stops the process, the agents must interrupt their activities. Then we briefly consider the implementation of the ExtBBuffer artifact (Section 4.2.2), and finally of the multi-agent system in the overall (Section 4.2.3).

4.2.1 Programming the Agents

Being inspired by the BDI architecture, the ***Jason*** language constructs that programmers can use can be separated into three main categories: *beliefs*, *goals* and *plans*. An agent program is defined by an initial set of *beliefs*, representing the agent initial knowledge about the world; a set of *goals*, the objectives that the agent wants to bring about; and a set of *plans* that the agent can dynamically compose, instantiate and execute to achieve such goals. All these information are specified by the agent programmer. Logic programming is used to uniformly represent any piece of data and knowledge inside the agent program, beliefs and goals in particular.

Beliefs are represented as Prolog [SSE86, CM03] like facts – that are atomic logical formulae – and represent the agent knowledge about:

- Its internal state — an example is given by the `n_items_produced(N)` belief, which is used by a producer agent to keep track of the number of items produced so far. Initially `N` is zero, and then it is dynamically updated by the agent in plans, by means of specific internal actions.

- The observable state of the artifacts that the agent is observing — in the example, every producer agent observes the ExtBBuffer artifact (line 14, the meaning of the `focus` operation will be clarified in the following), which has two observable properties: `n_items_to_produce`, representing the number of items still to be produced, and `stopped`, a flag which is set if/when the process needs to be stopped.

At design time the agent developer may want to define the agent initial belief base, by specifying some initial beliefs: then, beliefs can be added or removed at runtime, according to the agent changes to its state and – following the perception model described in Section 4.1.2 – to the resources that the agent dynamically decides to observe.

An agent program may explicitly define the agent initial belief base and the initial goal (or set of goals) that the agent has to perform, as soon as it is created. In ***Jason*** *goals* are represented

```
1  /* Producer agent */
2
3  n_items_produced(0).  /* initial belief */
4  !produce.             /* initial goal */
5
6  /* plans */
7
8  +!produce
9    <- !setup;
10      !produce_items.
11
12 +!setup
13   <- makeArtifact("sharedBuffer","ExtBBuffer",[],Id);
14      focus(Id).
15
16 +!produce_items : not n_items_to_produce(0)
17   <- !produce_item(Item);
18      put(Item);
19      -n_items_produced(N);
20      +n_items_produced(N+1);
21      !produce_items.
22
23 +!produce_items : n_items_to_produce(0)
24   <- !finalize.
25
26 +!produce_item(Item) <- ...
27
28 +!finalize : n_items_produced(N)
29   <- println("completed - items produced: ",N).
30
31 -!produce_items
32   <- !finalize.
33
34 +stopped(true)
35   <- .drop_all_intentions;
36      !finalize.
```

Figure 4.2: Source code of the ***Jason*** producer agent used in the producer-consumer example.

by Prolog atomic formulae prefixed by an exclamation mark. Referring to the example, the producer agent (Figure 4.2) has an initial goal, which is represented by the `!produce` atomic formulae (line 04). Actually, goals can also be assigned at runtime, by sending to an agent achieve-goal messages.

Then, the main body of an agent program is given by a set of plans, which define the pro-active and reactive behavior of the agent. Agent plans are described by rules of the type `Event : Context <- Body`. `Event` represents the specific event triggering the plan. `Context` is a boolean expression on the belief base, indicating the conditions under which the plan can be executed once it has been triggered. Finally, the plan `Body` specifies the sequence of actions to perform, once the plan is executed. The actions contained in a plan body can be split in three categories:

- *Internal actions* — that are actions affecting only the internal state of the agent. Examples are actions to create sub-goals to be achieved (`!g`), to manage the execution of intentions

(i.e., active goals) – for instance, to suspend or abort the execution of an intention – to update agent inner state – such as adding a new belief (`+b`), removing beliefs (`-b`). Internal actions include also a set of primitives that allow for managing Java objects – which is the data model supported by CArtAgO – on the ***Jason*** side: so it is possible to create new objects (`cartago.new_obj`), invoke methods on objects (`cartago.invoke_obj`), etc.) and other related facilities (the prefix `<lib-name>.` in ***Jason*** is used to identify the library to which the specific actions belong to).

- *External actions* — that are actions provided by the environment, to interact with artifacts—as will be detailed in next section, these actions correspond to the operations provided by artifacts and included in artifact usage interfaces: so, as described when introduced the A&A meta-model, the repertoire of the actions of an agent working inside an artifact-based environment is dynamic and depends on the number and type of artifacts available in the environment.

- *Communicative actions* — which make it possible to communicate with other agents by means of message passing based on KQML performatives (`.send`, `.broadcast`, etc.).

Referring to the example, the producer agent has a main plan (lines 08-10), which is triggered by an event `+!produce` representing a new goal `!produce` to achieve. Since the agent has an initial `!produce` goal (line 04), then this plan will be triggered as soon as the agent is booted. By means of an internal action `!g`, the main plan generates two further sub-goals to be achieved sequentially: `!setup` and `!produce_items`.

The plan to handle the `!setup` goal (lines 12-14) creates a new instance called `sharedBuffer` of type ExtBBuffer by means of a predefined action called `makeArtifact` returning the logical id of the created artifact (see Section 4.2.3), and then starts observing it by executing the predefined action `focus` specifying its identifier (see Section 4.2.3). Then, two plans are specified for handling the goal `!produce_items`. One (lines 16-21) is executed if there are still items to produce—i.e., if the agent has not the belief `n_items_to_produce(0)`. Note that the value of this belief depends on the current state of the sharedBuffer artifact. This plan first produces a new item (subtask `!produce_item`), then inserts the item in the buffer by means of a `put` action, whose effect is to execute the `put` operation on the artifact; if this action succeeds, the plan goes on by updating the belief `n_items_produced` incrementing the number of items produced and generates a new sub-goal `!produce_items` to repeat the task. Actually, when executing an external action – such as `put` – it is possible to explicitly denote the specific artifact providing that action, in order to avoid ambiguities in case of multiple artifacts of the same type, by means of ***Jason*** annotations as follows:

```
1  put(Item) [artifact_name("sharedBuffer")];
2  put(Item) [artifact_id(ArtId)]; /* ArtId must contain the Id of an existing artifact,
3                                   * returned by the makeArtifact operation */
```

```
1  /* Consumer agent */
2
3  !consume.
4
5  +!consume: true
6    <- ?bufferReady;
7       !consumeItems.
8
9  +!consumeItems: true
10   <- get(Item);
11      !consumeItem(Item);
12      !consumeItems.
13
14 +!consumeItem(Item) <- ...
15
16 +?bufferReady : true
17   <- lookupArtifact("sharedBuffer",_).
18
19 -?bufferReady : true
20   <- .wait(50);
21      ?bufferReady.
```

Figure 4.3: Source code of the ***Jason*** consumer agent used in the producer-consumer example. Section 4.2.1 does not provide a detailed description of the consumer agent implementation, the source code is however reported for sake of completeness.

The other plan (lines 23-24) is executed if there are no more items to produce: in this case the `!finalize` goal is executed, which prints on standard output the number of items produced by the agent. In particular, the `println` action corresponds to the operation with the same name provided by an artifact called console, which is available by default in every workspace.

The reactive behavior of an agent can be realized by plans triggered by a belief addition/change/removal – corresponding to changes in the state of the environment – and by the failure of a plan in achieving some goal. In the example, the producer agent has a plan (lines 34-36) which is executed when the belief `stopped` about the observable property of the artifact is updated to `true`. This means that the user wants to interrupt and stop the production. So the plan stops and drops all the other possible plans in execution – using an internal action `.drop_all_intention` – and the `!finalize` sub-goal is executed.

Finally, the producer agent has also a plan (lines 31-32) to react to the *failure* of the `!produce_items` goal, which is expressed by the event `-!produce_items`. This can happen when the agent, believing that there are still items to be produced, starts the plan to produce a new item and tries to insert it in the buffer. However, the `put` action fails because other agents produced in the meanwhile the missing items.

4.2.2 Programming the Environment

The implementation of the ExtBBuffer artifact used in the producer-consumer example is shown in Figure 4.4. Being CArtAgO a framework on top of the Java platform, artifact-based envir-

onments are implemented using a Java-based API, exploiting the annotation framework. Here we do not go into the full details of such API, for more information, the interested reader can refer to CArtAgO papers [RPV11, RPVO09] and the documents that are part of CArtAgO distribution [Alea].

In CArtAgO, an artifact type can be defined by extending a base Artifact class. Artifacts are characterized by a *usage interface*, containing a set of operations and observable properties. In the example, the artifact ExtBBuffer has three operations: `put`, `get` and `stop`. The `put` operation inserts a new element in the buffer – decrementing the number of items to be produced – if the stopped flag has not been set, otherwise the operation (action) fails. The `get` operation removes an item from the buffer, returning it as a feedback parameter of the action. The `stop` operation sets the `stopped` observable property to `true`.

Operations are implemented by methods annotated with the `@OPERATION` annotation. The `init` method is used as constructor of the artifact, getting the initial parameters and setting up the initial artifact state. Inside an operation, guards can be specified (`await` primitive), which suspend the execution of the operation until the specified condition over the artifact state (represented by a boolean method annotated with `@GUARD`) holds. In the example, the `put` operation can be completed only when the buffer is not full (`bufferNotFull` guard) and the `get` one when the buffer is not empty (`bufferNotEmpty` guard). The execution of operations inside an artifact is transactional. This implies that: *(i)* changes to the observable state of the artifact are done atomically and only in case of operations executed with success, and *(ii)* at runtime multiple operations can be invoked concurrently on an artifact but *only one operation* can be in execution at a time–the other ones are suspended. On the agent side, by adopting an action-as-a-process semantics (Section 4.1.1), when executing an external action the agent plan is suspended until the corresponding artifact operation has completed (i.e., the action completed). Then, the action succeeds or fails when (if) the corresponding operation has completed with success or failure. It is worth noting that, in the meanwhile, the agent execution cycle can go on, making it possible for the agent to get percepts and select and perform other actions.

Besides operations, artifacts typically have also a set of observable properties (`n_items_to_produce` and `stopped` in the example), as data items that are automatically perceived by agents as environment state variables (as defined by the perception model implemented by JaCa, see Section 4.1.2). Instance fields of the class – instead – are used to implement the non observable state of the artifact—for instance, the list of items `items` in the example. Observable properties can be defined, typically during artifact initialization, by means of the `defineObsProperty` primitive, specifying the property name and initial value (lines 11-12). Inside operations, observable properties value can be inspected and changed dynamically by means of two basic primitives: `getObsProperty` to retrieve the current value of an observable property (see, for instance, line 17 and 20) and `updateObsProperty` to update the value (line 42). An artifact can make it observable also events occurring when executing operations. This can be done by using a `signal` primitive, specifying the type of the event and a list of actual parameters. For instance, `signal("my_event", "test",0)` generates an

```
1  import cartago.*;
2
3  public class ExtBBuffer extends Artifact {
4
5    private LinkedList<Object> items;
6    private int bufSize;
7
8    void init(int bufSize, int nItemsToProd){
9      items = new LinkedList<Object>();
10     this.bufSize = bufSize;
11     defineObsProperty("n_item_to_produce",nItemsToProd);
12     defineObsProperty("stopped",false);
13   }
14
15   @OPERATION void put(Object obj){
16     await("bufferNotFull");
17     ArtifactObsProperty stopped = getObsProperty("stopped");
18     if (!stopped.booleanValue()){
19       items.add(obj);
20       ArtifactObsProperty p  = getObsProperty("n_item_to_produce");
21       p.updateValue(p.intValue() - 1);
22     } else {
23       failed("no_more_items_to_produce");
24     }
25   }
26
27   @GUARD boolean bufferNotFull(){
28     return items.size() < nmax;
29   }
30
31   @OPERATION void get(OpFeedbackParam<Object> result){
32     await("itemAvailable");
33     Object item = items.removeFirst();
34     result.set(item);
35   }
36
37   @GUARD boolean itemAvailable(){
38     return items.size() > 0;
39   }
40
41   @OPERATION void stop(){
42     updateObsProperty("stopped",true);
43   }
44 }
```

Figure 4.4: The implementation of the ExtBBuffer in CArtAgO used in the producer-consumer example.

observable event `my_event("test",0)`. In the ExtBBuffer for example, to notify the stop we could generate a `stopped` signal in the `stop` operation, instead of using an observable property. Observable events are perceived by all agents observing the artifact—which could react to them as in the case of observable property change. Java objects and primitive data types are used as data model binding the agent and artifact layers, in particular to encode parameters in operations, fields in observable properties and signals.

Following the action model presented in Section 4.1.1, operations are computational pro-

cesses occurring inside the artifact, possibly changing the observable properties and generating observable events, as environment signals that be relevant for agents using/observing the artifact. An operation is executed as soon as an agent triggers its execution – by executing the corresponding action. Given the transactional execution semantics adopted, only one operation can be in execution at a time—so *no interferences* and *no race conditions* occur if multiple agents use concurrently the same artifact. Like in the case of monitors, other operations that are possibly and concurrently triggered are blocked (suspended). The conditions which can be specified with the `await` command are conceptually similar to condition variables. Differently from the monitor case (with threads or monitors), if an operation (action) is suspended, the agent that executed it is not: its execution cycle goes on, to eventually react to percepts and/or select and execute other actions from other plans.

Following the A&A meta-model, other features of the artifact model implemented in CArtAgO include: *(i)* the capability of *linking* together artifacts, making it possible for an artifact to execute operations (called linked operations) on other artifacts; *(ii)* the capability of triggering the execution of *internal* operations – i.e., operations not available in the artifact usage interface, hence not visible for agents – from other operations of the same artifact; and *(iii)* the capability of specifying for each artifact type a *manual* – i.e. a machine readable document containing the description of the functionalities provided by the artifacts of this type – and the operating instructions—i.e. how to exploit such functionalities (this features is actually in its early development stages).

4.2.3 The Multi-Agent Program in the Overall

Finally, the *main* or entry point of a JaCa multi-agent program is given by a ***Jason*** source file – with extension `.mas2j` – describing the initial configuration of the system, in particular the name of the MAS and the initial set of the agents that must be created and possibly some information and attributes that concern environment and agent implementation.

The configuration file for the producer-consumer example is shown in Figure 4.5, where ten instances of producer agents and ten instances of consumer agents are declared. To launch multiple agents of the same type (e.g., ten producer agents) the cardinality can be specified as a parameter in the declaration (`#10`); the unique name of the agent in this case is given by the type and a progressive integer (in the example: producer1, producer2, etc).

```
1 MAS prodcons {
2
3     environment: c4jason.CartagoEnvironment
4
5     agents: producer agentArchClass c4jason.CAgentArch #10;
6             consumer agentArchClass c4jason.CAgentArch #10;
7 }
```

Figure 4.5: Source code of the main configuration file (`prodcons.mas2j`), describing the initial configuration of the the producer-consumer MAS example.

By default, a single workspace called `default` is created and the specified agents join this workspace. Actually a JaCa program can be composed by multiple workspaces, and agents can concurrently join and work in multiple workspaces, either locally or in remote JaCa nodes (Section 4.3.2). Workspaces can be created dynamically by agents exploiting functionalities that are provided by a set of artifacts that are available, by default, in each workspace. Among the others, such a set includes: *(i)* a console artifact, providing functionalities for printing on standard output; *(ii)* a `WorkspaceArtifact`, providing functionalities for managing the current workspace, including creating new artifacts (`makeArtifact` operation), disposing existing artifacts (`disposeArtifact`), discovering the identifier of existing artifacts (`lookupArtifact`), starting and stopping artifact observation (`focus` and `stopFocus`), setting the security policies ruling agent access to artifacts, etc.; and *(iii)* a `blackboard` artifact, functioning as a blackboard – or better as a tuple space [Gel85] – providing functionalities for enabling indirect communication and coordination among agents.

4.3 JaCa Programming: Focus on Further Features

In this section we focus on three of the main features among the others that are provided by the JaCa platform, namely the capability of exploiting both direct communication based on message passing and indirect interaction through artifacts (Section 4.3.1), the support for building distributed programs, and finally the capability of integrating existing libraries such as GUI toolkits (Section 4.3.3). The interested reader can find the description of further features in JaCa and CArtAgO technical documentation.

4.3.1 Integrating Direct Communication and Mediated Interaction

In JaCa agents can interact and communicate in two basic ways, either exchanging messages through speech acts (Section 3.4.1) or by sharing and co-using artifacts functioning as interaction and coordination media [ORV+04]. The first way is generally referred as direct communication, while the latter as indirect or mediated communication. Both types of communication are important in programming concurrent and distributed programs, and we allow for exploiting them together. As mentioned before, the direct communication model is the one provided by the ***Jason*** language, based on a comprehensive subset of the KQML agent communication language (Section 3.4.1). Among the available performatives, `tell` makes it possible to inform an agent about some information (stored in the target agent as a belief), `achieve` to assign a new goal to the receiver agent, and `ask` to request information. These performatives must be included in the communication action `.send` that actually sends the message, along with the specific parameters. An agent can react to the arrival of messages or, at a higher level, to the effect that the speech acts have, which are uniformly modeled as belief addition (for the `tell` performative) or goal addition (for the `achieve` performative).

To give a concrete taste of the approach, in the following we describe the realization of

```
/* announcer agent */

!allocate_task("t0",2000).

+!allocate_task(Task,Deadline)
  <- makeArtifact("cnp_board","ContractNetBoard",[]);
     announce(Task);
     .wait(Deadline);
     close(Bids);
     !select_bid(Bids,Bid);
     award(Bid);
     cartago.invoke_obj(Bid,getWho,Who);
     println("Allocating the task to: ",Who);
     .my_name(Me);
     .send(Who,achieve,task_done(Task,Me)).

+!select_bid([Bid|_],Bid).

+task_result(Task,Result)
  <- println("Got result ",Result," for task: ",Task).
```

Figure 4.6: Source code of the announcer agent.

```
/* bidder agent */

task_result("t0",303).
!look_for_tasks("t0").

+!look_for_tasks(Task)
  <-  +task_descr(Task);
      focusWhenAvailable("cnp_board").

+task_todo(Task) : task_descr(Task)
  <-  !make_bid(Task).

+!make_bid(Task)
  <-  !create_bid(Task,Bid);
      .my_name(Me);
      bid(Bid,Me,BidId);
      +my_bid(BidId);
      println("Bid submitted: ",Bid," - id: ",BidId).

-!make_bid(Task)
  <-  println("Too late for submitting the bid.");
      .drop_all_intentions.

+winner(BidId) : my_bid(BidId)
  <-  println("awarded!.").

+winner(BidId) : my_bid(X) & not my_bid(BidId)
  <-  println("not awarded.").

+!create_bid(Task,Bid)
  <-  .wait(math.random(3000));
      .my_name(Name);
      .concat("bid_",Name,Bid).

 +!task_done(Task,ResultReceiver): task_result(Task,Res)
  <-  println("doing task: ",Task);
      .send(ResultReceiver,tell,task_result(Task,303)).
```

Figure 4.7: Source code of bidder agents.

```
/* Contract Net Board artifact */

public class ContractNetBoard extends Artifact {
  private List<Bid> bids;
  private int bidId;

  void init(){
    this.defineObsProperty("state","closed");
    bids = new ArrayList<Bid>();
  }

  @OPERATION void announce(String taskDescr){
    defineObsProperty("task_todo", taskDescr);
    getObsProperty("state").updateValue("open");
    bids.clear(); bidId = 0;
    log("New task announced: "+taskDescr);
  }

  @OPERATION void bid(String bid, String who,
              OpFeedbackParam<Integer> id){
    if (getObsProperty("state").stringValue().equals("open")){
      bidId++;
      bids.add(new Bid(bidId,who,bid));
      id.set(bidId);
    } else {
      this.failed("cnp_closed");
    }
  }

  @OPERATION void close(OpFeedbackParam<Bid[]> bidList){
    getObsProperty("state").updateValue("closed");
    int nbids = bids.size();
    Bid[] vect = new Bid[nbids]; bids.toArray(vect);
    bidList.set(vect);
    log("Auction closed: "+nbids+" bids arrived on time.");
  }

  @OPERATION void award(Bid prop){
    signal("winner", prop.getId());
    log("The winner is: "+prop.getId());
  }

  static public class Bid {
    private int id;
    private String who, descr;
    public Bid(int id, String who, String descr){
      this.descr = descr; this.id = id; this.who = who;
    }
    public String getWho(){ return who; }
    public int getId(){ return id; }
    public String getDescr(){ return descr; }
    public String toString(){ return descr; }
  }
}
```

Figure 4.8: Source code of the ContractNetBoard artifact.

```
MAS cnp_example {
  environment: c4jason.CartagoEnvironment
  agents:
    announcer agentArchClass c4jason.CAgentArch;
    bidder    agentArchClass c4jason.CAgentArch #5;
}
```

Figure 4.9: Source code of the main configuration file of the CNP example, spawning one announcer agent and five bidder agents.

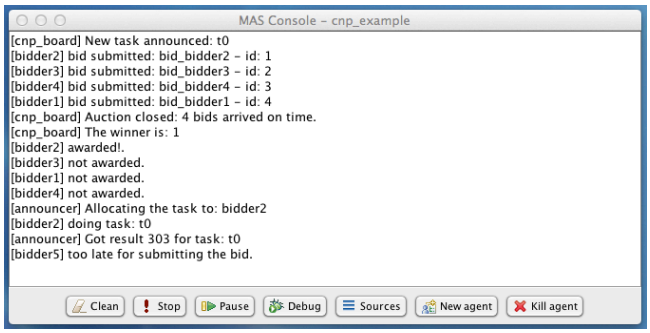


Figure 4.10: An execution trace of the CNP program, displayed on the JaCa output console.

simplified version of the Contract Net Protocol (CNP) [Smi80], in which both direct message passing and artifacts are used. In the example, a ContractNetBoard artifact (Figure 4.8) named cnp_board is used by an announcer agent (code shown in Figure 4.6) and five bidder agents (Figure 4.7) to help their coordination in choosing the agent to whom allocate a task to do; once the agent has been chosen, direct communication is used between the allocator of the task and the chosen agent to allocate the task and receive the results.

Some brief explanation of the program behavior follows. In the main configuration file (Figure 4.9), one announcer agent and five bidder agents are launched. The announcer opens the auction to allocate the task by performing a `announce` action over the cnp_board artifact (Figure 4.6, line 7). The artifact is observed and used also by a (possibly dynamic) set of bidder agents (Figure 4.7), who are available for doing tasks. The `announce` action/operation executed by the announcer creates a new observable property `task_todo`, storing information about the new task (Figure 4.8, line 12-17). As soon as each bidder perceives that there is a new task to do, it reacts (Figure 4.7, line 10-22) by computing a new bid and issuing them on the contract net board by performing a `bid` action. The action can fail if the auction has been already closed by the announcer: in that case a message is printed on the console (lines 20-22). On the artifact side, the `bid` operation (Figure 4.8, lines 19-28) just adds the new bid to the list of bids received so far, if the auction is still opened, otherwise the operation fails (by executing the `failed` artifact primitive). As a detail, the third parameter of the `bid` operation is an action feedback parameter storing the id of the placed bid.

The announcer waits some amount of time (2 seconds in the example), and then close the auction by invoking the `close` operation (Figure 4.6, lines 8-9), which results in changing the `state` observable property of the artifact to `"closed"` and returns the list of information about the received bids as an action feedback parameter (Figure 4.8, lines 30-36). Such information are represented by instances of the `Bid` class. Then, the agent selects a bid (in the example the first one) and awards the bidder by performing an `award` action (Figure 4.6, lines 10-11), which results in updating the content of the `winner` observable property in the artifact (Figure 4.8, lines 38-41). This change is perceived by bidder agents, which react in a different way depending on the fact that they are the winner or not (lines Figure 4.7 24-28). After award-

ing, the announcer then communicates directly with the winner bidder by sending an `achieve` message specifying the task to be done (Figure 4.6, line 15). To retrieve the identifier of the bidder agent to whom sending the message, the method `getWho` is invoked on the selected bid object by means of the `cartago.invoke_obj` internal action.

Then, the awarded bidder reacts to the new goal to achieve (Figure 4.7 lines 35-37), just printing a message and then sending a message to inform the announcer about the task result (Figure 4.6, line 37). Finally the announcer reacts to the new belief communicated by the bidder (Figure 4.6 lines 19-20) by printing the result on the console.

A possible execution trace that can be obtained by launching the program is reported in Figure 4.10, which shows the content of the JaCa output console. In that specific execution, four bidders were able to submit their bid on time and the winner was the bidder bidder2 (whose bidder identifier assigned by the cnp_board was 1).

4.3.2 Distributed and Open Systems Programming

JaCa directly supports distributed programming: an agent running on some node can join workspaces that are hosted on remote nodes, and then work with artifacts of the remote workspaces transparently. A simple example is shown in Figure 4.11, in which an agent joins a remote `test` workspace located in `acme.org`, and there, the agent prints some information on the console, creates a new Counter artifact called c0 and uses it, by executing the `inc` operation and reacting to changes to the `count` observable property.

While working on multiple workspaces, in JaCa a notion of *current* workspace is defined, being it the workspace which is implicitly referred when the agent invokes an operation over an artifact without specifying its full identifier. `current_wsp` is a predefined agent belief keeping track of the current workspace. When an agent starts its execution, the current workspace is set by default to the `default` workspace. Then, it is automatically updated as soon as the agent joins other workspaces (including remote ones) or the agent executes a predefined `set_current_wsp` action. So, in the example, by joining the remote `test` workspace, this becomes the current workspace, and then the `println` action acts on the console artifact there, as well as the `makeArtifact` action that creates a new artifact there too. It is worth noting that in the plan reacting to a change to the `count` observable property (mapped on `count` belief), the agent prints a message on the console in the *original* workspace (lines 19-21): to disambiguate what console to use, in the action an annotation reporting the workspace where the artifact is stored is specified (line 21). The agent source code includes also a plan reacting to a failure in the plan handling the `!use_remote` goal, in the case in which a Counter artifact called c0 was already present in the remote workspace.

So in the overall this facility makes it possible to implement open systems with dynamic and distributed structure and behavior, given by the capability of agents of spawning other new agents dynamically[2], of joining dynamically existing workspaces or creating new ones, of cre-

[2]This is possible thanks to the `.create_agent` ***Jason*** internal action, see [BHW07] for more details.

```
!test_remote.

+!test_remote
  <- ?current_wsp(Id,_,_);
     +default_wsp(Id);
     println("testing remote..");
     joinRemoteWorkspace("test","acme.org",WspID2);
     ?current_wsp(_,WName,_);
     println("hello there ",WName);
     !use_remote;
     quitWorkspace.

+!use_remote
  <- makeArtifact("c0","examples.Counter",[],Id);
     focus(Id);
     inc;
     inc.

+count(V)
  <- ?default_wsp(Id);
     println("count changed: ",V)[wsp_id(Id)].

-!use_remote [makeArtifactFailure("artifact_already_present",_)]
  <- ?default_wsp(WId);
     println("artifact already created ")[wsp_id(WId)];
     lookupArtifact("c0",Id);
     focus(Id);
     inc.
```

```
public class Counter extends Artifact {
  void init(){
    defineObsProperty("count",0);
  }
  @OPERATION void inc(){
    ObsProperty prop = getObsProperty("count");
    prop.updateValue(prop.intValue()+1);
  }
}
```

Figure 4.11: An agent joining and working in a remote workspace (*top*), and the source code of the counter used and observed remotely (*bottom*).

ating/disposing artifacts belonging to a workspace. Given the distributed programming facility, a workspace can be joined by unknown agents of JaCa programs that have been spawned independently from the program where the workspace has been defined. The possibility of explicitly specifying security policies at a workspace level – by exploiting the functionalities provided by the `workspace` artifact – makes it possible to govern such openness according to the need.

4.3.3 Wrapping Existing Libraries and External Resources

Specific kind of artifacts can be designed and used to wrap and reuse existing libraries – written in Java but also in other languages, such as C and C++, exploiting the JNI [SUNb] (Java Native Interface) mechanism – making their functionalities available to agents, with a clean and

```
package c4jexamples;
...
public class View extends GUIArtifact {
  private MyFrame frame;

  public void setup() {
    frame = new MyFrame();
    defineObsProperty("value",0);
    linkActionEventToOp(frame.stopButton,"stop");
    linkWindowClosingEventToOp(frame, "close");
    frame.setVisible(true);
  }

  @INTERNAL_OPERATION void stop(ActionEvent ev){
    signal("stopped");
  }

  @INTERNAL_OPERATION void close(WindowEvent ev){
    signal("closed");
  }

  @OPERATION void setOutput(int value){
    frame.updateOutput(""+value);
    getObsProperty("value").updateValue(value);
  }

  class MyFrame extends JFrame {
    private JButton stopButton;
    private JTextField output;

    public MyFrame(){
      setTitle(".:: View ::.");
      setSize(200,100);
      JPanel panel = new JPanel();
      setContentPane(panel);
      stopButton = new JButton("stop");
      stopButton.setSize(80,50);
      output = new JTextField(10);
      output.setText("0");
      output.setEditable(true);
      panel.add(output);
      panel.add(stopButton);
    }
    public void updateOutput(String s){
      output.setText(s);
    }
  }
}
```

```
count(0).
!do_task_with_view.

+!do_task_with_view
  <-  makeArtifact("gui",
        "c4jexamples.View",[],Id);
      focus(Id);
      !do_task.

+!do_task
  <-  -count(C);
      C1 = C + 1;
      +count(C1);
      setOutput(C1);
      !do_task.

+stopped : value(V)
  <-  .drop_all_intentions;
      println("stopped - value: ",V).

+closed
  <-  .my_name(Me);
      .kill_agent(Me).
```

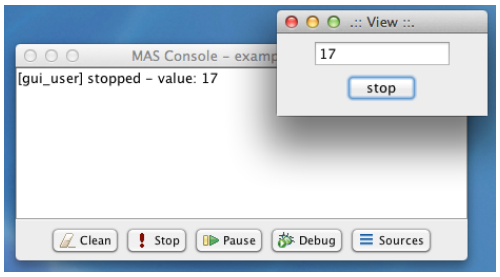


Figure 4.12: Implementing and using GUI in JaCa: the View artifact (*left*), the agent using the GUI (*right–top*) and the output of the program (*right–bottom*).

uniform interface—which is the one provided by the artifact use and observation model.

A main example of a JaCa library wrapping and exploiting existing technologies is the one that allows for building and exploiting graphical user interface (GUI) toolkits. GUIs inside a JaCa program are modeled as artifacts mediating the interaction between humans and agents. A basic abstract artifact GUIArtifact is provided to be extended – through the Java standard

inheritance mechanism – in order to create concrete GUI. A GUI is designed then to make it observable to interested agents the events generated by the components (buttons, edit fields, list boxes,...) inside the GUI. Also, as an artifact, it provides operations that allows agents to interact with the GUI itself, for instance to set the content of text fields.

Figure 4.12 shows a simple example, in which an agent uses a GUI to repeatedly display the output of its work and to promptly react to user inputs. In particular, the agent creates a GUI artifact called View, providing one `stop` button and one output edit text. The structure of the GUI – based on Java Swing library – is defined by the MyFrame class, as it would be in a traditional object-oriented program. An instance of this class is created inside the View artifact and events generated by the GUI components are linked to internal operations of the artifact by means of a set of predefined methods implemented in GUIArtifact. In particular an action event generated by `frame.stopButton` causes the execution of the internal operation `stop`, which generates an observable event `stopped`, and the window closing event is mapped onto the `close` operation, which generates a `closed` event.

The agent first creates an instance called gui of the View artifact (Figure 4.12 top-right, lines 5-6), and then repeatedly uses the view to display the results of its task, by means of the `setOutput` operation (Figure 4.12 top-right, lines 10-15). While doing this task, the agent also observes the GUI and as soon as a `stopped` event is perceived, the agent reacts by suspending all its current ongoing activities (intentions) and printing in standard output a message (Figure 4.12 top-right, lines 17-19). If a `closed` event is perceived, the agent terminates (Figure 4.12 top-right, lines 21-23).

4.4 JaCa-Android: Programming Smart Mobile Applications in JaCa

Mobile hardware (HW) technologies have witnessed an extraordinary development and progress in recent years. Starting from first devices with small processors (tens of Megahertzs) and small memory (tens of Megabytes), no sensors and no Internet access, current smart-phones have multi-cores Gigahertzs CPU and several hundreds of Megabytes of memory, different kinds of sensors (camera, GPS, accelerometers, etc.) and full-fledge network connectivity (given by High-Speed Downlink Packet Access (HSDPA), 3G, 4G, Long Term Evolution (LTE) protocols, WIFI, Bluetooth).

This hardware development naturally leads to rethink the kind of *software* that could be used on mobile devices, and related functionalities. First, the HW allows for running operating systems (OSs) and applications that are similar – in functionalities and complexities – to the ones adopted in desktop systems, from rich Internet applications to video-games. Furthermore, it allows for conceiving applications that provide functionalities not typically found in desktop systems, such as context-aware, pervasive computing and ambient intelligence applications. So the new generation of mobile devices opens the way to a new generation of *smart mobile applications*, integrating all these features [LJR09, Wri09].

The systematic design and development of such mobile applications introduce a new level of complexity. On the one side, such complexity is comparable to the one that we have when engineering desktop applications: to this end, high-level programming languages and platforms used for desktop environments become meaningful also for the mobile context, along with specific middleware that allows for fully exploiting mobile device resources. The Android platform is a main example [Gooa]. On the other side, the systematic development of smart mobile applications accounts for dealing with aspects that make their programming quite challenging, even using mainstream high-level programming languages, such as: *(i) reactiveness*—programming applications that must be reactive to events related to user inputs, sensors, the network; *(ii) pro-activeness*—applications that must be able to integrate a task oriented behavior with the capability to react to events, possibly taking autonomously some action (without user intervention) by virtue of the goals of the application; *(iii) flexibility*—applications that are capable to adapt their computational behavior according to the changes that dynamically occur in the external environment – which can include the user context, the network context; *(iv) interaction-oriented-ness*—applications that frequently need to interact with some remote service or application, even mobile applications running on other mobiles devices. These features can be easily recognized in most of the advanced application scenarios which are typically ascribed to smart mobile applications [LJR09].

Given that, a relevant research issue is how to design these applications, and so looking for high-level programming tools and development platforms that *(i)* would provide a proper *level of abstraction* to deal with such complexities, *(ii)* would be general enough to be reused in different application domains, and *(iii)* would make it possible to be integrated with existing platforms and programming technologies (such as Android) to fully exploit their capabilities.

Accordingly, we extended JaCa and realized an agent-oriented mobile platform named JaCa-Android [SGA10, SGR11, SR11a, Anda], providing an agent-oriented level of abstraction to design, program and execute smart mobile applications on top of the Android platform. By adopting the JaCa programming model, a mobile Android application can be designed and organized as a multi-agent system, composed by one or multiple workspaces in which ***Jason*** agents are used to encapsulate the logic and the control of the tasks involved in the mobile application, and artifacts are used as tools for agents to seamlessly exploit available Android device/platform components and services.

4.4.1 The JaCa-Android Platform

JaCa-Android makes it possible to run JaCa applications on top of the Android platform. The application and programming model adopted in JaCa-Android is entirely rooted on JaCa, which here has been properly extended with a layer specialized for the mobile context. So, a JaCa-Android application runs on top of a JaCa runtime/infrastructure, equipped with a layer that makes it possible to access/control resources provided by the Android Framework (see Figure 4.13). Being fully developed in Java, the core JaCa runtime/infrastructure runs as a normal Android application.

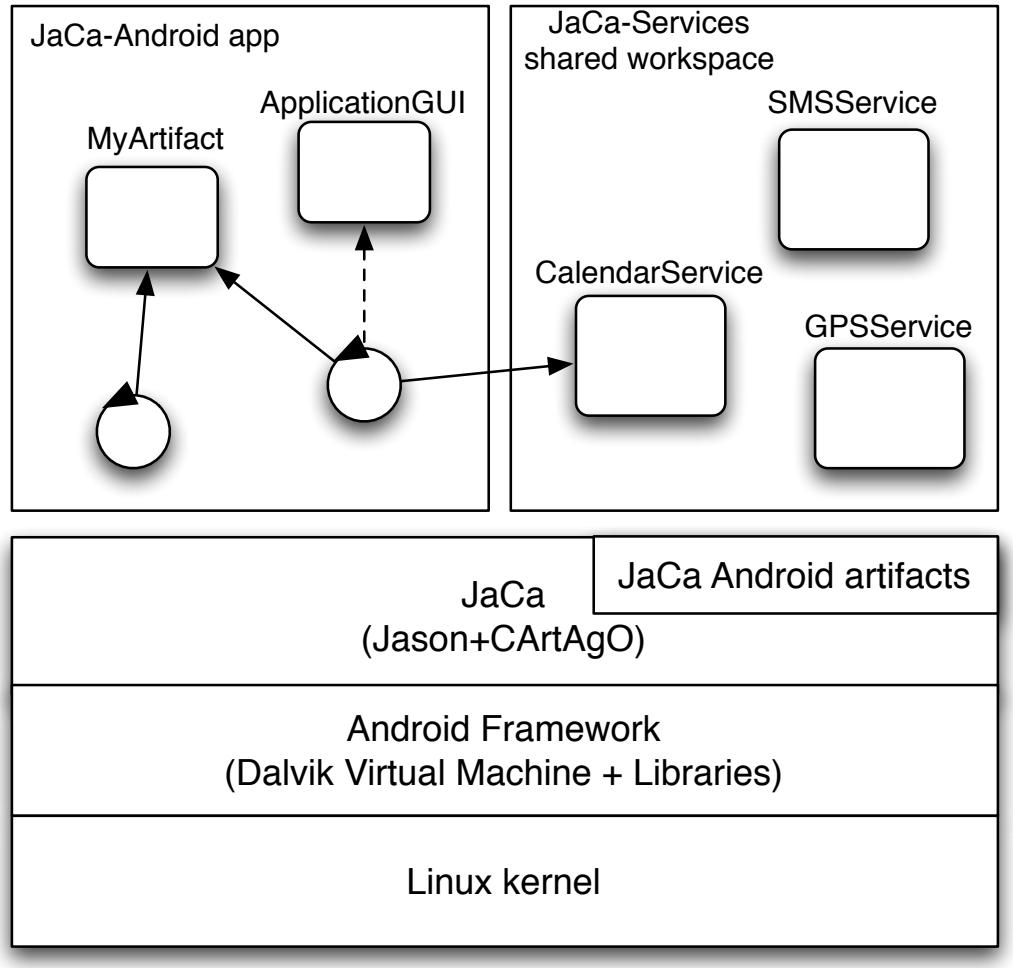


Figure 4.13: Abstract representation of the JaCa-Android platform – with in evidence the different agent technologies upon which the platform is based – and of a generic application running on top of it.

JaCa-Android has been realized with the aim to provide to developers a platform able to guarantee a seamless access and management of Android features thanks to artifacts, and to the agent and artifact interaction model (Section 4.1). Indeed, the key part of the platform is a proper set of artifacts encapsulating the main features provided by Android, allowing agents to access and use the functionalities they need abstracting from low-level implementation details. The platform guarantees direct access to the most common functionalities required by smart mobile applications providing a shared workspace called JaCa-Services (see Figure 4.13) containing a set of general-purpose artifacts. This workspace – being executed at each startup of

the device and stored into a proper Android service[3] installed with the JaCa-Android platform – is shared among all the JaCa-Android applications. The artifacts in this workspace can be seen like singletons in the context of object-oriented applications—i.e., is needed only one instance of them, which is shared among all their users. In detail the artifacts contained in the JaCa-Services workspace are:

- SMSService/MailService: providing functionalities related to SMSs/mails (send and receive SMSs/mails, retrieve stored SMSs/mails, etc.).
- GPSService: providing GPS-related functionalities (e.g., geolocalization of the device).
- AccelerometerSensorSevice/GyroscopeSensorService .. : providing information related to device sensors (e.g., accelerometer, gyroscope..).
- CallService: providing functionalities to answer/reject phone calls.
- ConnectivityService: managing the access to the different kinds of connectivity supported by the mobile device.
- CalendarService: for managing the built-in Google calendar.
- PhoneSettingsService: used for managing the device ringtone/vibration.

The platform also includes a set of predefined types of artifacts (GUIArtifact[4], BroadcastReceiverArtifact, ContentProviderArtifact, ServiceArtifact) specifically designed to build compliant Android components. So, standard Android components can be used as artifacts that agents and developers can exploit without worrying and knowing about infrastructural issues related to the Android SDK. This makes it possible for developers to conceive and realize mobile applications that are seamlessly integrated with the Android SDK, possibly reusing components and applications developed using the standard SDK. Indeed these artifacts on the one side allow developers to build – in terms of artifacts – classical Android components, so directly usable also from standard Android applications; and on the other side they allow JaCa-Android applications to interact with any existing Android components. This integration is fundamental in order to guarantee to developers the re-use of existing legacy – i.e., the standard Android components and applications – and for avoiding the development of the entire set of functionalities required by an application from scratch.

Besides the set of artifacts described so far, a developer designing a JaCa-Android application can – and should – also develop a set of artifacts specific for the particular application she is realizing—i.e., artifacts representing the external resources needed by agents to achieve their goals in the context of *that* specific application (Figure 4.13 top left).

[3] A service is an Android application component exploited to do long-running operations while not interacting with the user. See [Goob] for further details.

[4] This is a specialization of the GUIArtifact discussed in Section 4.3.3, specifically designed to be the basic artifact for building Android GUIs.

4.4.2 A Concrete Case Study

In this section we describe an example of mobile application developed with JaCa-Android, featuring some of the complexities identified in the previous sub-section. In particular, the example will focus on the management of asynchronous interactions with external resources, such as – for example – Web Services and on the capability to express context-sensitive behaviors. The application considered is a SmartNavigator (Figure 4.15 able to assist the user during her trips in an *"intelligent"* way, taking into the account the current traffic conditions.

```
1  preferences([...]).
2  relevance_range(20).
3
4  !assist_user_trips.
5
6  +!assist_user_trips
7    <- focus("GPSService");
8       focus("GoogleMapsArtifact");
9       focus("SmartNavigatorGUI");
10      focus("TrafficConditionsNotifier").
11
12 +route(StartLocation, EndLocation)
13   <- !handle_navigation(StartLocation, EndLocation).
14
15   +!handle_navigation(StartLocation, EndLocation)
16   <- ?relevance_range(Range); ?current_position(Pos);
17      -+leaving(StartLocation);-+arriving(EndLocation);
18      calculateRoute(StartLocation, EndLocation, OutputRoute);
19      -+route(OutputRoute);
20      subscribeForTrafficCondition(OutputRoute, Range);
21      set_current_position(Pos);
22      updateMap.
23
24 +new_traffic_info(TrafficInfo)
25   <- ?preferences(Preferences);
26      ?leaving(StartLocation); ?arriving(EndLocation);
27      !check_info_relevance(TraffincInfo,Preferences);
28      !update_route(StartLocation, EndLocation, TrafficInfo, NewRoute);
29      !update_subscription(NewRoute);
30      updateMap.
31
32 +current_position(Pos)
33   <- ?route(Route);
34      !check_position_consistency(Pos, Route);
35      set_current_position(Pos);
36      updateMap.
37
38 -!check_position_consistency(Pos, Route)
39   : arriving(EndLocation)
40   <- !handle_navigation(Pos, EndLocation).
```

Figure 4.14: Source code snippet of the nav-manager ***Jason*** agent used in the SmartNavigator example.

The application is realized using a single ***Jason*** agent (nav-manager) and four different artifacts: *(i)* the GPSService used for the smart-phone geolocalization (Figure 4.16), *(ii)* the

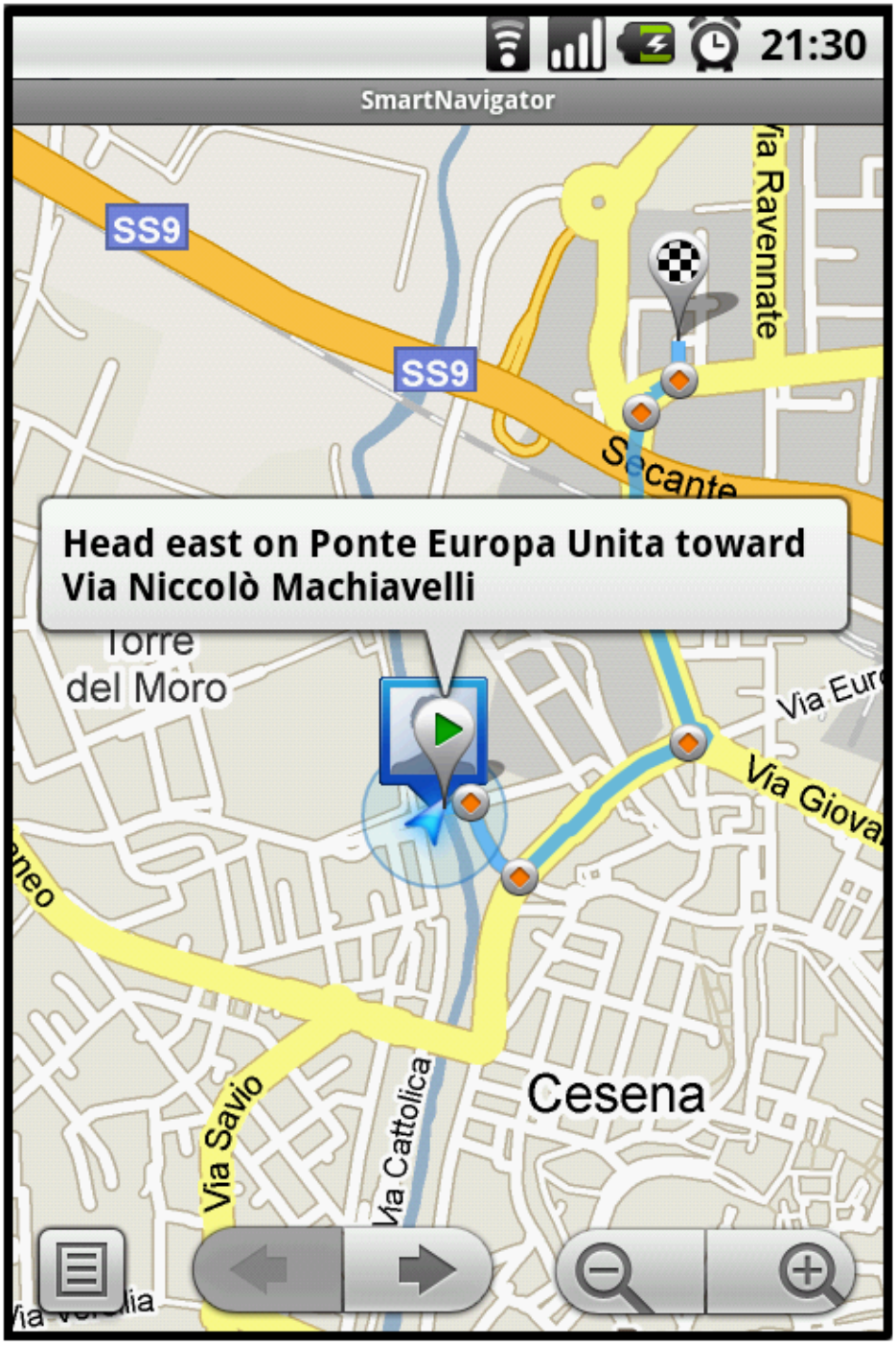


Figure 4.15: A screenshot of the SmartNavigator application that integrates in its GUI some of the Google Maps components for showing: *(i)* the user current position, *(ii)* the road directions, and *(iii)* the route to the designed destination.

```
1  public class GPSService extends LocationManagerArtifact {
2
3    /* Init */
4    public void init(int minTime, int minDistance) {
5      super.init(minTime, minDistance);
6
7      /* Link of events coming from the GPS to the internal
8       * operations responsible of their management */
9      linkOnLocationChangedEventToOp(LocationManager.GPS_PROVIDER, "onLocationChange");
10     linkOnProviderEnabledEventToOp(LocationManager.GPS_PROVIDER, "onProviderEnabled");
11     linkOnProviderDisabledEventToOp(LocationManager.GPS_PROVIDER, "onProviderDisabled");
12
13     Location location =
14       getLocationManager().getLastKnownLocation(LocationManager.GPS_PROVIDER);
15     defineObsProperty("current_position",
16                       new Position (location.getLatitude(), location.getLongitude());
17     defineObsProperty("altitude", location.getAltitude());
18     defineObsProperty("accuracy", location.getAccuracy());
19     defineObsProperty("bearing", location.getBearing());
20     defineObsProperty("speed", location.getSpeed());
21   }
22
23   @INTERNAL_OPERATION void onLocationChange(Location arg0) {
24     getObsProperty("current_position").updateValue(
25                    new Position (arg0.getLatitude(), arg0.getLongitude()));
26     getObsProperty("altitude").updateValue(arg0.getAltitude());
27     getObsProperty("accuracy").updateValue(arg0.getAccuracy());
28     getObsProperty("bearing").updateValue(arg0.getBearing());
29     getObsProperty("speed").updateValue(arg0.getSpeed());
30   }
31
32   @INTERNAL_OPERATION void onProviderEnabled(String provider) {
33     signal(ON_PROVIDER_ENABLED,provider);
34   }
35
36   @INTERNAL_OPERATION void onProviderDisabled(String provider) {
37     signal(ON_PROVIDER_DISABLED,provider);
38   }
39 }
```

Figure 4.16: Source code of the GPSService artifact used in the SmartNavigator example.

GoogleMapsArtifact, an artifact specifically developed for this application, used for encapsulating the functionalities provided by Google Maps (e.g., calculate a route, show points of interest on a map, etc.), *(iii)* the SmartNavigatorGUI, an artifact developed on the basis of the GUIArtifact and some other Google Maps components, used for realizing the GUI of the application, and *(iv)* an artifact, TrafficConditionsNotifier, used for managing the interactions with a Web Service[5] that provides real-time traffic information. Figure 4.14 shows a snippet of the agent source code.

The nav-manager has a set of initial beliefs (lines 1-2) storing user preferences about the trip – e.g., consider/avoid highways – and the range of interest for traffic conditions updates—i.e., a 20 Kms radius from the current position in the case of this example. Terminated the initialization of the artifacts that will be used by the agent during its execution (lines 6-10), the agent

[5] http://www.stradeanas.it/traffico/

main goal `!assist_user_trips` is managed by a set of reactive plans that are structured in a hierarchy of sub-goals, handled by a set of proper sub-plans. The first reactive plan, reported at lines 12-13, is executed after the reception of an event related to the modification of the SmartNavigatorGUI `route` observable property—i.e., a property that contains both the starting and arriving locations provided in input by the user. The handling of this event generates a new sub-goal `!handle_navigation`, managed by the plan at lines 15-22 that: *(i)* retrieves (line 16) and updates the appropriate agent beliefs (lines 17 and 19), *(ii)* computes the route for arriving to the target destination using an operation provided by the GoogleMapsArtifact (`calculateRoute`, line 18), *(iii)* makes the subscription – for the route of interest – to the Web Service that provides the traffic information using the TrafficConditionsNotifier artifact (line 20), and finally *(iv)* updates the map showed by the application (using the SmartNavigatorGUI operations `setCurrentPosition` and `updateMap`, lines 21-22) with both the current position of the mobile device (provided by the observable property `current_position` of the GPSService) and the new route.

In the case in which no relevant changes occur in the traffic conditions and the user strictly follows the indications provided by the SmartNavigator, the map displayed in the application GUI will be updated until arriving to the designed destination, simply moving the current position of the mobile device using the plan reported at lines 32-36. This plan, activated by a change of the observable property `current_position`, simply considers (using the sub-plan `check_position_consistency` instantiated at line 34, not reported here for simplicity) if the new device position is consistent with the current route (retrieved from the agent beliefs at line 33) before updating the map with the new geolocation information (line 35-36). In the case in which the new position is not consistent – i.e., the user chose the wrong direction – the sub-plan `check_position_consistency` fails. This failure is handled by a proper ***Jason*** failure handling plan (lines 38-40) that simply re-instantiates the `handle_navigation` plan for computing a new route able to bring the user to the desired destination from her current position (that was not considered in the previous route).

The `+new_traffic_info` plan (lines 24-30) is worth of particular attention since is the one that makes it possible the integration of the application reactive behavior – i.e, the asynchronous reception of traffic information from the Web Service – with its pro-active behavior—i.e, assisting the user during her trips. The plan reacts to the reception of updates related to the traffic conditions from the Web Service. If the new information are considered relevant w.r.t. the user preferences (sub-plan `check_info_relevance` instantiated at line 27 and not shown) then, on the basis of this information, the current route (sub-plan `update_route` instantiated at line 28), the Web Service subscription (sub-plan `update_subscription` instantiated at line 29), and finally the map displayed on the GUI (line 30) are updated.

4.5 JaCa-WS: Programming Applications based on the Service-Oriented Architecture and Web Services in JaCa

Agents and multi-agent systems are more and more recognized in the literature as a suitable paradigm for engineering systems rooted on the Service Oriented Architecture (SOA) and Web Services (WSs), since they provide a conceptual and engineering background that naturally fits many complexities concerning such systems at a high abstraction level [Hun06, NHSB05, GC04, Huh02]. Actually, this view is also promoted both by the official service-oriented model described by the W3C [Wor04], and by the Object Management Group (OMG) initiative about the definition of an agent meta-model and profile in the SOA perspective [Obj08].

Besides being an effective meta-model to *design* SOA, we argue that agent-oriented programming languages and technologies can be effective tools for concretely *programming* SOA and Web Services applications, in particular for those kind of service-oriented systems that need to integrate advanced features such as autonomy, flexibility, reactiveness and asynchronous interaction management [CFNS05, NHSB05].

In that perspective we devised a proper extension of the JaCa platform named JaCa-WS [PRS09, PSR09], which has been engineered on the basis of our work in the context of a CArtAgO extension named CArtAgO-WS [RDP10]. JaCa-WS enables the development of service-oriented applications as JaCa multi-agent systems, in particular as CArtAgO workspaces in which BDI-based ***Jason*** agents work together, sharing and exploiting artifact-based environment facilities:

- Agents encapsulate the logic and control of tasks, activities and business processes characterizing the SOA-specific scenario.
- Artifacts instead are used – as usual – to represent specialized resources and tools inside the workspaces that agents can exploit. In this case they are useful in particular to model and engineer those parts in the agent world that encapsulate Web Services aspects and functionalities – e.g., interaction with existing Web Services (agents as service consumers), implementation of Web Services (agents as service providers) – possibly wrapping existing non-agent-oriented code.

4.5.1 The JaCa-WS Platform

As depicted in Figure 4.17, the JaCa-WS platform is currently implemented on top of the Axis2 [Foua] open-source application server, in order to conform to the Basic Profile specification [WSI] of the Web Service Interoperability Organization (WS-I). Basically JaCa-WS relies on the standard JaCa programming model and runtime infrastructure, which here have been properly extended with a library composed by different kind of specialized artifacts aimed at working with Web Services. In detail these artifacts are:

- *Basic WS artifacts* — aimed at enabling basic interactions between agents and WSs.

- *WS-* artifacts* — aimed at supporting an enriched set of Web Service features (security, distributed transactions, etc.), as the ones envisaged by advanced WS specifications.

- *Business artifacts* — they are application-specific and aimed at providing functions for supporting agents in their business activities—e.g., storing information relevant for the ongoing tasks in a database, wrapping an external resource, control a GUI, etc.

Below, is provided a description of the artifacts belonging to the first two groups, while examples of artifacts in the third group are given in the next sub-section. Artifacts of the basic group allow, on the one side, agents to work with existing Web Services and, on the other side, the construction and the deployment of new Web Services controlled by agents. To this end, two configurable artifacts have been introduced: WSInterface and WSPanel.

To interact with an existing Web service, an agent instantiates a WSInterface artifact specifying its WSDL [Wor07] document, which describes the service to interact with. Once created, the WSInterface provides basic operations to interact with the specified Web Service: sending a message to the service in the context of an operation (`sendWSMsg`, `doRequestResponse`) or getting replies to messages previously sent during an operation (`getWSReply`). The current implementation makes use of SOAP [256] messages for executing operations and to get replies sent back by the service, according to message exchange patterns and quality of service (e.g., security and reliability) defined in the WSDL. In future implementation of this artifact we plan to add support for resource-oriented interaction with services, as promoted by the REST architectural style [FT02].
For creating, configuring and controlling a new Web Service, a WSPanel artifact is provided. Analogously to the previous case, a WSPanel artifact must be instantiated specifying a WSDL document. Once created, the WSPanel provides basic functionalities to manage SOAP requests, including receiving and sending messages according to the specific message-exchange pattern (MEP) described in the WSDL, and basic controls to configure security and reliability policies. As for the WSInterface, also for the WSPanel alternative transport mechanisms could be envisaged in the future (e.g., REST). Operations are available to retrieve or to be notified about messages received by the Web Service exposed by the WSPanel, optionally specifying filters to select messages on the basis of their content (`getWSMsg, getWSMsgWithFilter` and `subscribeWSMsgs`), and to send replies accordingly (`sendWSReply`).

Besides the basic interactions promoted by the aforementioned artifacts, JaCa-WS introduces an additional group of artifacts (WS-* artifacts). This group aims at supporting an extensible set of Web Service specifications, particularly those appearing in the Web Services Interoperability Technologies (WSIT) set [Sund]. For doing this, two different artifacts are provided: the WSRequestMediator and the Wallet. The WSRequestMediator (RM) artifact is meant to be used by agents to retrieve (or create) dynamic information such as those required by complex specifications WS-Coordination [osftis08b] (WS-C), WS-AtomicTransaction [osftis08a] (WS-AT) or the ones included in the WS-Security framework [osftis06] (WS-SEC). For instance, suppose that an agent needs to create a new distributed transaction following the WS-AT

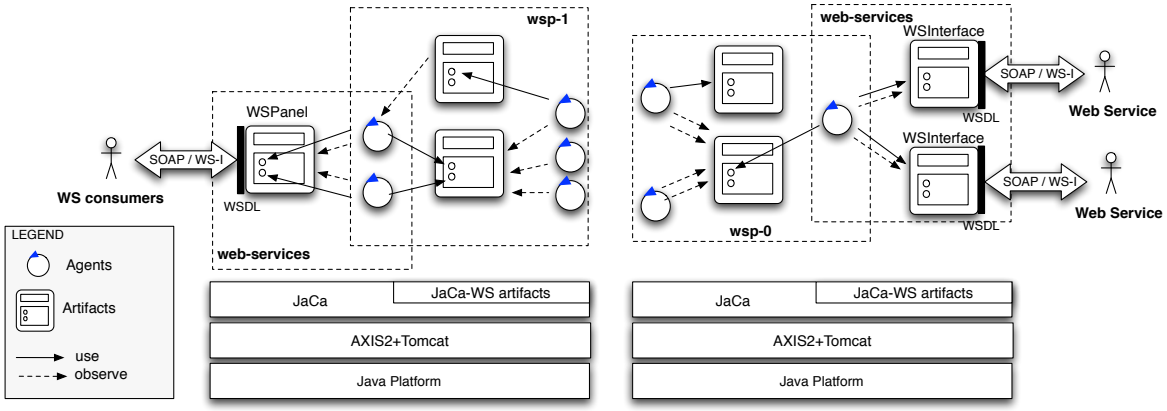


Figure 4.17: Abstract representation of the JaCa-WS platform. On the left side, a JaCa-WS application composed by two workspaces: *(i)* `web-services`, and *(ii)* `wsp-1`. In `web-services` workspace, an instance of WSPanel artifact is shared and used by two agents to process requests and send replies. On the right side, a JaCa-WS application using existing services through two instances of WSInterface artifact exploited by the same agent to interact concurrently with two distinct Web Services.

specification: to this end, it can use a RM to automatically create and retrieve a specific coordination context, which has been properly configured following WS-C and WS-AT standards.

Besides RM, a Wallet artifact is introduced as a *personal artifact* for an agent, to support the management of profile/context information related to some WS specifications. The Wallet works in synergy with RM artifacts and its function is to dynamically store a portfolio of various policies which are required to conform messages to Web Service protocols. This information can range from security tokens (as required by WS-SEC) to dynamic coordination contexts (as used in WS-C). In doing so, a user agent can completely externalize the management of the required policies on the wallet. In a typical scenario, an agent using a Web Service firstly gets profile information using a RM and stores it in its personal Wallet. Then, when needed, the information is retrieved from the Wallet and used to properly configure the desired WSInterface or WSPanel used for the communication.

So, generally speaking the WS-* artifacts allow programmers to build articulated applications abstracting as much as possible from low-level details that concern specific protocol management (e.g., the management of the coordination contexts in WS-C), and to focus on the high-level functionalities (e.g., distributed transactions) that agents may need to setup/exploit.

4.5.2 A Concrete Case Study

In this section we consider a concrete case study of a SOA/WS application realized with JaCa-WS aiming at showing: *(i)* how the behavior of complex Web Services can be pro-

grammed as business processes conceived as a complex chain of interleaved tasks expressed in a goal-oriented format, and thus managed in terms of agent's plans; and *(ii)* how ***Jason*** BDI agents rooted on practical reasoning can suitably find a proper course of actions to achieve a given goal given the current context conditions experienced in the specific SOA/WS scenario.

The case study is inspired by a typical example used in SOA/WS contexts: a user wants to book an holiday for a given date by exploiting a series of web services providing the required resources for hotel reservation, transport facilities and payment. As an additional element of the scenario, we imagine for the client the possibility to be further notified when a selected range of dates has become available for additional hotel reservations. This allows clients to express an interest for a given date, and thus to re-try the booking activity when the hotel signals a last minute availability (e.g., due to other reservations being canceled). The whole booking process must be managed via a distributed atomic transaction (as specified by WS-AT) in order to ensure that the entire booking operation is either performed correctly up to completion or aborted in case of any problems (e.g., hotel full, insufficient funds available, etc.). On these basis, the involved services need to shape their activities based on situated conditions:

- A transaction can have success or not, given the resources which are *actually* available.
- A new booking operation can be retried, based on changed contexts for which, at the moment of the first attempt, the client could not finalize the task.

The application is centered on two main side: a server side and a client side (Figure 4.18). On the server side we have three Web Services that the user needs to interact with in order to perform the holiday booking. The first one is the Hotel Manager (HM), a service that manages the booking for the target hotel and also provides notification functionalities to subscribers. It has been designed using two agents, the Hotel Basic Agent and the Hotel Notifier Agent, sharing and exploiting an instance of WSPanel to expose the service (see Figure 4.18 right). To support their tasks, the agents use two additional artifacts (i.e., business artifacts following the classification given in Section 4.5.1): *(i)* the HotelBookingRegistry artifact, to manage the requests related to bookings and cancellations, and *(ii)* the SubscribersMap artifact (Figure 4.20), used mainly by the Hotel Notifier Agent in combination with the former one, to notify interested subscribers as soon as changes regarding date availabilities are observed.

The other two services on the server side are the TransportManager service (TM) that manages the booking for the transports used for arriving to (and leaving from) the specified destination, and the PaymentManager service (PM) which manages online payments.

On the client side we only have the Booking Service (BS), built around the role played by a Booking Requestor Agent (BRA), whose final goal is to plan the required reservation related to an holiday by interacting with the WSs on the server side.
Figure 4.19 shows the implementation of the BRA. For sake of simplicity, some of the agent's sub-goals concerning low level computations are not fully specified here (e.g., the sub-goal `!retrieve_date` at line 4 which is used to retrieve the date information provided by the human user, and to store it in form of an agent belief `date(Date)`). The BRA initial

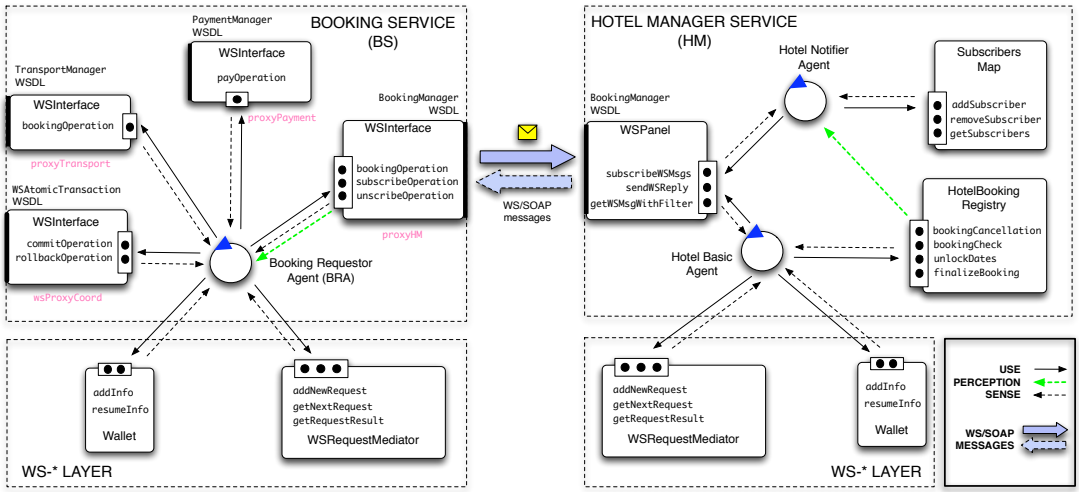


Figure 4.18: Structural architecture showing the services involved in the holiday booking example scenario. On the left side, the Booking Service controlled by the Booking Requestor Agent which manages the WSInterface artifacts for contacting the Transport Manager, Payment Manager, Hotel Manager and WSAtomicTransaction. On the right side, the Hotel Manager Service uses two agents (Hotel Notifier and Hotel Basic) and two artifacts (SubscribersMap and HotelBookingRegistry) in order to provide the booking service and the notification events exploitable by the clients. The two services make use of the WS-* artifacts to coordinate the transactions according to WS-* protocols.

goal `!start_booking` (line 1) is decomposed in a series of sub-goals. The initial one `!setup_tools` is used to retrieve and create the needed artifacts (artifact identifiers are stored as beliefs in the form `artifact_id(a_name, a_id)`). Then, the booking activity is performed by the sub-goal `!book_an_holiday`. The plan for this sub-goal: *(i)* creates the WS-AT context through the RequestMediator (line 12) and stores it in the Wallet for further use (line 14), and then *(ii)* configures – with the WS-AT context just created – the WSInterfaces used to interact with the HM, TM and PM (lines 16-18). Then the plan decomposes its behavior by defining a workflow of further sub-goals, realized by dedicated plans, as they are specified at lines 19-21.

The first sub-goal (`!book_hotel`, line 19) manages the booking of the hotel for the given dates: after retrieving the belief related to the `proxyHM` from the agent belief base, a request message for the HM is prepared (line 25) and sent by means of the request-response MEP. This is done by using the `doRequestResponse` operation of the WSInterface artifact (line 26).

We may assume that the hotel has already reached the maximum amount of reservations for some of the dates in the requested period. In that case, the HM service replies to BS with a message notifying the inability to finalize the reservation: this message is then analyzed by a special `inspect_h_response` plan that can provide an `available` or `not_available`

```
!start_booking.

+!start_booking
  <- !setup_tools; !retrieve_date; !book_an_holiday.

+!setup_tools : true
  <- /* create and locate artifacts, not shown for sake of simplicity */

/* Top Level Goal */
+!book_an_holiday
  :  date(Dates) & artifact_id(proxyHM, ProxyID)
  <- createWSATContext(ATContext);
     //* add the related ATContext into the Wallet */
     addWalletInfo(ATContext);
     ?artifact_id(proxyHM, ProxyID);
     configure(ATContext)[artifact_id(ProxyID)];
     /* further WSInterface configurations are not
        reported for sake of simplicity */
     !book_hotel(Dates, Res_H);
     !book_accessories(Dates, Res_A);
     !finalize(Res_H, Res_A).

+!book_hotel(Dates, Res_A)
  :  artifact_id(proxyHM, ProxyID)
  <- !createBookingMessage(hotelBooking, Dates, MsgBookHotel);
     doRequestResponse(ProxyID, bookingOperation(MsgBookHotel), HotelResponse);
     !inspect_h_response(HotelResponse, Res_H);
     Res_H == "available".      // fail if not available

+!book_accessories(Dates, Res_H)
  :  artifact_id(proxyTransport, TranID) & artifact_id(proxyPayment, PayID)
     & hPrice(HotelPrice) & tPrice(TransportPrice) & bank_account_id(BankID)
  <- !createBookingMessage(transportBooking, Dates, MsgTransport);
     doRequestResponse(TranID, bookingOperation(MsgTransport), ResponseTransport);
     !createPayMessage(BankID, (HotelPrice+TransportPrice), MsgPay);
     doRequestResponse(PayID, payOperation(MsgPay), ResponsePayment);
     !inspect_acc_responses(TransportResponse, PaymentResponse, Res_A);
      Res_A == "available".     // fail if not available

/* Fail Event Handling */
-!book_an_holiday
  :  artifact_id(proxyHM, ProxyID) & dates(Dates)
  <- !createSubscribeMessage(Dates, MsgSubscription);
     focus(ProxyID);
     subscribeOperation(MsgSubscription)[artifact_id(ProxyID)];
     !finalize("not_available", "").

/* Notification from HM */
+dateNotMoreFull(Dates) [source(proxyHM)]
  :  artifact_id(proxyHM, ProxyID) & dates(Dates)
   <- stopFocusing(ProxyID);
     !book_an_holiday;

/* Finalize */
+!finalize(Res_H, Res_A)
  :  Res_H == "available" & Res_H == "available"
     & wallet_entry(wsatcontext, ATContext) & artifact_id(wsProxyCoord, CoordID)
  <- !createCommitMessage(WS-AT-Context, MsgCommit);
     doOneWay(CoordID, commitOperation(MsgCommit));

+!finalize(Res_H, Res_A)
  :  (Res_H =/= "available" | Res_A =/= "available")
     & wallet_entry(wsatcontext, ATContext) & artifact_id(wsProxyCoord, CoordID)
  <- createRollbackMessage(ATContext, MsgRollback);
     doOneWay(CoordID, rollbackOperation(MsgRollback)).
```

Figure 4.19: Source code snippet of the Booking Requestor Agent (BRA) used in the holiday booking example.

```
1  public class SubscribersMap extends Artifact {
2
3    private HashMap<String, ArrayList<WSMsgInfo>> map;
4
5    /* Artifact initialization */
6    @OPERATION void init(){
7      map = new HashMap<String,ArrayList<WSMsgInfo>>();
8    }
9
10   /* Operation that add a subscriber for the dates given in input */
11   @OPERATION void addSubscriberForDates(ArrayList<String> dates, WSMsgInfo msg){
12     for(String date : dates){
13       if(!map.containsKey(date)){
14         map.put(date, new ArrayList<WSMsgInfo>());
15       }
16       map.get(date).add(msg);
17     }
18   }
19
20   /* Operation that removes a certain subscriber for the dates given in input */
21   @OPERATION(guard="checkDatesPresence")
22     void removeSubscriberForDates(ArrayList<String> dates, WSMsgInfo msg)
23   {
24     for(String date : dates){
25       map.get(date).remove(msg);
26     }
27   }
28
29   /* Operation that returns the list of the subscribers for a specified date */
30   @OPERATION(guard="checkDatePresence") void
31     getSubscribersForDate(String date, OpFeedbackParam<ArrayList<WSMsgInfo>> res)
32   {
33     ArrayList<WSMsgInfo> list = map.get(date);
34     if (list!=null){
35       signal("subscribers", list.clone());
36       res.set((ArrayList<WSMsgInfo>) list.clone());
37     } else {
38       res.set(new ArrayList<WSMsgInfo>());
39     }
40   }
41
42   /* Dates presence guard */
43   @GUARD boolean checkDatesPresence(ArrayList<String> dates, WSMsgInfo msg){
44     for(String date : dates){
45       if(!map.containsKey(date)){
46         return false;
47       }
48     }
49     return true;
50   }
51
52   /* Date presence guard */
53   @GUARD boolean checkDatePresence(String date,
54     OpFeedbackParam<ArrayList<WSMsgInfo>> res)
55   {
56     return map.containsKey(date);
57   }
58 }
```

Figure 4.20: Source code of the SubscribersMap artifact used in the holiday booking example.

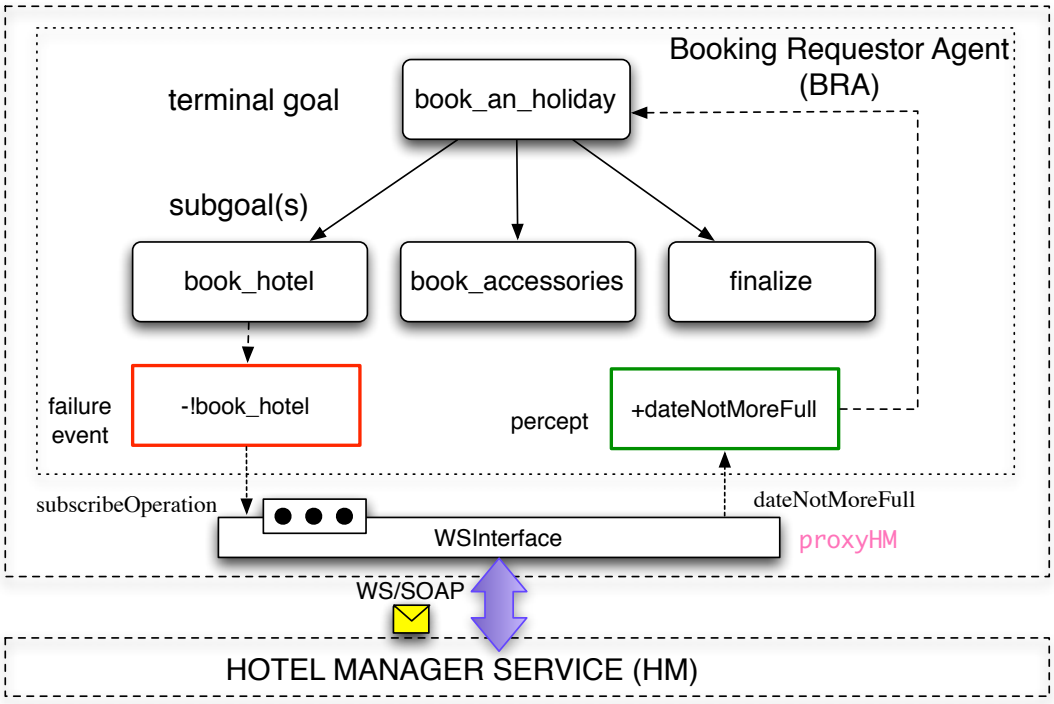


Figure 4.21: Goal decomposition tree for the Booking Requestor Agent. The picture shows the structure of the various plans related to each sub-activity needed to achieve the terminal goal. Notice the interaction with `proxyHM` artifacts, in particular for the subscribe operation, performed after a failure in the `book_hotel` plan, and the execution of a new `book_an_holiday` plan, once a new availability is signaled by the HM.

result. The returned literal is then matched to verify the success of the booking operation. In so doing, a fail event will occur when the booking operation has failed and the `Res_H` is `not_available` (line 28). Thanks to the ***Jason*** execution model, this fail event causes the root plan to fail too. Hence, the failure can be handled by a `-!book_an_holiday` plan (lines 40-46 and Figure 4.21), by which the agent can subscribe to the HM with the aim to be notified when some new availability is signaled. In the hope that some client will cancel a reservation for the desired date the agent: *(i)* focuses the HM proxy (WSInterface) and uses it for self-subscribing to the notification of possibly further availabilities (lines 44-45), then *(ii)* instantiates a sub-goal to rollback the current transaction (line 46), and *(iii)* finally waits for possible HM notifications. The rollback of the WS-AT (managed by the plan at lines 61-65) is managed through a Coordinator Service which is part of the runtime infrastructure of the JaCa-WS platform, together with the set of Web Services required by WS-C specification.

Each BRA subscription is handled within the HM service by the Hotel Notifier Agent, which stores the request in the SubscribersMap business artifact. Meanwhile, if some other agent interacting with the HM cancels its reservation for the subscribed date, that change is signaled –

within the HM side – to the HotelBookingRegistry artifact, which stores the reservations' data. In this case, the Hotel Notifier Agent is supposed to receive a percept from the registry: as soon as a `+data_status_changed` signal is perceived, the Hotel Notifier Agent creates a new sub-goal to process such information, by retrieving the subscribers matching the given date, and by sending back a notification message to the BS who subscribed.

Once a new availability occurs – i.e., the message coming from the HM arrives to the BS – is automatically translated by WSInterface, and is then signaled to the BRA agent. Also in this case, the event is received in form of percept and it succeeds to awaken the focusing BRA: the arriving percept `+dateNotMoreFull(Dates)[source(proxyHM)]` contains a date identifier (`Dates`) by which the agent can match the event and thus recognize it as a meaningful one, w.r.t. its goals (lines 49-52). In so doing, the BRA can now adopt a new instance of the `!book_an_holiday` goal (line 52 and Figure 4.21), by which the activities needed to achieve it are re-planned from scratch. Differently from what happened in the first attempt, the BRA now finds the resources to succeed in booking the hotel for the requested dates (HM response is, in this case, `available`). Given this, the BRA can now proceed with the following activity (`!book_accessories`, line 20). It contacts the TM and PM services (lines 33-36), and, after having received the responses, it can control the results (line 37) and, in so doing, achieve the terminal goal (`!book_an_holiday`). Finally, the WS-AT transaction is committed by exploiting the `Coordinator Service` mentioned before (sub-goal `!finalize(Res_H, Res_A)` at line 21, managed by the plan reported at lines 55-59).

4.6 JaCa-Web: Programming Rich Internet Applications in JaCa

Due to the continuous growth of machine computational power and the overcoming of network speed bottlenecks, the client-side of so-called Rich Internet Applications[6] [FRSaF10] (RIAs) is constantly evolving in terms of complexity. Moreover, thanks to the Web 2.0 transition and the advent of cloud-computing, the gap between Web and traditional desktop applications is tailing off. Web 2.0 applications are starting to share more and more features with desktop applications in order to combine their better user experience with Web benefits, such as distribution, openness and accessibility. One of the most important features of Web 2.0 is a new interaction model between the client user interface of a Web browser and the server-side of the application. Such rich Web applications allow the client to send multiple concurrent requests in an asynchronous way, avoiding complete page reload and keeping the user interface live and responding. Periodic activities within the client-side of the applications can be performed in the same fashion, with clear advantages in terms of perceived performance, efficiency and interactivity [FRSaF10].

So, the more complex web applications are considered, the more the application logic put on the client side becomes richer, eventually including asynchronous interactions – with the

[6]The term was first introduced in a white paper of March 2002 by Macromedia [All02].

user, with remote services – and possibly also concurrency—due to the concurrent interaction with multiple remote services. This situation is exemplified by well known applications such as Gmail [Incc], Google Maps [Incd], etc.

The direction of decentralizing responsibilities to the client is also apparent by considering the new HTML 5.0 standard [Worb], which enriches the set of APIs and features that can be used by web applications on the client side. Among the others, some can have a strong impact on the way an application is designed: this is the case of the Web Worker mechanism [Word], which makes it possible to spawn background workers running scripts in parallel to their main page, allowing for thread-like operations with message-passing as the coordination mechanism. Another one is cross-document messaging [Worc], which defines mechanisms for communicating between browsing contexts in HTML documents.

Besides devising enabling mechanisms, a main issue is then how to design and program applications of this kind [FRSaF10]. A basic and standard way to realize the client side of RIAs is to embed scripts in the page, written in some scripting language such as JavaScript [Wora]. Originally, such scripts were just meant to perform checks on the inputs and to create visual effects. Now instead, due to the new requirements of RIAs, they are progressively becoming one of the reference ways to engineer complex programs. Accordingly, in the last years a very big amount of frameworks and libraries for developing the client side of RIAs have been developed. Main examples are: jQuery [Theb], Ext JS [Sen], Dojo [Thea], etc. However the problem is that scripting languages – like JavaScript or some of its more advanced extensions like the one just cited – have not been designed for programming in the large, so using them to organize, design, implement complex programs without principled software engineering approaches becomes – in general – too complicated, expensive, and unmaintainable [FRSaF10].

To address the problems related to scripting languages, higher level approaches started to proliferate in the very last years. A main example is Google Web Toolkit (GWT) [Gooc], which allows to develop client-side applications with Java. A second example is given by TypeScript [Cor12], a programming language recently introduced by Microsoft. Typescript conceptually extends JavaScript by introducing explicit support for typing, classes, modules, and interfaces. All these features are available at design time for helping developers building robust Web applications, but in the end TypeScript code is translated into plain old JavaScript—i.e., this allows TypeScript code to be possibly executed in any browser, any host, any OS [Cor12]. However, these solutions do not provide significant improvements in tackling all those aspects that are still an issue for object-oriented programming languages, such as concurrency, asynchronous events management, and so on.

We argue then that these aspects can be better captured and tackled by adopting an higher level of abstraction and related programming approaches. This is actually the idea behind the development of Google DART, a new programming language for developing *structured web applications* [Incb]. Besides providing well know features and constructs of object-oriented languages (e.g., typing, inheritance, classes and interfaces, etc.) DART comes along with an actor-based library [Inca] in oder to provide developers better means to deal with issues such as concurrency and asynchronous event management. In our opinion this is a clear evidence

of the need to promote novel and higher level programming approaches for dealing with the complexities introduced by RIAs.

Following this perspective, we realized a proper extension of the JaCa platform named JaCa-Web [MRS11, MSR10] providing an agent-oriented level of abstraction to design, program and execute the client-side of rich Internet applications. By exploiting the JaCa programming model, we directly program the Web 2.0 application as a normal JaCa multi-agent system, composed by a workspace with one or multiple ***Jason*** agents working with an artifact-based CArtAgO environment, including a set of predefined artifact types specifically designed for the Web domain. Generally speaking, agents are used to encapsulate the logic of control and execution of the tasks that characterize Web 2.0 application, while artifacts are used to implement the environment needed for executing the tasks, including those coordination artifacts that can ease the coordination of the agents' work.

4.6.1 The JaCa-Web Platform

JaCa-Web is an extension of the JaCa platform, specifically conceived for allowing developers to build the client-side of Web 2.0 rich Internet applications as JaCa multi-agent systems. As with the two JaCa extension presented before (JaCa-Android and JaCa-WS), the application and programming model adopted in JaCa-Web are inherited from the JaCa platform, which here has been properly enriched with a new layer composed of specific artifacts that enable the development of RIAs. As depicted in Figure 4.17, the JaCa-Web platform is currently implemented on top of existing technologies. Java applets are used as the enabling technology to download and manage the execution of JaCa-Web applications inside the user's browser. So, as with any other Java applet, once the user points her browser to the URL of a JaCa-Web application, a pop-up is displayed in order to ask the permission to run the targeted application. If the user grants the permission, the browser starts downloading a signed Java applet containing both the JaCa-Web runtime and the application sources. The LiveConnect [SUNc] open-source library is exploited to enable JaCa-Web programs access DOM (Document Object Model) elements – e.g., buttons, text fields, radio buttons, etc. – displayed in the application Web page.

As soon as the page and the JaCa-Web applet are downloaded by the browser, the application is launched–i.e., are created the workspaces, the initial set of agents and artifacts. Among the predefined types of artifact that are available in the new layer introduced by the JaCa-Web platform, two main ones are the Page artifact and the HTTPService artifact.

The former allows to: *(i)* access and change the application's web page – internally exploiting the LiveConnect library – allowing for dynamically updating its content, structure, and visualization style; *(ii)* make events related to user actions on the page observable to agents as percepts. An application may either exploit directly the Page artifact or define its own extension with operations and observable properties linked to the specific content of a web page.

The HTTPService provides instead basic functionalities to interact with a remote HTTP service, hiding the use of sockets and low-level mechanisms. Analogously to the Page artifact, it

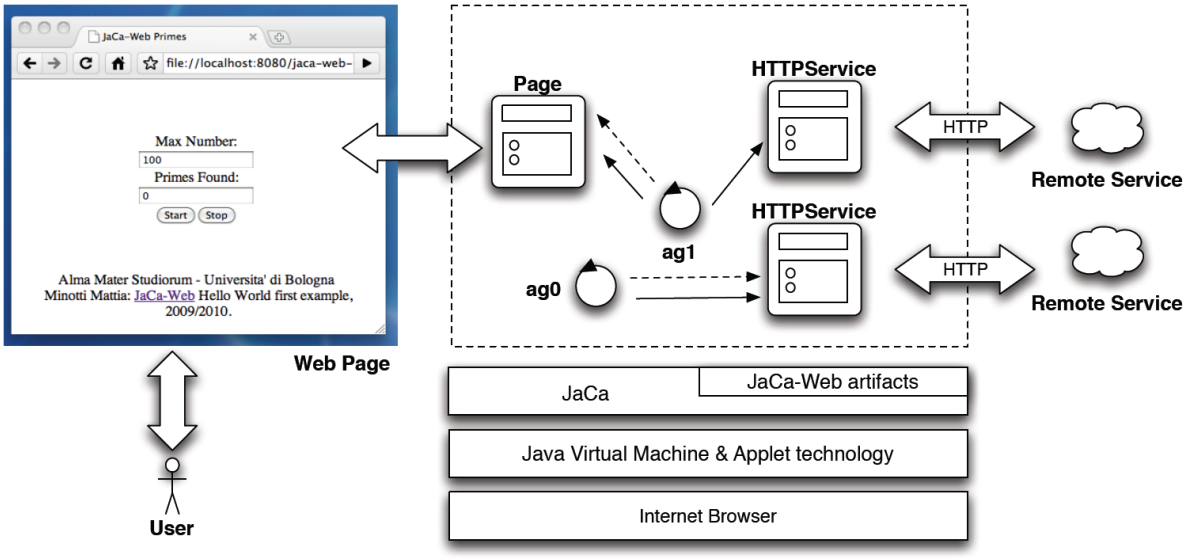


Figure 4.22: Architecture of the JaCa-Web platform – with in evidence the different technologies upon which the platform is based – and of a generic application running on top of it.

can be used as it is – providing actions to do raw HTTP requests – or can be extended providing an higher level application specific usage interface hiding the HTTP level.

4.6.2 A Concrete Case Study

To stress the features and test-drive the capabilities of the platform, in this section is presented a relevant example of Web application developed with JaCa-Web. The example is useful in particular to stress some of the keys complexities of RIAs identified in Section 4.6: *(i)* the management of concurrent behaviors, *(ii)* the management of asynchronous events coming either by the user and external services, and *(iii)* the integration of the handling of such events with the application autonomous behavior.

The application is about searching products and comparing prices from multiple services (Figure 4.23). To this end, we imagine the existence of N REST services that offer a set of product catalogs describing products features and prices, codified in some standard machine-readable format (e.g., XML or JSON [Int]). The client-side in the Web application needs to search all services for products that satisfy a set of user-defined constraints (e.g., a specific product description, price lower than a certain threshold, etc.).

The application also needs to periodically monitor services so as to search for new offerings of the same product. A new offering satisfying the constraints should be visualized only when its price is more convenient than the current best price for that product. The client may finish its search and monitoring activities when some user-defined conditions are met—e.g., a

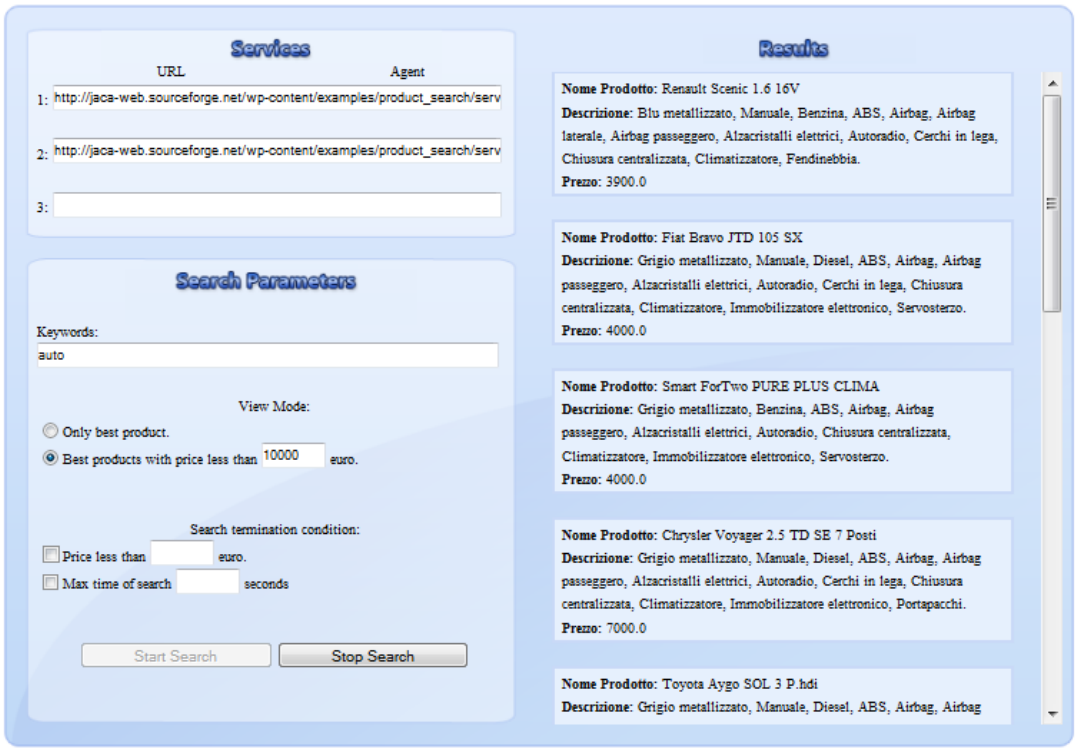


Figure 4.23: A screenshot of the JaCa-Web product search application in action. In evidence: *(i)* the preferences specified for the search by the user (*left*), and *(ii)* the current search result displayed by the application (*right*).

certain amount of time is elapsed, a product with a price less than a specified threshold has been found, or the user interrupts the search with a click on a proper Stop Search button in the page displayed by the browser (see Figure 4.23). Finally, if such an interruption took place, by pressing again the Start Search button it must be possible to let the search continue from the point where it was blocked. The characteristics of concurrency and periodicity of the activities that the client-side needs to perform and the asynchronous events it needs to manages, make this case study a significant prototype of a typical Web 2.0 rich Internet application.

The solution adopted to realized the described application includes two kinds of agents: *(i)* a usr-assistant agent – which is responsible of setting up the application environment and managing the interactions with the user – and *(ii)* multiple product-finder agents, which are responsible to periodically interact with remote product services to find the products satisfying the user-defined parameters. To aggregate data retrieved from services and coordinate the activities of the usr-assistant and product-finder agents we introduced a ProductDirectory artifact (Figure 4.26), while a MyPage artifact and multiple instances of Service artifacts (a specialization of the HTTPService artifact) are used respectively by the usr-assistant and product-finders to

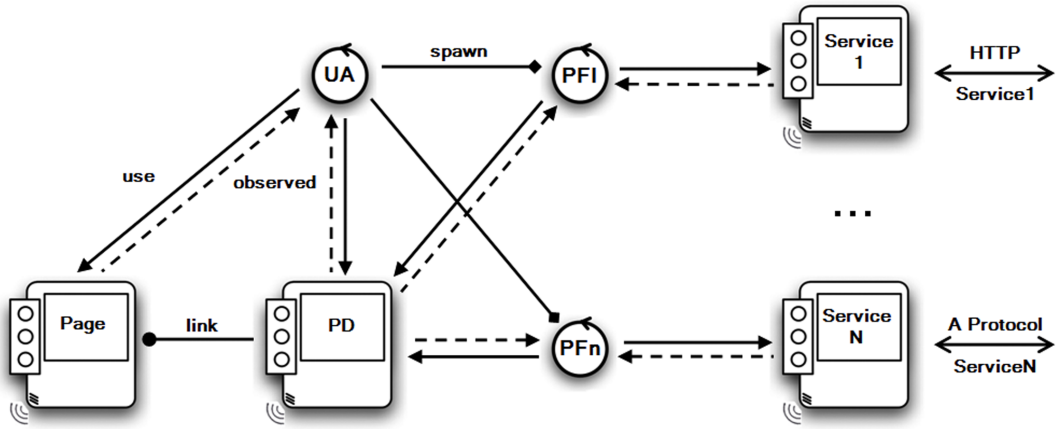


Figure 4.24: Client-side architecture of the product search application in terms of agent, artifacts, and their interactions. UA is the usr-assistant agent, PFs are the product-finder agents, PD is the ProductDirectory artifact and finally Services are the specialization of the HTTPService artifact, used to communicate with the external product services.

interact with the user and with remote product services.

More in detail, the usr-assistant agent is the first agent booted on the client side, and it setups the application environment by creating the ProductDirectory artifact and spawning a number of product-finder agents, one for each service to monitor. Then, by observing the MyPage artifact, the agent monitors user actions and inputs. In particular, the web page provides controls to start, stop the searching process and to specify and change dynamically the preferences related to the products to search, along with the conditions to possibly terminate the process (see Figure 4.23). Button click events are mapped onto `stateSearch(X)` (where `X` can be either `"start"` or `"stop"`) observable events generated by the MyPage artifact, while other MyPage specific observable properties – i.e., `searchPreferences` containing the user-defined preferences for the search and `terminationCondition` specifying the search termination condition – are used to make observable the input information specified by the user.

The usr-assistant reacts to changes to these observable properties, and interacts with product-finder agents via message passing to coordinate the search process. The interaction is also mediated by the ProductDirectory artifact, which is used and observed by both the usr-assistant and product-finders. In particular, this artifact provides a usage interface with operations to aggregate product information found by *product-finder*s—in particular `addProducts`, `removeProducts` to respectively add and remove products from the current list. The artifact stores and makes observable the products found so far through the `matchingProducts` observable property. Changes of this observable property are handled by the usr-assistant agent which reacts in order to update the application GUI with current search results.

```
1  /* product-finder agent */
2  service_url("...").
3  monitoring_frequency(...).
4
5  +start_search
6    <-  !setup_tools;
7        !search.
8
9  +!search: searchPreferences(Prefs) & monitoring_frequency(Delay)
10   <-  !request_products(Prefs, ProductList);
11       !process_products(ProductList, ProductsToAdd, ProductsToRemove);
12       addProducts(ProductsToAdd);
13       removeProducts(ProductsToRemove);
14       .wait(Delay);
15       !search.
16
17 +!request_products(Prefs, ProductList) : art_id(proxy, ArtID) & service_url(SrvURL)
18   <-  !prepare_http_request(Prefs, SrvURL, ReqURL);
19       send("GET", ReqURL, Response) [artifact_id(ArtID)];
20       !get_products_from_service_response(Response, ProductList).
21
22 +searchPreferences(Preferences)
23   <-  .drop_intention(search);
24       !search.
25
26 +stop_search
27   <-  .drop_intention(search).
28
29 +!setup_tools
30   <-  makeArtifact("proxyHTTP","jacaweb.HTTPService",[],Id);
31       focus(Id);
32       +art_id(proxy, Id).
33
34 +!process_products
35   <- ...
```

Figure 4.25: Source code snippet of a product-finder agent used in the product search example.

The full source code of the application can be consulted on the JaCa-Web official website [Andb], where the interested reader can also find a running instance of the application that can be used for tests[7]. Here we just report a snippet of a product-finder agent source code (Figure 4.25), with in evidence: *(i)* the plans used by the agent to react to changes to the search state (lines 5-7 and 26-27) communicated by the usr-assistant via `tell` messages—which cause the addition or removal of a `search` goal, *(ii)* the plan used to achieve the `!search` goal (lines 9-15), periodically – on the basis of the `monitoring_frequency` initial belief – getting the product list by means of the `!request_products` sub-goal (line 10) and then updating the ProductDirectory accordingly by adding new products and removing products no more available (lines 12-13), and *(iii)* the plan used to adapt the search as soon as a change of the MyPage artifact `searchPreferences` observable property is perceived (lines 22-24).

[7] `http://jaca-web.sourceforge.net/?page_id=88`

```
1  public class ProductDirectoryArtifact extends Artifact{
2
3    private ArrayList<Product> ProductsList;
4
5    void init(){
6      ProductsList=new ArrayList<Product>();
7      defineObsProperty("matchingProducts", new ArrayList<Product>());
8    }
9
10   /**
11    * Add products to the directory
12    * @param product An XML representation of the product
13    * @param serviceId
14    */
15   @OPERATION void addProducts(List<String> products, String serviceId){
16     List<Product> newProducts = new ArrayList<Product>();
17     for (String currProduct: products) {
18       /* Transforms the XML representation in a corresponding Product
19        * Java Object and add it to the newProducts list */
20     }
21     /* Private method that updates the matchingProducts observable property */
22     this.addNewProducts(newProducts);
23   }
24
25   /**
26    * Remove products from the directory
27    * @param product An XML representation of the product
28    * @param serviceId
29    */
30   @OPERATION void removeProducts(List<String> products,String serviceId){
31     List<Product> delProducts = new ArrayList<Product>();
32     for (String currProduct: products) {
33       /* Transforms the XML representation in a corresponding Product
34        * Java Object and add it to the delProducts list */
35     }
36     /* Private method that updates the matchingProducts observable property */
37     this.removeProducts(delProducts);
38   }
39
40   /* Private methods */
41   private void AddProducts(){...}
42   private void removeProducts(){...}
43 }
```

Figure 4.26: Source code snippet of the ProductDirectory artifact used in the product search example.

4.7 Concluding Remarks

Following the discussion made in Section 3.5, in this chapter we presented JaCa, which is both an agent-oriented development platform and programming approach targeting the general purpose development of MASs. The platform is built upon the integration of two existing agent-oriented technologies: the BDI-based ***Jason*** agent-oriented programming language (Section 3.2.2), and the CArtAgO environment framework (Section 3.3.2).

The main feature/novelty of JaCa w.r.t. other state-of-the-art agent-oriented APLs/frame-

works/platforms is its programming model, which relies on the synergistic integration of the agent and the environment dimensions. Such an integration has been engineered on the basis of a dedicated action and perception model (described in Section 4.1), which has been studied to make BDI-based agents work effectively in endogenous environments, without the need to rely on ad-hoc bridges/integration mechanisms between the two dimensions.

The first part of the chapter describes the general JaCa programming model (Section 4.2) and puts the focus on some of its more relevant features (Section 4.3). Then have been presented some of the most relevant extensions of the JaCa platform that we have developed in order to exploit and stress the JaCa programming model in some of the most relevant and modern application domains (Section 4.4, Section 4.5 and Section 4.6). For each application context considered: *(i)* we provided an overall description of the domain and the motivations that lead us to test/apply JaCa there, *(ii)* we introduced the extended version of the JaCa platform that has been engineered in order to apply its programming model in the considered application domain, and finally *(iii)* we presented and discussed a relevant case study.

The main objective of this chapter – and also of our work with JaCa – is not demonstrate/-prove that a specific application/program – e.g., the different case studies discussed – or that all the applications of a specific domain are *best solved in general* by using the JaCa technology. Indeed, on the one side it is not possible – for many reasons – to provide an exact quantitative evaluation and comparison among classic and JaCa-based solutions; moreover on the other side both JaCa and all its extensions are still far beyond from being considered as robust and mature as reference technologies available in the state-of-the-art for coding programs in the application domains considered. On the contrary, our general aim is to study the effectiveness – and also the limitations – of the JaCa programming model – and in particular the synergistic integration of the agent and environment dimensions – for developing MASs in some of the most relevant application domains, effectively tackling relevant programming issues for such domains such as: *(i)* the management of asynchronous interactions/events, *(ii)* the integration of the handling of such events with the application autonomous behavior, *(iii)* the implementation of context-sensitive behaviors, and *(iv)* the programming and management of concurrent behaviors. The examples described in this chapter show in detail how all these issues can be easily tackled in JaCa.

Concluding, besides all the aspects already considered, the synergistic integration of the agent and the environment dimensions on a unified programming model give developers the opportunity to move from one application context to another in a quite straightforward manner. Indeed, by knowing the JaCa programming model, they can continue to engineer the applications business logic by suitably defining the behavior of ***Jason*** agents, and they only need to acquire the ability to work with the artifacts that are specific of the new application context, which wrap and hide all the complexities and low-level technical aspects of the new domain.

A more comprehensive discussion about the good findings, the current weaknesses and limitations of JaCa and its related extensions is postponed to Chapter 6. This is an intentional choice. Indeed, given the strong connections with JaCaMo that will be presented only in the next chapter (Chapter 5), we decided to provide only one general and comprehensive descrip-

tion of these aspects after having presented both JaCa and JaCaMo, avoiding the repetition of concepts and issues that are in common with the two platforms and programming approaches.

5 The JaCaMo Platform

Following the path we started to delineate with JaCa, in this chapter we describe JaCaMo [BBH+11]. JaCaMo is both a concrete programming approach and development platform for the engineering of MASs, which is rooted on a comprehensive integration of three multi-agent programming dimensions, namely the agent, environment, and organization. It has been realized taking as a starting point three existing agent-oriented technologies: ***Jason*** for programming agents (Section 3.2.2), CArtAgO (Section 3.3.2) for programming environments – and in particular the previous work and the expertise maturated during the study of their synergistic integration in JaCa (Chapter 4) – and $\mathscr{M}$OISE (Section 3.4.2) for programming organizations.

Then, beyond a simple technological integration, a key point has been the synergistic integration of the related programming (meta-)models so as to come up with an approach that allows programmers to take advantage of such connections to simplify the development of complex MASs. The result and main contribution of this integration is the JaCaMo conceptual framework and platform, which provides high-level first-class support for developing MASs exploiting the agent, environment and organization dimensions, preserving a strong separation of concerns but, at the same time, exploiting such dimensions in a synergistic way. The chapter is organized as follows:

- Initially is described the JaCaMo programming approach (Section 5.1).
- Then in Section 5.2, are described both the JaCaMo programming model – focusing in particular on the synergies among the different programming dimension that emerged during its definition – and the runtime infrastructure at its support.
- Finally, to provide a concrete evaluation, are described a couple of relevant real world projects in which JaCaMo has been used with success (Section 5.3).

5.1 The JaCaMo Approach

A JaCaMo multi-agent system – i.e., a software system programmed in JaCaMo – is given by an agent organization programmed in $\mathscr{M}$OISE, organizing autonomous agents pro-

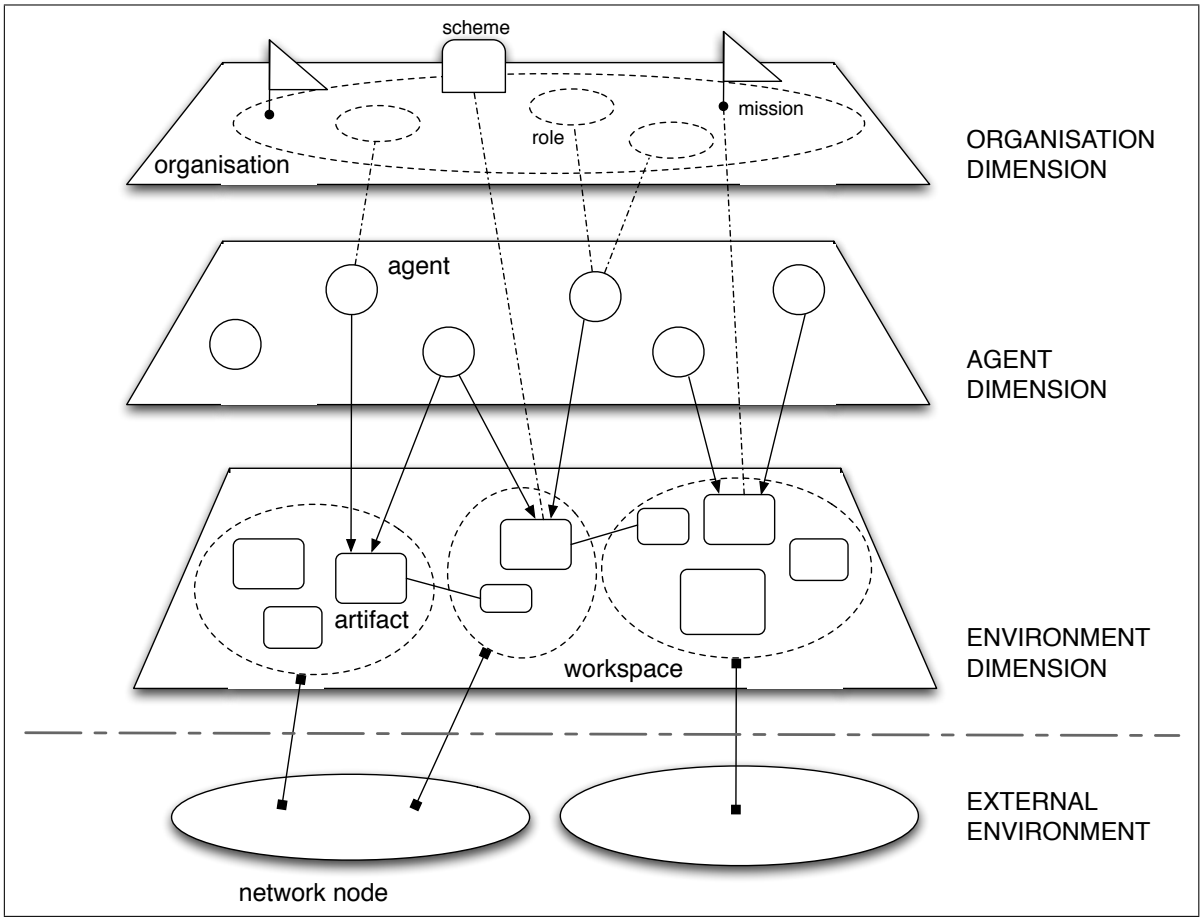


Figure 5.1: Overview of a JaCaMo multi-agent system, highlighting its three dimensions.

grammed in ***Jason***, working in a shared distributed artifact-based environment programmed in CArtAgO (see Figure 5.1).

So, JaCaMo integrates these three platforms by defining in particular a semantic link among concepts of the different programming dimensions – agent, environment and organization – at the meta-model and programming levels, in order to obtain a uniform and consistent programming model aimed at simplifying the combination of those dimensions when programming multi-agent systems.

Even if it is not possible, for many reasons, to provide an exact quantitative evaluation, we argue that the approach simplifies MAS programming, and makes it possible to have cleaner and typically shorter programs. This is possible because of the reasoning/interpreting/monitoring engines/infrastructures that are available in the three platforms incorporated into JaCaMo, and the interfacing between them which also make automatic many things that programmers would normally have to worry about themselves.

5.1.1 Overview of the JaCaMo Programming Meta-Model

Figure 5.2 shows the integrated programming meta-model of JaCaMo. The abstractions strictly related to each specific dimension are grouped by dashed lines. Being an integrated *programming* meta-model – that is, a meta-model focused on programming abstractions and constructs – it does not include concepts or abstractions which are not part of the programming languages or frameworks—for instance, the concept of *intention* which is part of the ***Jason*** runtime but not of the ***Jason*** programming language is not included. Besides presenting the main concept of JaCaMo programming, the main objective of this meta-model is to show explicitly the dependencies, connections and conceptual mappings between the abstractions belonging to the different programming dimensions.

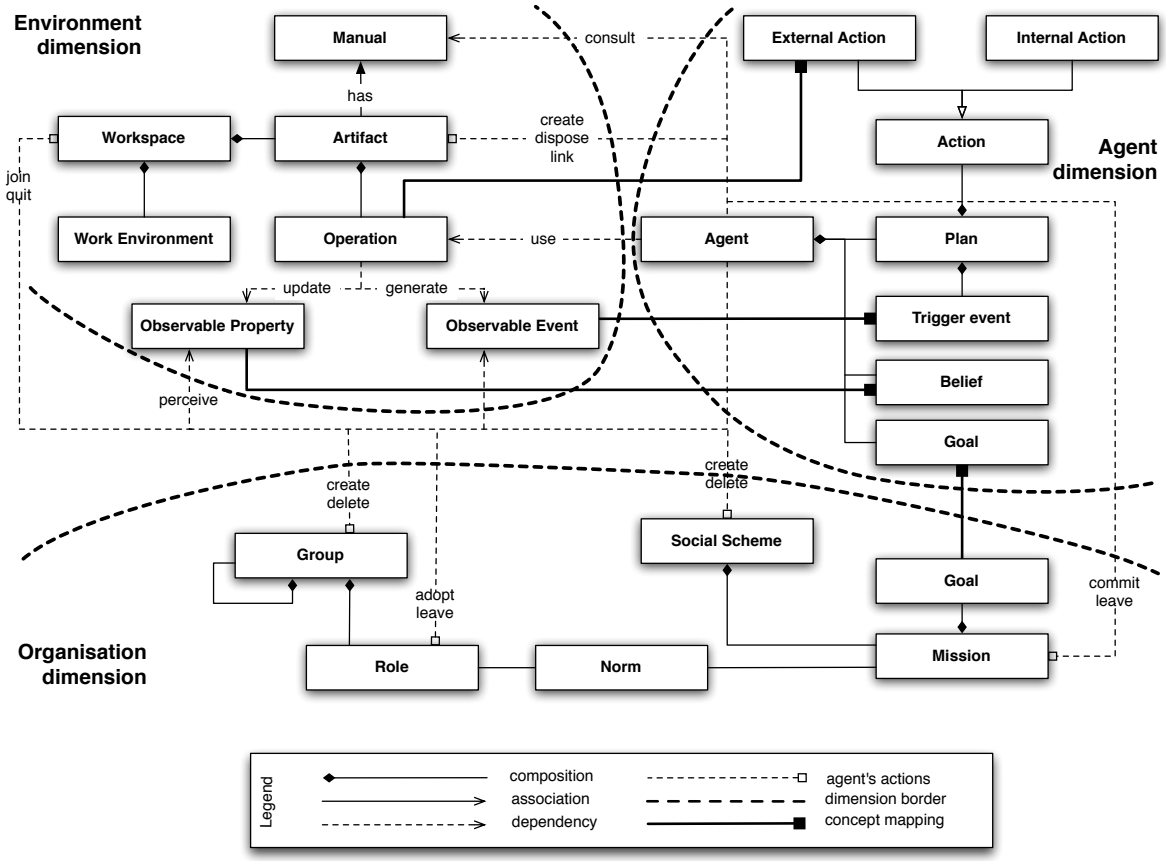


Figure 5.2: JaCaMo Programming Meta-Model. For readability reasons cardinalities are not reported.

The abstractions belonging to the agent dimension, related to the ***Jason*** meta-model, are mainly inspired by the BDI architecture upon which ***Jason*** is rooted. Instead, the abstractions on the

environment side refer to the A&A meta-model, which is at the basis of the CArtAgO environment framework. Both ***Jason*** and CArtAgO, along with their main programming abstractions and features, have been described in detail in the previous chapter during the presentation of the JaCa platform. Therefore the interested reader can refer to Chapter 4 for a comprehensive description of the elements in Figure 5.2 that refer to the agent and environment dimensions.

Finally, the abstractions belonging to the organizational side are related to the $\mathscr{M}$OISE organization model, which has been presented in Section 3.4.2. As in the previous case, the interested reader can go back to the referenced section in order to have a complete view of the abstractions shown in Figure 5.2 which belong to the organization dimension.

5.1.2 Synergies Among the JaCaMo Programming Dimensions

In Figure 5.2, the synergies among the three programming dimensions are represented by connections terminating with a square (either filled or not). The connections terminating with a filled square are the most important part of our integrated meta-model as they *explicitly represent the synergies and the conceptual mappings* we have identified during the definition of our integrated programming approach: such connections provide for free (i.e., transparently and without extra programming efforts) the integration between the different dimensions, an integration that in other approaches must be programmed by users in an *ad hoc* manner.

The connection links between the Agent and Environment (A-E) dimensions are realized by exploiting the action and perception model described in Section 4.1 and already implemented in JaCa (Section 4.2). Basically, these links are given by semantically mapping agents' external actions into artifacts' operations, and artifacts' observable properties and events into agents' percepts (leading to beliefs and triggering events). This means that – at runtime – in a JaCaMo program an agent can do an action α if there is (at least) one artifact providing α as operation—as usual, if more than one such artifact exist, the agent may contextualize the action explicitly specifying the target artifact. On the perception side, set of observable properties of the artifacts that an agent is observing are directly represented as (dynamic) beliefs in the agent's belief base—so, as soon as their values change, new percepts are generated for the agent that are then automatically processed (within the agent reasoning cycle) and the belief base updated. So, in programming an agent it is possible to write down plans that directly react to changes in the observable state of an artifact or that are selected based on contextual conditions that include the observable state of possibly multiple artifacts. Briefly recalling what already described with more detail in Section 4.1, this mapping brings significant improvements w.r.t. the classical action and perception model provided in general by agent programming languages:

- *Dynamic action repertoire:* the repertoire of an agent's action is dynamic and can be extended/reshaped dynamically by agents themselves by creating/removing artifacts. This is an improvement w.r.t. existing agent programming languages, where the set of (external) actions available to an agent is given by the set of actuators that are statically defined for the agent, typically implemented in an *ad-hoc* way.

- *More expressive action model:* by inheriting the action-as-a-process semantics of the operation model defined for artifacts, the expressivity of the agent action model is increased in various ways [RSP12].

- *Well-defined success/failure semantics:* there is no more the burden of understanding if an action done by an agent succeeded or not by reasoning about the beliefs (percepts) in the agent code. This in general simplifies agent programming and reduces agent program size, although agents might still need to reason about beliefs to ensure successful action execution in non-deterministic environments.

The connections that terminate with a non-filled square represent a set of predefined actions that agents can perform and which are mapped into operations in a set of predefined artifacts available in each JaCaMo application. These actions refer to the basic functionalities provided by the overall infrastructure, including the environment and organization dimensions. This makes it possible in particular to avoid the introduction of *ad-hoc* specific mechanisms to exploit infrastructure services concerning organization and coordination, for instance to adopt a role or to interact with a tuple space. Furthermore, since artifacts can be created and disposed of dynamically, this makes it possible (also for agents) to update and adapt the infrastructure itself at runtime.

This idea was explored to effectively connect the Organization and the Environment dimensions (O-E), in particular by uniformly designing the organizational infrastructure as part of the (artifact-based) environment in which agents are situated [HBKR10]. In such an approach, the different concrete computational entities aimed at managing, outside the agents, the current state of the organization in terms of groups, social schemes, and normative states are reified in the environment by means of *organizational artifacts*, encapsulating and enacting the organization behavior as described by the organization specifications. From an agent point of view, such organizational artifacts provide those actions that can be used to pro-actively take part in an organization (for example, to adopt and leave particular roles, to commit to missions, to signal to the organization that some social goal has been achieved, etc.), and provide specific observable properties dynamically to make the state of an organization perceivable along with its evolution. Besides, they provide actions that can be used by *organizational agents* to manage the organization itself. The specific concrete types of organizational artifacts introduced in the JaCaMo programming model will be briefly described in Section 5.2.

Overall, the O-E mapping provides some important outcomes which are important from a design and programming perspective:

- *Uniformity:* the same action and perception model is used also to enable the interaction between agents and the organization, without the need for introducing specific *ad-hoc* primitives and mechanisms concerning the organization.

- *Distribution:* the organization management infrastructure is distributed, in terms of collections of (interconnected) artifacts possibly belonging to different workspaces running on distinct network nodes.

- *Dynamism:* organizational agents can change dynamically the shape of an organization by acting on the set of organizational artifacts used by the agents to interact.
- *Heterogeneity:* the O-E mapping also opens the way for scenarios in which heterogeneous agents – belonging to different platforms and programmed in different languages – could easily take part in a JaCaMo organization, as soon as they have been equipped with the capability to work within artifact-based environments.
- *High-level reorganizing capabilities:* the $\mathscr{M}$OISE-based specifications of the organization are part of the information made observable by organizational artifacts to agents. This means that there is the potential for agents that understand the $\mathscr{M}$OISE specification formats to reason about the organizations in which they partake and therefore to change them at runtime. This allows for complex on-the-fly re-structuring of computational systems to be done at very high level.

Finally, as part of the Agent and Organization (A-O) mapping, goals defined at the organization dimension (those goals that are to be achieved by the organization, i.e., the overall system) are mapped into individual agent goals (see Figure 5.2), which agents may decide to adopt or not. This delegation of goals from the organization to the agents is expressed by obligations. The state of goals from the organization dimension (which agents should fulfill them and when) is maintained by an organizational artifact and is displayed as obligations for the agents (e.g., agent a is obliged to fulfill goal x in one week). An obligation is fulfilled when the corresponding goal is achieved by the agent before the deadline. At the agent side, when such obligations are perceived, and the agent chooses to adopt it, a corresponding (individual) agent goal is created. It should be noted that, besides being free to adopt or not the goals from the organization, the agent is also free to decide which courses of action should be used to achieve each goal, and that in turn might mean adopting further individual goals. As we can see, the mapping of goals prescribed by the organization into agent individual goals is under the complete control of the agent's decision making process (so as to preserve its autonomy).

5.2 Impact on Multi-Agent System Programming: The JaCaMo Programming Model and Platform

A main objective of the integration of the dimensions described in Section 5.1 is to simplify the programming model adopted in the development of complex multi-agent systems. Accordingly, in this section we concretely describe the impact of the dimension integration on programming, by using a toy example called *Building-a-House* – included with the JaCaMo distribution available on JaCaMo website [Oli] – which is simple yet effective in showing the integration of the dimensions at the programming level.
Before focusing on the example and on the programming model, first we give a brief overview of the structure of a general JaCaMo application at runtime and of the supporting JaCaMo infra-

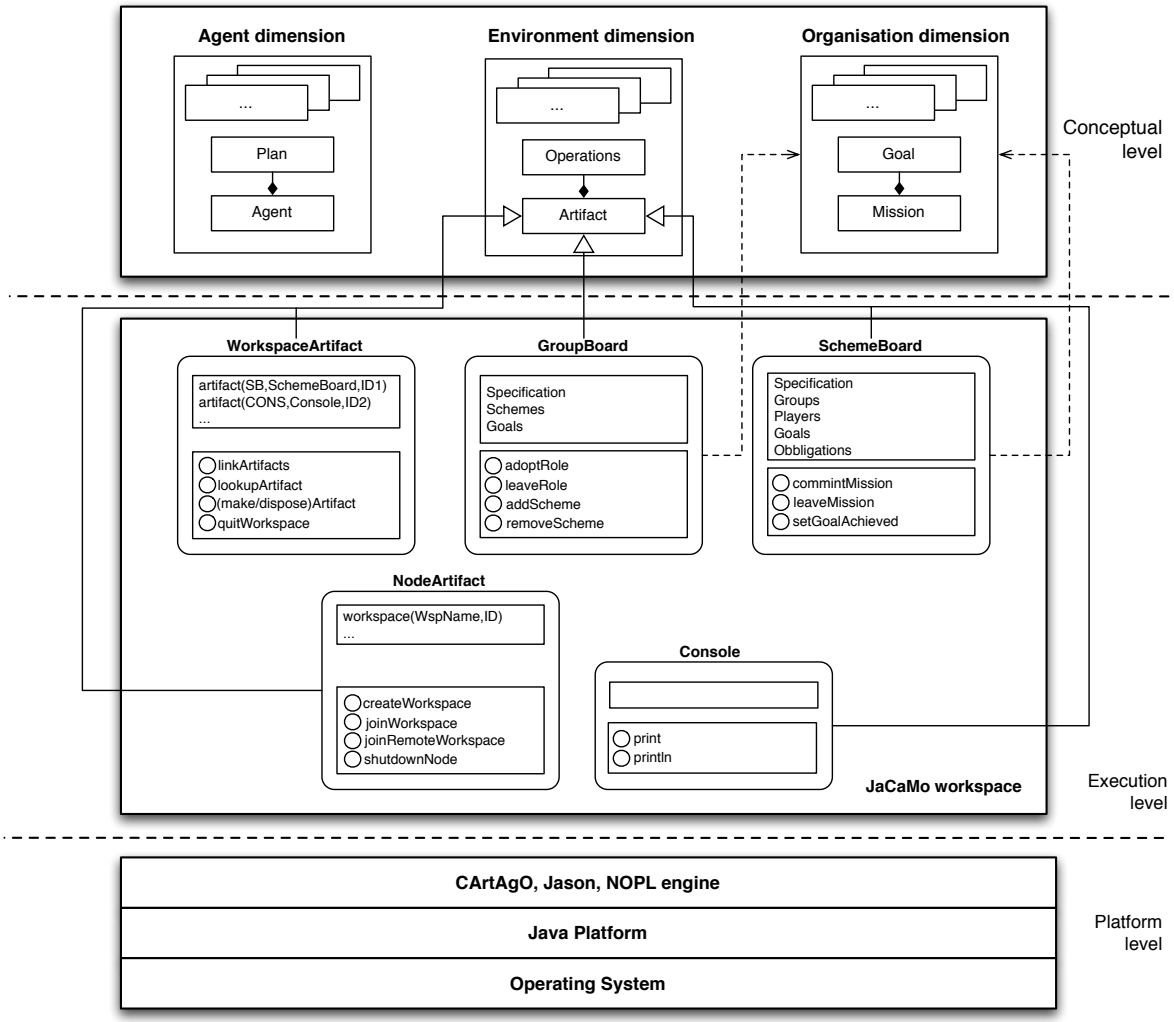


Figure 5.3: The JaCaMo platform runtime, with the predefined set of artifacts available in all JaCaMo applications.

structure. The overall picture of runtime support provided by JaCaMo is shown in Figure 5.3. A JaCaMo (distributed) application runs on top of possibly multiple JaCaMo infrastructure nodes. At the platform (infrastructure) level, each infrastructure node integrates the ***Jason***, CArtAgO, and $\mathscr{M}$ OISE platforms as well as the required interfacing technology, running on top of a Java Virtual Machine which itself makes transparent the access to all resources of the operating system. Then, at the execution level, a JaCaMo application is represented by an organization composed by one or multiple workspaces, running on JaCaMo nodes. Agents running on a JaCaMo node can join and work concurrently in multiple workspaces, including remote ones

(i.e., workspaces not hosted by the same node where they are executing). Among the artifacts that populate a JaCaMo workspace, a fundamental role is played by those encapsulating infrastructural functionalities related to the agent, environment, and organization management. They represent a direct reification of the concepts defined in the meta-model (the conceptual level in the figure). These include artifacts used to manage/inspect the structure of the environment itself – WorkspaceArtifact and NodeArtifact, providing functionalities to manage/inspect respectively the set of artifacts inside a workspace and the set of workspaces inside the environment – as well as *organizational artifacts*, introduced in the previous section.

In particular, the basic types of organizational artifacts adopted in a JaCaMo application include:

- OrgBoard artifacts — used to keep track of the current state of deployment of the organizational entities in the overall (one instance for each organization).

- GroupBoard artifacts — used to manage the life-cycle of specific groups of agents. For instance, if an agent chooses to adopt a role in a particular group, it will perform the adoptRole operation (action) on the GroupBoard artifact representing the group.

- SchemeBoard artifacts — used to support and manage the execution of social schemes. As an example, an agent can commit to a mission or tell the organization it has achieved one of the social goals it was assigned to in a particular mission by executing actions such as commitMissions and setGoalAchieved on the specific SchemeBoard representing the scheme.

Besides the actions, organizational artifacts have specific observable properties to make the dynamic state of an organization observable. For example, the GroupBoard artifact has an observable property about the available roles in a group, therefore agents can find out the existing roles and then reason about them so as to autonomously decide whether to adopt a role or not.

From a computational behavior point of view, organizational artifacts encapsulate and enact the organizational behavior specified in $\mathscr{M}$OISE. To this end, a specific language called NOPL (Normative Organization Programming Language) is adopted as a target language into which $\mathscr{M}$OISE specifications are translated [HBB10]. So organizational artifacts embed a NOPL interpreter to provide the organizational infrastructure needed to manage a $\mathscr{M}$OISE organization at runtime.

As a final remark, the set of organizational artifacts of the same organization are connected together by means of the *linkability* features provided by CArtAgO Section 4.2.2, to execute operations among artifacts. This is necessary, under the hood, to keep consistent the overall state of an organization infrastructure that is distributed over various separate artifacts. Such a distribution of a JaCaMo application over multiple workspaces and network nodes are essential to scale with organization and application complexity.

5.2.1 The Building-A-House Example

The example is about a multi-agent system representing the inter-organizational workflow involved in the construction of a house. An agent called Giacomo owns a plot and wants to build a house on it. In order to achieve this overall goal, first Giacomo will have to hire various specialized companies (the *contracting phase*), and then ensure that the contractors coordinate and execute the various tasks required to build the house (the *building phase*). For each company, there is a *company agent* – possibly running on a different network node – participating in the contracting phase and then, possibly, in the building one.

In this simple example, the organization is composed of a single workspace called `HouseBuildingWsp`. Giacomo is responsible for creating and setting up the workspace, creating also the artifacts that will be used to interact with company agents. It is worth remarking that in larger applications – like the ones described in Section Section 5.3 – multiple workspaces possibly distributed on multiple nodes are often used, implementing then a distributed environment and organization.

Contracting Stage: Agent-Environment Programming

In the contracting phase the objective for Giacomo is to hire one company (the one that offers the cheapest service) for each of the several tasks involved in building the house, such as site preparation, laying floors, building walls, building the roof, etc. The same company can be hired for more than one task, if they have more than one working field and offer the cheapest service in more than one of the required tasks. An auction-based mechanism is used by Giacomo to select the best company from among the available ones for each of the tasks; one auction is run concurrently for each of the tasks. The auction starts with the maximum price Giacomo can pay for a given task, and companies that can do that kind of task may offer a lower price than the current bid. After a given deadline (unknown to bidders), Giacomo clears the auctions and selects the contractors to build its house based on the lowest bid by the time the auction closed.

To exemplify the Agent-Environment programming in JaCaMo, auctions are here realized by means of an artifact providing auction functionalities. There will be one instance of such artifact (created by Giacomo) for each of the house-building tasks, concurrently used by Giacomo and the company agents. The left-hand side of Figure 5.4 shows the source code of the auction artifact implemented using the CArtAgO API, and its right-hand side shows and an excerpt of a company agent, implemented in ***Jason***, that uses an instance of that artifact. Auction artifacts have only a `bid` operation – used by company agents to submit a new bid – and there are four observable properties: the task name (`task`), the maximum value the agent that created the auction is willing to pay for the service (`maxValue`), the current lowest bid (`currentBid`), the agent that placed that bid (`currentWinner`). The bidding operation – which is seen on the agent side as a `bid` action – simply updates the current bid and winner if a better bid is submitted, or it fails if the bid is higher than the current one.

On the agent side, the company agent has the initial goal `!discover_art("auction_for_SitePreparation")` to discover the auction for

```
public class AuctionArt extends Artifact {

  void init(String taskDs, int maxValue)  {
    defineObsProperty("task",taskDs);
    defineObsProperty("maxValue",maxValue);
    defineObsProperty("currentBid",maxValue);
    defineObsProperty("currentWinner",
      "no_winner");
  }

  @OPERATION  void bid(double bidValue) {
    ObsProperty cb =
      getObsProperty("currentBid");
    ObsProperty cw =
      getObsProperty("currentWinner");
    if (bidValue < cb.intValue()) {
      cb.updateValue(bidValue);
      cw.updateValue(getOpUserName());
    } else {
      failed("invalid_bid");
    }
  }
}
```

```
!discover_art("auction_for_SitePreparation").

my_price(1500).

i_am_winning(Art) :-
  .my_name(Me) &
  currentWinner(Me)[artifact_id(Art)].

+currentBid(V)[artifact_id(Art)]
  : not i_am_winning(Art)
    &  my_price(P) & P < V
  <- bid(math.max(V-150,P))[artifact_id(Art)].

+!discover_art(ToolName)
  <- joinWorkspace("HouseBuildingWsp");
     lookupArtifact(ToolName,ToolId);
     focus(ToolId).

+!contract("SitePreparation",GroupBoardId)
  <- adoptRole(site_prep_contractor)
     focus(GroupBoardId).

+!site_prepared
   <- ... // actions to prepare the site..
```

Figure 5.4: (*left*) Source code of the auction artifact. (*right*) Source code of a company agent.

preparing the site. To achieve this goal, the agent has a plan triggered by a new goal to achieve event `+!discover_art(ToolName)`. In that plan, the agent first joins the workspace, retrieves the unique identifier of the artifact with the specified name (`lookupArtifact` action), and then uses it to start observing the artifact by executing the `focus` predefined action. By doing that, the artifact's observable properties are automatically mapped into the agent's belief base. Changes in the belief base produce events that can be handled by plans.

In this specific case, the company agent has a plan to react to changes in the `currentBid` (the one with `+!currentBid(V)` triggering event), in order to place a new bid (`bid` action execution in the plan body[1]) in case the agent is not the current winner[2] and also the current bid is higher than the minimum price this agent can offer (stored as the initial belief `my_price(1500)`).

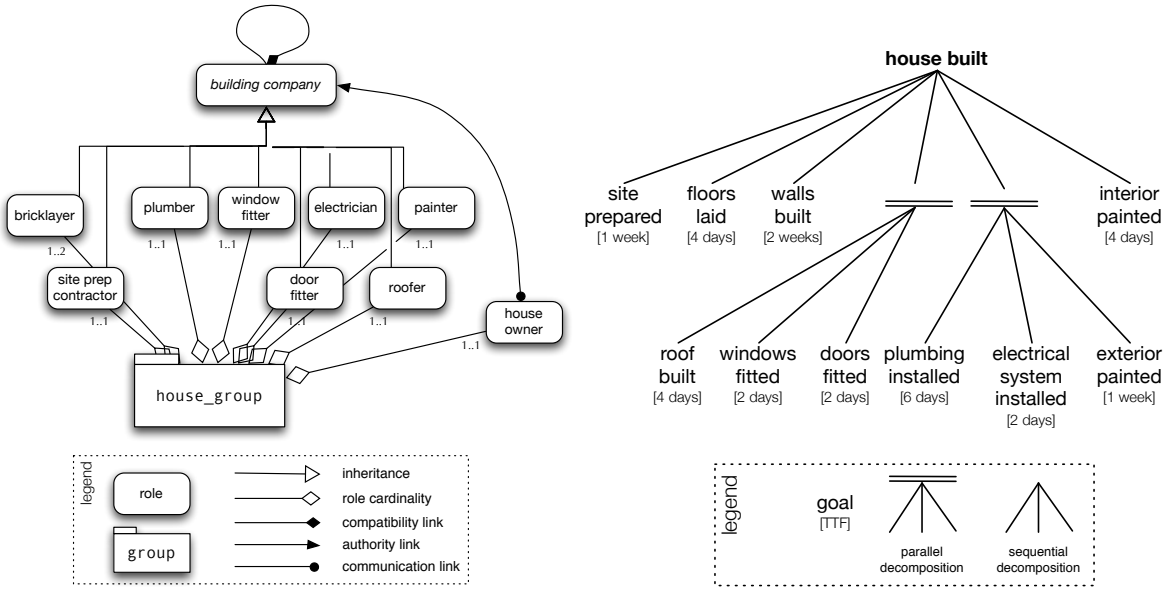
[1]Recalling from Section 4.2.1, the annotation `[artifact_id(Art)]` that appears next to plan triggering events related to belief updates (e.g., `+currentBid(V)`) and to actions execution (e.g., `bid`) is used in general to explicitly retrieve or specify information about the artifact which is the context of the observable property or the operation. This is necessary when multiple artifacts with the same observable properties and operations are used concurrently (as in this case).

[2]`i_am_winning(Art)` here is a Prolog-like rule which is specified initially as part of the belief base of the agent. The conclusion of the `rule` is true if the current value of the `currentWinner` belief (acquired by the agent through an observable property) coincides with the agent's name (retrieved through the ***Jason*** internal action `.myName`).

Building Stage: Incorporating the Organization Dimension

After the companies have been hired, in the building phase the contractors have to execute their own tasks on time and in coordination with each other. Some tasks depend on other tasks being completed first, while some tasks can be done in parallel with some others, as represented by the following workflow (where '`;`' is used for sequence and '`|`' for parallel composition):

`a ; b ; c ; (d | e | f) ; (g | h | i) ; j`



norm	role	mission / goals
`n1`	house owner	house built
`n2`	site prep contractor	site prepared
`n3`	bricklayer	floors laid, walls built
`n3`	roofer	roof built
`n3`	window fitter	windows fitted
`n3`	door fitter	doors fitted
`n4`	plumber	plumbing installed
`n3`	electrician	electrical system installed
`n3`	painter	interior painted, exterior painted

Figure 5.5: Structural Specification (*left*), Functional Specification (*right*), and Normative Specification (*bottom*).

At this stage, the organization dimension is used to help the coordination and the monitoring of the companies that will build the house. Companies that won the auction will join the organization playing specific roles and, by doing so, become responsible for some goals in the

overall process of building the house. The roles and goals of this organization are specified in Figure 5.5 using the *M*OISE notation [HSB07].

The structural specification defines a group (`house group`) where company agents will play sub-roles of `building company` and the Giacomo agent plays `house owner`. The organization constrains the number of role players in the group (cf. the role cardinality arrow). In our application, most of the roles must have only one player, except `bricklayer` that can have one or two players (the winner(s) of auctions for tasks "Floors" and "Walls" are supposed to adopt this role). Agents however can play several roles as allowed by the compatibility link: an agent can play more than one sub-role of `building company`, but the same agent cannot play a company role and `house owner` at the same time. For instance, the agent that won the auction for task "WindowsDoors" is supposed to adopt both the `window fitter` and `door fitter` roles. The specification also includes a communication link from companies to the owner, allowing them to communicate, and an authority link from the owner to the companies.

The functional specification decomposes the organization goals into sub-goals, defines the sequence in which each will be achieved, and gives a time-to-fulfill (TTF) for each sub-goal. These goals are assigned to roles by norms as defined in the table that appears at the bottom of Figure 5.5. For instance, norm `n3` states that whenever an agent adopts the role `bricklayer`, it is obliged to achieve goals `floors laid` and `walls built`. Of course, this obligation is active only when the preceding goals have already been achieved (`site prepared`, in this example).

```
1  !have_a_house.     // initial goal
2
3  +!have_a_house     // plan to achieve the have_a_house goal
4    <-  !contract;
5        !execute.
6
7  +!contract <- ... // plan to manage the first stage (auction)
8
9  +!execute          // create the artifact to manage the group
10   <-  makeArtifact("hsh_group","GroupBoard", ["src/house-os.xml", ... ], GrArtId);
11       adoptRole(house_owner)[artifact_id(GrArtId)];
12       !contract_winners("hsh_group");
13       ...
14
15 +!contract_winners(GroupBoard)
16   <-  for ( currentWinner(Ag)[artifact_id(ArtId)] ) {
17         ?task(Task)[artifact_id(ArtId)];
18         .send(Ag, achieve, contract(Task,GroupBoard))
19       }.
```

Figure 5.6: Source code snippet of the Giacomo agent.

Once the auctions are finished, Giacomo adopts the role `house owner` in the group and asks the auction winners to adopt the corresponding roles. The code excerpt in Figure 5.6 illustrates how these steps are programmed. Giacomo has the initial `!have_a_house` goal and

plan to handle it, by instantiating two subgoals – `!contract` and `!execute`, corresponding to the contracting stage and the building stage that are achieved sequentially. As soon as the `!contract` goal is achieved, which means that the auctions are concluded, the plan managing the building stage is executed, in which a GroupBoard called hsh_group artifact is created (specifying some parameters, including the *M*OISE specification in XML), the house_owner role is adopted by executing an `adoptRole` action and then a sub-goal is instantiated to contract winners. In the plan handling that sub-goal, the agent asks (by means of an `achieve` message) to all the company agents that won the auctions to sign the contract and finally adopt the role corresponding to the specific task. The required information for all that is made available as observable properties of the auction artifacts.

When the company agents receive the request sent by Giacomo, they adopt the roles by acting on the group artifact. Back to the source code of the company agent (Figure 5.4), this is done by the "`+!contract("SitePreparation", GroupBoardId) <- ...`" plan. The group artifact ensures that the specified organization constraints are satisfied at all times.

The main purpose of the scheme artifact is to keep track of which goals are ready to be pursued (those whose preceding goals in the functional specification have already been achieved) and create obligations for the agents accordingly. Initially, only the `site prepared` goal can be pursued, thus only the obligation `obligation(companyB, achieved(s1, site_prepared), "4/1/2013")` is created, where `companyB` is the agent playing the role `site prep contractor`, `s1` is the identification of the scheme instance, and 4/1/2013 is a week after the start of the building work. Such obligations are observed by the agents and corresponding goals are automatically created[3]. In the example, the goal `!site_prepared` is created within the `companyB` agent, which can then react by executing plan of the form `!+site_prepared <- ...` (see Figure 5.4). As soon as other goals become ready to be pursued, new obligations are created and the agents can then work toward the goals at the right moment. In the case of parallel goals, several obligations are created and the agents will work in parallel, as expected from the specification.

A main advantage of this approach is that by simply changing the scheme specification – which can be done by the designer or by the agents themselves – at very high level, say to change the order or the dependencies among goals, we will change the overall behavior of the agent team without changing a single line of their code. We can see the scheme specification as the program for the social coordination and the scheme artifact as its interpreter. This artifact also manages the state of the obligations, checking, for instance, their fulfillment or violations. This feature is very useful for Giacomo who wants to monitor the execution of the scheme to ensure that the house is built correctly and on time.

[3]This is done under the hood by predefined plans from a library made available with JaCaMo. This library facilitates the programming of (specially norm-abiding) agents and in this application we provided such plans to every agent

5.3 Using JaCaMo for Real World Applications

In order to assess the applicability, advantages, and limitations of the approach, JaCaMo is being used in various projects that involve the development of real world agent-based applications. In this section we provide a brief overview of some of them. The interested reader can find at the JaCaMo web site [Oli] an up-to-date list of the projects in which JaCaMo is being used, each one described in detail. All these projects share some elements of complexity (distribution, openness, dynamism, need of flexibility, autonomy) which make it possible to effectively stress test JaCaMo's different programming dimensions and the usefulness of the integration.

5.3.1 Engineering Smart Co-Working Spaces

The first project is about rooms management in smart co-working spaces (e.g., a school, an office building, etc.); initial results can be found in [Sor11, SBPS11]. This is a fairly common Ambient Intelligence (AmI) scenario consisting in the management of a room allocation problem in the context of a smart co-working space [Kam10] where people can book and use rooms according to their needs and to the current occupancy schedule. Rooms are equipped with different items – projectors, whiteboards, TV sets etc. – and are tagged by different usage categories – meeting room, teaching room etc. – on the basis of user-defined room requirements (e.g., number of seats, availability of specific equipments, etc.). Each room is augmented with a proper set of sensors and actuators for managing room temperature, lights level, presence of equipment and people, etc.

The application has to set an autonomous and adaptive room management behavior in accordance with: *(i)* the events that are currently held – e.g., regulating the room temperature in accordance with the number of the event's participants, automatically turning off the lights for teaching events involving a projector etc. – and also *(ii)* on the basis of (re)allocation of the rooms according to the user requests. Users – professors, engineers etc. – can: *(i)* demand the scheduling of new events in the building, *(ii)* modify or cancel scheduled events, *(iii)* register themselves as an event's participant, and *(iv)* inquire information about scheduled events. According to the request details, made in term of room category, layout, number of seats and required equipment, the event will be either allotted to the appropriate room, if any is available, or discarded.

The frequent changes in the activity of people – e.g., deadlines approaching, start/end of school courses etc. – coming in the co-working space, make it possible to have spikes of request types – lectures, brainstorming sessions, meetings etc. – that must be correctly handled by the system at the best of its capacity, (re)allocating rooms to host more constrained event types. This application presents features such as distribution, openness, need of flexibility etc.

Figure 5.7 provides an overview of a JaCaMo solution to the problem. A virtual organization is used to manage and coordinate the functioning of the whole system. The organization structure, defined by the $\mathscr{M}$OISE structural specification, defines a set of groups, sub-groups and related roles for: *(i)* managing the rooms' behavior (responsibility of the agents playing

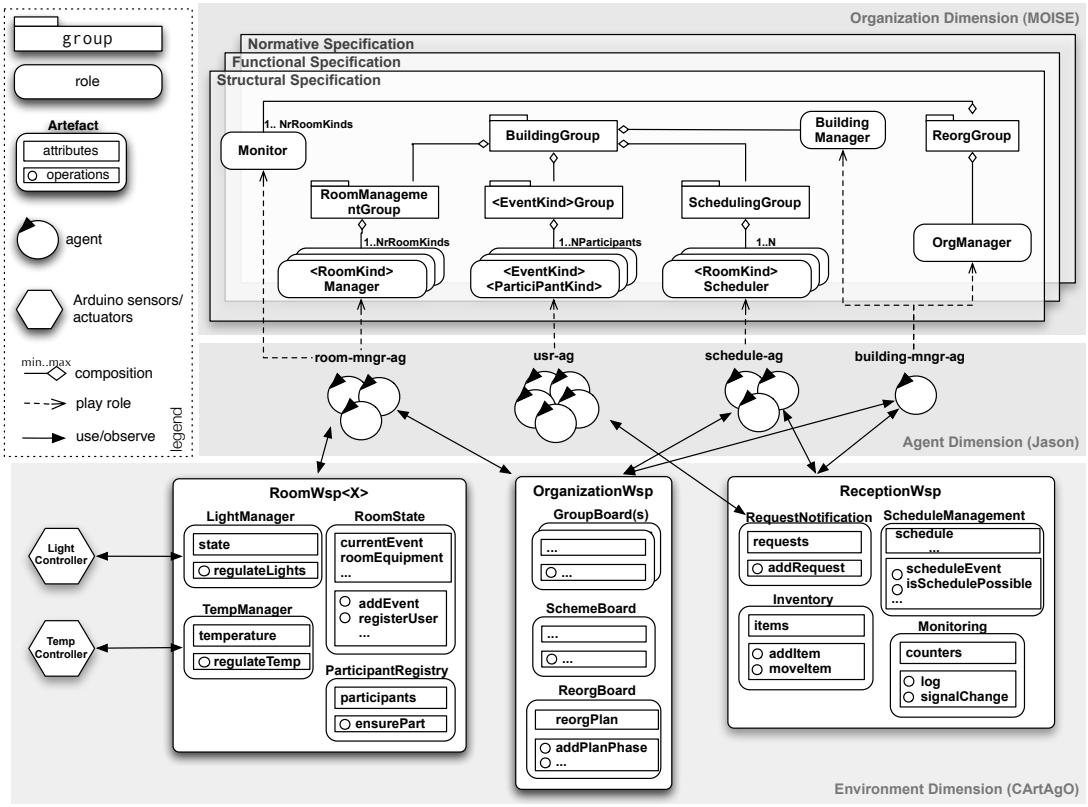


Figure 5.7: Architecture of the JaCaMo application for the governance of room allocation in a smart co-working space.

the <RoomKind>Manager role), *(ii)* monitoring spikes of incoming requests and changes in the equipment position, adapting the functioning of the system accordingly (responsibility of the agents playing the Monitor role), *(iii)* managing the current schedule on the basis of incoming requests (responsibility of the agents playing one of the <RoomKind>Scheduler roles), and *(iv)* managing user interactions with the system (responsibility of the agents playing one of the <EventKind><ParticipantKind> roles).

The application has a main organizational goal (ManageBuilding) that has been decomposed through the $\mathscr{M}$OISE functional specification in a proper hierarchy of sub-goals. Then, through the $\mathscr{M}$OISE normative specification, proper groups of goals – i.e., missions – defined in the functional specification are assigned to agents' roles by means of norms, hence mapping organizational goals into agent goals. For sake of simplicity here we do not go into the details of this mapping.

The whole application is distributed among several workspaces physically situated in different network nodes. A set of RoomWsp<X> – one for each room, where the X stands for the room number/id inside the building – have been introduced for managing rooms behavior.

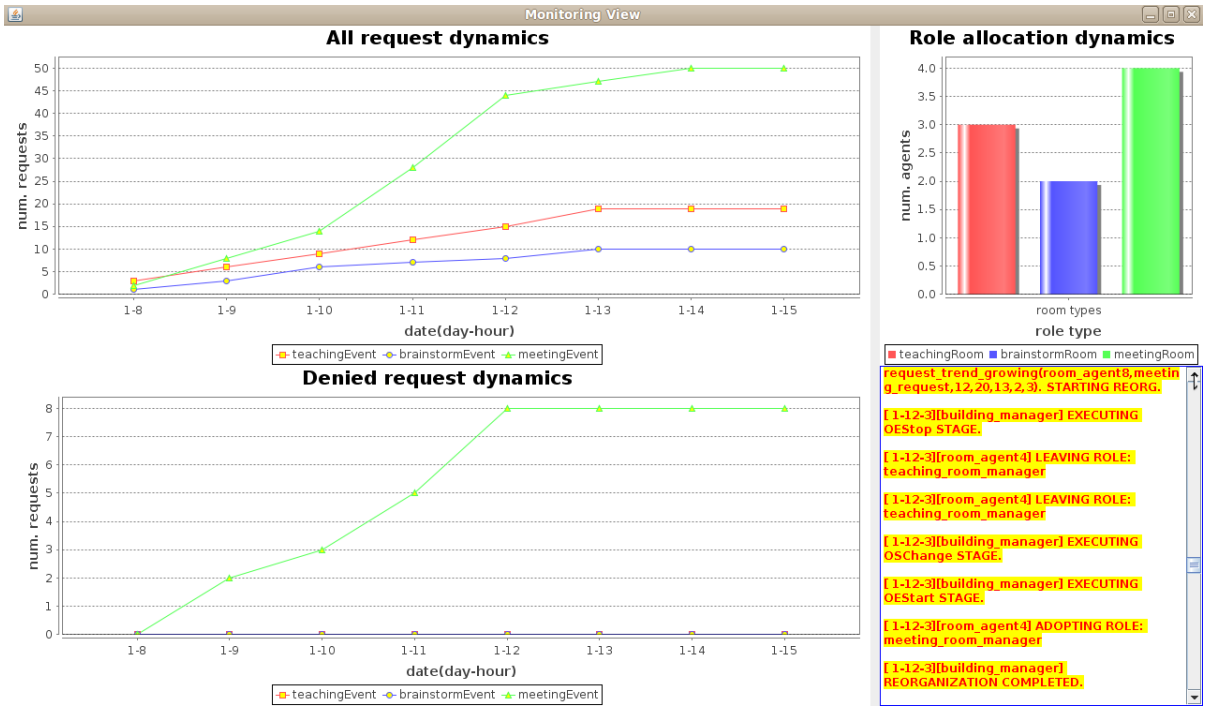


Figure 5.8: Screenshot of the test environment showing requests dynamics (issued at *top-left* and denied at *bottom-left*), roles allocation dynamics (*top-right*) and log info (*bottom-right*).

Each `RoomWsp<X>` workspace – deployed on a dedicated workstation machine located inside the respective `X` room – contains a set of artifacts enabling the access to all the sensors and actuators available inside the room[4].

For each `RoomWsp<X>` a room-mngr-ag agent playing the `<RoomKind>Manager` role is supposed to manage the room behavior – accordingly to the kind of events assigned to that room – by properly exploiting the available artifacts that interact with sensors and actuators. room-mngr-ag agents also play the `Monitor` role, thus allowing for the distribution of the task of observing inadequate system states and adapting it accordingly. As soon as some known trend is observed – e.g., the constant growth of the of brainstorming events during a fixed period – these agents works cooperatively, using the ReorgBoard artifact for sharing and coordinating a proper reorganization plan, trying to adapt – if possible – the system in accordance with the current dynamics—e.g., increasing the cardinality of agents permitted to play the `BrainstormingRoomManager` role.

Finally a `ReceptionWsp` workspace, physically deployed in a workstation machine at the reception of the building, contains the artifacts used for: *(i)* managing incoming requests from users (stored inside the RequestNotification artifact), *(ii)* providing the event scheduling ser-

[4]These artifacts – which have been realized in the context of another project [Alec] – wrap the access to Arduino [Ard] sensors and actuators dislocated inside the rooms.

vice (thanks to the ScheduleManagement artifact) , and *(iii)* managing information about the equipment dislocation (stored inside the Inventory artifact). For what concerns the scheduling service, the schedule information are stored inside the ScheduleManagement artifact observable properties while the scheduling service functionalities are managed by a dynamic pool – the pool size can vary on the basis of the current amount of incoming requests – of scheduler agents, playing a specific `<RoomKind>Scheduler` role, working with both the ScheduleManagement and RequestNotification artifacts.

In order to evaluate the governance application described, a virtual test and monitoring environment (see Figure 5.8) was designed, which allows to stress the adaptation capability of the realized application. By carefully setting values for parameters of the test environment that determine the request dynamics – i.e., distribution of requests for different event types during a given period of time – a situation is created whereby during one day one event type sees a strong increase in request numbers whereas other, compatible, types experience a normal demand (e.g., a quickly augmenting number of meetings against a fair number of brainstorm sessions or lectures). The expected outcome of this test scenario consists in a shift of agents' roles from the management of a room hosting an unconstrained event type to that of a room manager for event types under pressure.

Figure 5.8 shows an example of such adaptation dynamics. As can be seen from the monitoring window output, the smart building application experiences a great increase in requests for meetings, which leads to a considerable number of unsatisfied requests due to of the lack of appropriate rooms (see *Denied request dynamics* in Figure 5.8). When the agents of the application notice this, they trigger a reorganization. The results of such reorganization process can be seen in the output presented in the log-window: a teaching room manager agent leaves its old role and adopts the more constrained one, causing a decrease of the incoming requests.

5.3.2 An Agent-Based Machine-To-Machine Management Infrastructure

This project concerns the realization of an agile governance application for a Machine-To-Machine (M2M) management infrastructure. M2M refers to technologies allowing the realization of automated and advanced services and applications (e.g., smart metering, traffic redirection, and parking management) that largely make use of smart devices (sensor and actuators of different kinds, possibly connected through a Wireless Sensor and Actor Network (WSAN)) communicating without human intervention.

In this context, the Senscity FUI project [Ora] proposes an infrastructure to enable the deployment of city-scale M2M applications that share a common set of devices and network services. Implementing such an infrastructure raises the problem of providing an agile governance with suitable scalability in different dimensions [Fir10]. In fact, it is hardly possible to define all M2M requirements due to its heterogeneity and openness. For this purpose, in collaboration with Orange Labs and based on the communication and IT infrastructure proposed in the Senscity Project, a JaCaMo-based governance application has been developed in order to study how to properly adapt and evolve the Senscity infrastructure without running into scalability

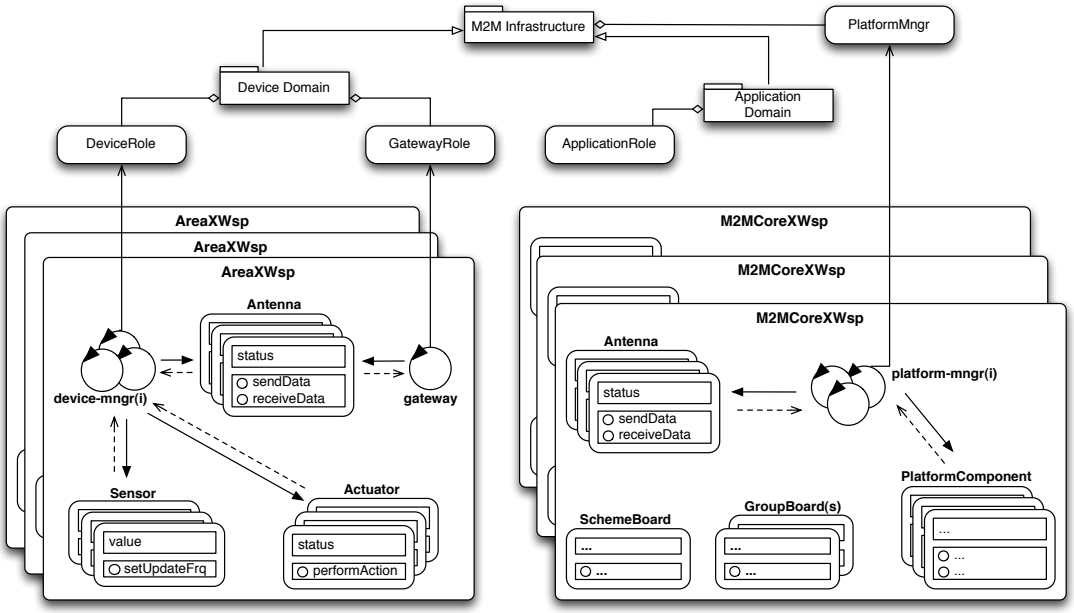


Figure 5.9: Abstract architecture of the JaCaMo-based governance application for the Senscity M2M infrastructure.

issues [PPR11]. The governance infrastructure is evaluated using a smart parking management scenario, in which an M2M system monitors the parking occupation in order to reduce traffic and to guide drivers through the streets.

The governance application developed in JaCaMo (see Figure 5.9) is designed in terms of an organization that provides agents a global strategy to manage and coordinate the functioning of the whole system, distributed among several workspaces. Agent roles, to which proper missions have been assigned, are introduced for covering all the required functionalities of the governance application. In particular, we mention here some of the key roles:

- `PlatformMngr` role: for managing one of the M2M infrastructural core nodes (as defined in the ETSI (European Telecommunications Standards Institute) specification [ETS10]). Besides providing administration and management functionalities related to the M2M node, agents playing this role are also responsible for interfacing end-user applications to the physical world by communicating with agents playing the `Gateway` role (see below).

- `Device` role: for managing a device in a WSAN area group.

- `Gateway` role: responsible for managing the communications between agents playing the `PlatformMngr` role and `Device` role.

- `Application` role: responsible for providing a service using sensors data and controlling actuators.

The governance infrastructure is distributed among several workspaces. For each WSAN area group we have a `AreaXWsp` workspace – the `X` stands for an area id – in which an agent playing the `Gateway` role collects data sent from all the agents playing the `Device` role – data is sent through the `Antenna` communication artifact – in that area. Agents playing the `Device` role manage – via proper CArtAgO artifacts enabling the access to both sensors and actuators – a M2M device which interacts with the physical world to realize the desired M2M function.

For each M2M infrastructural core node a `M2MCoreXWsp` has been introduced. A dynamic pool of `platform-mngr` agents playing the `PlatformMngr` role regulate the functioning of the node in accordance with the current workloads experimented in the M2M infrastructure, by properly using a set of dedicated `PlatformComponent` artifacts. Such artifacts are used to provide agents the access to all the functionalities related to a M2M core node such as: transaction management for both sensor reading and complex actuator commands, the possibility to enable/disable the interfacing of applications to the managed WSAN, etc.

In the context of the considered smart parking scenario, the governance application allows the adaptation of the M2M infrastructure in several situations: *(i)* steep increase/decrease in requests coming from the different applications, *(ii)* changes in the WSAN topology covered by applications (e.g., adjusting the area to monitor for the urban planning service from district-scale to city-scale), and *(iii)* integration of new client applications (e.g., smart parking integrated with a multi-modal transportation system). This is done by dynamically adapting the role cardinalities in the organization specification, thus changing the size of the agents pools playing in the organization. Changes in the physical structure of the WSAN managed by the M2M infrastructure are handled by dynamically deploying (destroying) a workspace for each WSAN introduced (removed) and then adequately registering the new information in the governance infrastructure.

5.4 Concluding Remarks

Following the discussion made in Section 3.5, in this chapter we presented JaCaMo, a concrete programming approach and development platform for the engineering of MASs, which is the natural evolution of JaCa (Chapter 4) since it is rooted on the synergistic integration of three multi-agent programming dimensions: the agent, the environment – these first two are already covered by JaCa – and the organization.

From a technological point of view, JaCaMo is built upon three existing agent-oriented technologies: ***Jason*** (Section 3.2.2), CArtAgO (Section 3.3.2) and $\mathscr{M}$ OISE (Section 3.4.2). Like in the case of JaCa, the key novelty of the platform is its programming model. It has been devised by defining in particular a semantic link among concepts of the different programming dimensions at the meta-model and programming levels, in order to obtain a uniform and consistent programming model aimed at simplifying the combined use of those dimensions when

programming multi-agent systems.

The chapter first describes the JaCaMo programming approach Section 5.1, first presenting an overview of its integrated programming model (Section 5.1.1) and then focusing on its main features (Section 5.1.2)—i.e., the semantic links and connections that have been devised between the different programming dimensions. Then, in Section 5.2 has been discussed the impact of the adoption of such programming approach in practice, through the use of a simple guiding example (i.e., the building house example). Finally, the chapter is concluded by the description of two relevant case studies in which JaCaMo has been successfully applied (Section 5.3). To conclude, on the same line of the discussion made in Section 4.7:

- The general objective of the chapter – and of our work with JaCaMo – is to investigate and stress the effectiveness – and also the weaknesses and limitations – of the platform and its agent-oriented unified programming model for developing MASs able to deal with issues and complexities that are relevant for the application domains considered. So, like in the case of JaCa and for the same reasons – i.e., difficulty to provide a clear quantitative evaluation, and technology still at the prototype level – we do not claim that a particular problem or class of problems are *generally best solved* by using the JaCaMo technology. However, we argue that both the description of the JaCaMo programming model and the presented case studies demonstrate that the approach simplifies MASs programming, in particular when there is the need to take into the account different programming dimensions, hence finally making it possible to have cleaner and typically shorter programs. In particular, in addition to the expertise and the findings matured with JaCa, the complex scenarios considered (Section 5.3) – i.e., need to take into the account highly dynamic domains, complex and interchangeable interaction dynamics – given us the opportunity to appreciate the value of the organization dimension for MASs programming – which was absent in JaCa – and especially the usefulness of its synergistic integration with the agent and the environment dimensions.

- As already introduced in Section 4.7, a complete discussion about the good points, current issues and limitations of both the JaCaMo framework and its programming model will be done in the next chapter (Chapter 6). In this way we are able to make a comprehensive discussion of these aspects, covering and describing only in one place – and in an organized manner – those that are in common between JaCa and JaCaMo.

6

Agent Oriented Programming: Shifting from the Development of Intelligent Software Systems to General Purpose Computing

As already discussed in the introduction of this dissertation (Chapter 1), historically agent-oriented programming has been mainly applied in distributed artificial intelligence contexts, as a programming paradigm to develop *intelligent software systems*.

However, taking a broader perspective, we argue that the exploitation of both JaCa and JaCaMo platforms in some of the most relevant application domains, allowed us to highlight, in a concrete manner, how an agent-oriented level of abstraction can help tackling some of the main complexities that arise in modern distributed and concurrent software development (Chapter 1). In details:

- The BDI control architecture makes it possible to realize a *basic form* of integration between pro-active and event-driven behaviors. As already mentioned in the introduction, this is a relevant programming issue in mainstream programming approaches since it can lead to several problems such as: inversion of control, proliferation of callbacks, fragmentation of the application business logic, etc. We showed concretely how to realize this kind of integration in several of the examples that we have presented (e.g., the producer-consumer (Section 4.2), the smart navigator (Section 4.4.2), the product search example (Section 4.6.2), etc.). A detailed analysis concerning the integration of autonomous and event-driven behaviors in JaCa and JaCaMo is postponed to Section 7.3.1. There, we compare the support provided by simpAL to deal with this issue with both mainstream approaches and the support provided by JaCa and JaCaMo.
- Autonomous behaviors can be *explicitly* mapped onto agents, possibly choosing different kinds of concurrent architectures according to the needs—either using multiple agents to concurrently execute tasks, or using a single agent to manage the interleaved execution of multiple tasks.

- The capability to apply different courses of actions (plans) on the basis of current context information can be used to easily code context-sensitive behaviors, and finally for introducing a *basic form of polymorphism* in agent programs.

- Differently from mainstream programming paradigms – e.g., object-oriented programming – and emerging ones – e.g., actor-oriented programming – *different abstractions* are used to model *conceptually different things*—i.e., active entities can be modeled as agents, passive resources can be modeled as artifacts. Moreover, a specific interaction model Section 4.1 rooted on use and perception allows to ease the interactions among the conceptually different parts of a program.

Despite the set of good points just listed, when moving from the distributed artificial intelligence context to general-purpose computer programming and software development – the reference context for this thesis work – several main weaknesses and limitations arise, in particular w.r.t. the second macro-research objective identified in Chapter 1. This was well expected: JaCa and JaCaMo, as well as the set of agent-oriented programming languages and technologies that have been proposed in the state-of-the-art (Chapter 3), are not fully adequate when general-purpose software development is of concern. Indeed, agent-oriented programming is lacking a big majority of aspects and features that have been – and still are – foundational for mainstream programming paradigms such as the object-oriented one, and which are also the key for software engineering processes at their support.

A main example, as in the case of the actor paradigm, is the lack of an explicit notion of type for what concerns the main programming abstractions introduced: agents, artifacts[1] and the overall organization. This causes two main drawbacks. First, the support for (static) error detection in current state-of-the-art agent-oriented technologies is quite limited, much weaker indeed compared to what we have e.g. in (statically) typed object-oriented programming languages. This is also the case in JaCa and JaCaMo where error checking controls are limited to simply syntactical aspects. As a result, MAS developers are forced to deal *at run-time* with a set of programming errors that should be detected statically instead, before running the MAS program. This increases the cost of errors detection from both a temporal and economic point of view, and causes complicated – and possibly long – debugging sessions for detecting errors at run-time—e.g., an error that occurs only after several complex computations and long interaction dynamics. For sake of concreteness in Figure 6.1 is reported a snippet of a JaCa program with some main examples of programming errors that is not possible detect statically, due to the lack of an explicit notion of type. As already mentioned, analogous considerations also apply for JaCaMo, in which in addition is not possible to statically check all the aspects related to the interactions of agents with the organization—e.g., adoption of unknown roles, declaration of plans for the achievement of wrong organizational goals, etc.

[1] Being CArtAgO based on Java, it is possible to define types for artifacts relying on the object-oriented layer, however this approach is not expressive enough to characterize at a proper level of abstraction the features of environment programming—i.e, operations, observable properties, etc.

```
// agent ag0
iterations("zero").

!do_job.

+!do_job
  <-  ...
      /* error: wrong type
         String vs int (N+1) */
      -+iterations(N+1);
      ...
      /* error: the correct
         belief is iterations(N) */
      ?num_iterations(N).

/* error: the message sent by ag1
   is msg_bel */
+msgbel
  <-  printf("Message received").

// agent ag1
!send_msg.

+!send_msg
  <- .send(ag0, tell, msg_bel).
```

```
public class DummyArtifact extends Artifact{
  void init(){
    defineObsProperty("propA",10);
    defineObsProperty("propB","value");
  }
  @OPERATION void actionA(int a, int b){..}
  @OPERATION void actionB(String s){..}
  @OPERATION void actionC(int a){..}
}
```

```
// agent ag2
!do_job.

+!do_job
  <-  /* error: unknown goal
         floor_cleaned */
      .send(ag3,achieve,floor_cleaned);
      ...
      /* typo: the right name is do_job */
      !dojob.

//agent ag3
+!car_cleaned
  <-  ...
```

```
// agent a4
iterations("zero").

+envPerceptA(ValueA)
  <-  ...
      actionA(10,20);
      ?iterations(I)
      /* error: wrong type provided for I */
      actionA(10,I); f
      actionB(I);
      /* error: wrong type provided
         int vs String */
      actionB(10);
      /* error: unknown observable property */
      ?envPerceptC(ValueC);
      /* error: unknown operation */
      actionD.

/* error: unknown observable property */
+envPerceptC(Value)
  <- ...
```

Figure 6.1: Source code of four ***Jason*** agents working in a cooperative manner sharing the usage of a DummyArtifact. The picture shows a set of typical programming errors in JaCa concerning: *(i)* belief-related errors (*left*), *(ii)* goal-related errors (*top right*), and *(iii)* agent-environment interaction errors (*bottom right*).

As second main drawback, it deprives developers of a fundamental conceptual tool when programming a system. Without typing support, it is not possible to model generalization/specialization relationships among concepts and abstractions, eventually specializing existing ones through the definition of proper sub-types, and finally making it possible to fully exploit the principle of substitutability [WZ88] for supporting a safe extension and reuse in programming.

Besides typing, in agent-oriented programming in general – and this is also the case in JaCa and JaCaMo – are also lacking strong and well established mechanisms to support inheritance and polymorphism. The former is the key to provide reuse and modularization in programs. Providing good support for modularity is a main issue already recognized in the agent literature [BPL06, DMS08, Hin08], where constructs such as capabilities have been proposed to this

end. ***Jason*** still lacks of a construct to properly modularize and structure the set of plans defining an agent's behavior—a recent proposal is described here [ML10] However, even if several proposals do exist, up to now there is not a well established solution to the problem.

Similar considerations also apply to polymorphism. Previously we mentioned that both JaCa and JaCaMo provide a first basic support for polymorphism by giving developers the opportunity to write different plans that can be applied on the basis of the actual context conditions. However, without the support for typing this can be considered only a very low level mechanism to realize polymorphism – moreover only from the agent side – which is not related to an explicit (possibly formal) characterization of the entity that provides it.

Part III

The simpAL Project

7

The simpAL Programming Language and Ecosystem

This chapter represents the core of this thesis work. Here are presented both the simpAL [RS11b] agent-oriented programming language and the ecosystem at its support, composed by an Eclipse-based Integrated Development Environment (IDE) and a distributed runtime infrastructure. The chapter is structured as follows:

- Section 7.1 provides an overview of simpAL, by presenting the vision and the background metaphor on top of which the programming language is rooted and the specific computational models adopted for two of the main first-class abstractions of the language, agents and artifacts.
- Section 7.2 describes the simpAL programming model. A concrete example is used to guide the reader through the description of the programming of the main first-class abstractions available in the language.
- Section 7.3 puts the focus on some of the most relevant features of simpAL, namely: *(i)* the seamless integration of autonomous and event-driven behaviors (Section 7.3.1); *(ii)* the support for typing (Section 7.3.2) and polymorphism (Section 7.3.3); and *(iii)* the simpAL distributed runtime infrastructure (Section 7.3.4).
- Finally, the chapter is concluded by presenting the development of concrete case studies (Section 7.4), the simpAL IDE (Section 7.5), and a discussion about the current performance (Section 7.6) and limitations of the language (Section 7.7).

7.1 simpAL Overview

simpAL is a strongly-typed programming language, extending a pure object-oriented layer – based on Java – with an orthogonal high-level abstraction layer introducing agent-oriented abstractions. *Pure* means that object-oriented programming (OOP) here is meant to be exploited solely to define abstract data types, so data structures and related algorithmic computations

without any specific OS-related mechanisms for dealing with I/O, concurrency, etc. These aspects are meant to be fully captured by the new abstraction layer.

7.1.1 Background Metaphor: Human-Inspiered Computing

Quoting Lieberman [Lie06], *"the history of object-oriented programming can be interpreted as a continuing quest to capture the notion of abstraction—to create computational artifacts that represent the essential nature of a situation, and to ignore irrelevant details"*. Following this perspective, agent-oriented programming as supported by simpAL can be framed as an evolution of object-oriented and actor-oriented programming, by adding a *human-inspired* abstraction layer integrating elements from the A&A conceptual model and the BDI agent architecture, which have been described in detail respectively in Chapter 3 and Chapter 4.

In particular human organizations are taken as a natural high-level metaphor to define the structure and behavior of (complex) programs, where articulated, concurrent and coordinated activities take place, distributed in time and space. The complexity of work calls for some division of labor among the members of the organization, who do their jobs eventually interacting with other members and/or by exploiting the organization environment (e.g., resources, tools). Analogously, a program in simpAL is conceived like a human organization where the members are called *agents*. They are the main abstraction used to model those parts of the program that are in charge of performing autonomously some *tasks* eventually interacting with other agents and with the *environment* where they are situated (see Figure 7.1). *Autonomously* means in this case that, given a task to do, they pro-actively decide what are the best actions to perform and when to do them, promptly reacting to relevant events from their environment, fully encapsulating the control of their behavior.

Compared to existing abstractions used to model active components – actors in particular – the agent abstraction is characterized by the introduction of a further layer of first-class concepts (besides pure message passing) aiming at easing the design and programming of complex behaviors, integrating pro-activity and reactivity. Such first-class concepts include: *(i) tasks*, to represent the description of the jobs that agents have to do; *(ii) plans*, encapsulating the procedural knowledge and recipes about how to accomplish the tasks; *(iii) actions*, which are the moves that agents can do, depending on the environment in which they are situated, in order to do their tasks; *(iv) percepts*, which are the events that agents asynchronously observe from the environment to which they may need to react, in order to do their jobs.

As in the human case, in simpAL agents can interact by means of direct communication based on message passing, in an actor-like style, revisited taking into account the expertise coming from all the work that have been done in this field by the agent community (Section 3.4.1). However, differently from actors, which are based on the *reactivity-principle* (Section 2.1), agents do something not because they receive a message, but because they have some tasks to do; indeed, it is often the case that in order to accomplish such tasks, they may need to react to messages sent by other agents or by changes observed into their environment, as prescribed by the plans. In that sense, recalling what stated in Section 3.1, we can say that they are based on

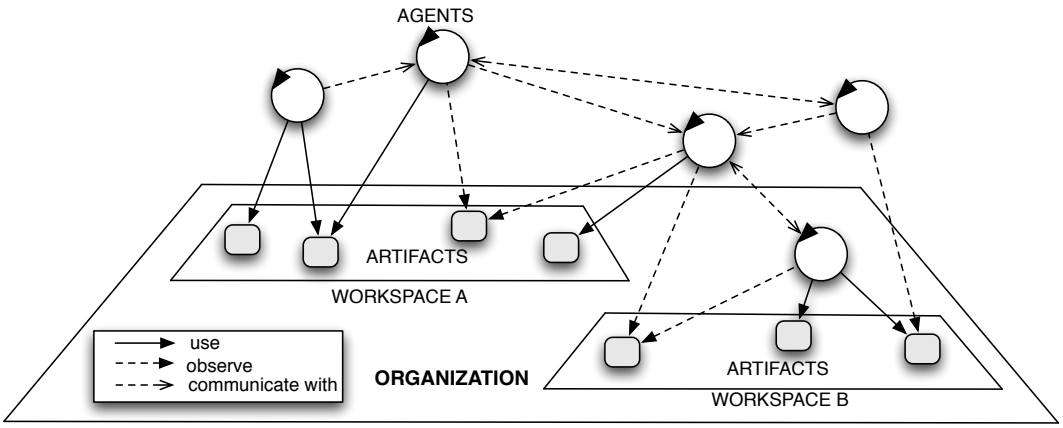


Figure 7.1: An abstract representation of a simpAL program.

a *pro-activity* principle, autonomously choosing and executing actions in order to fulfill some tasks.

The environment in human organizations play a key role, as the context mediating and then supporting members' individual and cooperative tasks, through the use of shared tools and resources [Nar96]. So, if on the one side agents are the abstraction meant to model active, task-oriented, autonomous behaviors, artifacts on the other side are meant to be effective for modeling non-autonomous components encapsulating and modularizing functionalities that can be suitably exploited by agents. Examples of artifacts are bounded buffers, a clock, or rather a database, an external web service. Indeed, in some cases an entity of a program may be modeled as an agent or as an artifact: it is a designer choice, depending on the fact that it is more effective or appropriate to conceive it as an autonomous task-oriented entity or as a function-oriented passive one. Analogously to artifacts in human organizations, artifacts in simpAL can play an essential role to support indirect forms of communication and coordination. Main examples are blackboards or tuple spaces, which in the case of simpAL are naturally modeled as artifacts. Like artifacts in the human case, artifacts in simpAL can be dynamically created (by agents) and disposed, and eventually can be designed to be composed, so as to create complex artifacts by connecting simpler ones. The use of the environment as a first-class abstraction is a further main difference with respect to the actor model, where actors – being the only abstraction – are finally used to model both real autonomous entities but also those entities that are not meant to exhibit any specific autonomous behavior.

So, the picture so far includes two main computational abstractions, agents and artifacts. Agents can *talk* with other agents, and *use* and *observe* artifacts available (or created) in their organizational environment. Then, a further concept is needed in order to explicitly define the overall structure and topology of the program, which can be distributed. To this end, in simpAL we introduce the notion of *workspace*. The overall set of agents and artifacts of an organization

may be partitioned into a set of workspaces as logical containers, possibly running on different nodes of the Internet network. Actually, while being located into a specific workspace, an agent can work concurrently and transparently also with artifacts of other workspaces belonging to the same organization.

It is worth remarking that the notion of organization used in simpAL *is not meant to be as rich as the one that appears in MAS organization modeling* (Section 3.4.2): here the main objective for introducing such a notion is to have a way to define rigorously the overall context and structure of programs. This will be useful from a concrete programming perspective, in particular to check errors related to the implementation of the overall program structure at compile time. These aspects will be described in detail in Section 7.2.3 and Section 7.3.2.

Finally, as already mentioned at the beginning of the section, pure objects – as defined in modern OOP – are used to define the data model of programs. That is, agents and artifacts are meant to be used as coarse grain abstractions to define the shape of the organization (i.e., of the program), in particular of the control part of it (decentralized, distributed). This layer is fairly independent from the paradigm and language adopted to represent data structures and purely transformational computation. In the case of simpAL to this end we adopted an object-oriented programming language, in particular a subset of Java, that is the pure object-oriented part of the language, excluding constructs and mechanisms introduced for concurrency, I/O management, etc. Objects are then the basic data structures used inside agents, artifacts and related communications and interactions.

Before getting into the simpAL language where these concepts are provided as first class constructs, in the remainder of the section we provide some details about the specific computational model adopted for defining the structure and behavior of agents and artifacts.

7.1.2 The Agent Model and Control Architecture

A key aspect in simpAL is the agent control architecture, which allows for integrating both an active, task-driven and reactive, event-driven behavior (this aspect is discussed with full details in Section 7.3.1) and then the *reasoning cycle* (or *agent control loop*) that conceptually defines such an integrated behavior. It is inspired by the BDI reasoning cycle (Section 3.2.1), and can be framed here as an extension of the basic event loop found in actors Section 2.1.

Conceptually, a simpAL agent is a computational entity executing continuously a loop which involves three distinct stages executed in sequence: *(i)* a *sense* stage, in which the agent fetches percepts (inputs) coming both from the environment and other agents, updating its internal state; *(ii)* a *plan* stage in which, given the current state and the set of current tasks that the agent is actually pursuing, the set of actions to do is selected, and finally *(iii)* an *act* stage in which the selected actions are executed. From a conceptual point of view, an agent is never blocked: it is continuously looping on these stages, eventually without choosing any action to perform if there are no active tasks or there is nothing to do in a specific moment in the tasks it is pursuing. Figure 7.2 shows an abstract representation of the simpAL agent architecture along with an indication of the dynamics of the execution cycle.

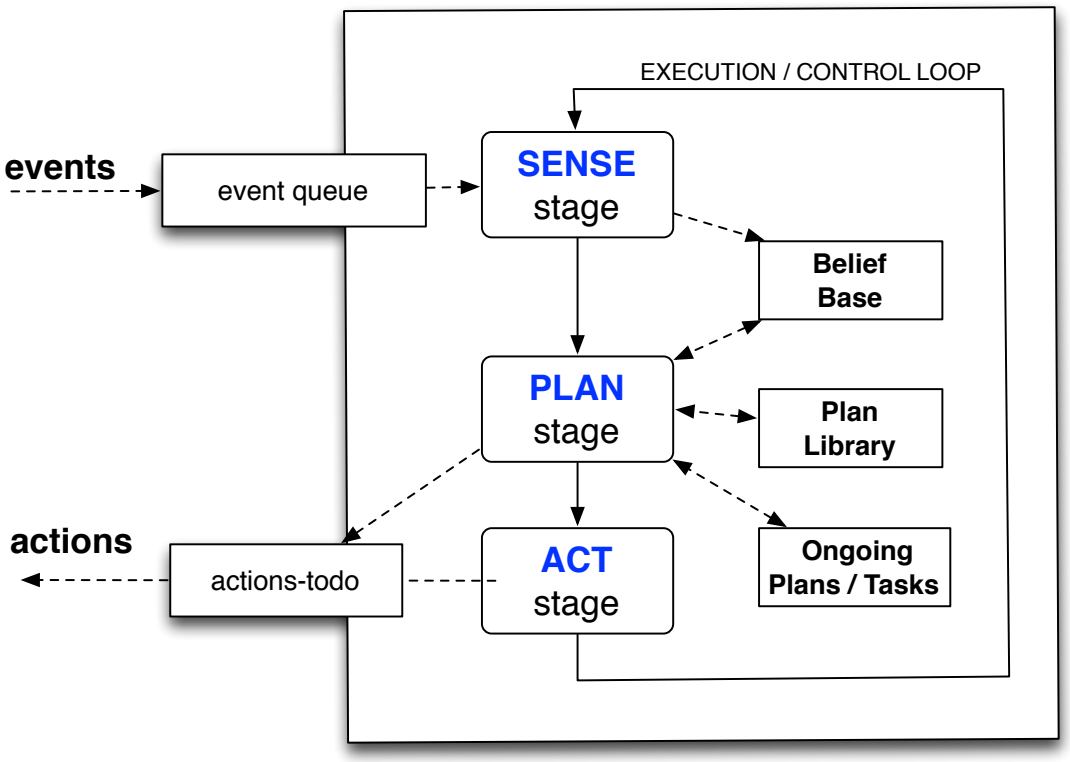


Figure 7.2: An abstract representation of a simpAL agent's control architecture.

The agent architecture is composed by:

- A *belief base* — which is the long term private memory of the agent, storing information about agent's private state, about the observable state of the artifacts the agent is using and about information communicated by other agents.
- A *plan library* — storing the current set of plans available for doing the agent's tasks.
- A *tasks-todo* list — containing the current list of tasks that need to be done.
- An *ongoing-tasks* list — each including the runtime structure related to the plans instantiated and in execution to accomplish the tasks.
- A *suspended-tasks* list — including those tasks which have been intentionally suspended.
- An *external event queue* — where inputs from the environment / other agents are enqueued.
- An *actions-todo* list — including a list of actions to execute.

Algorithm 2 simpAL Agent Control Loop

```
1:  while true do
2:                                                          ▷ SENSE stage
3:      if external events queue not empty then
4:          ev ← PICKEXTEVENT()
5:          UPDATEAGENTSTATE(ev)
6:      end if
7:                                                          ▷ PLAN stage
8:      if new tasks todo then
9:          for each new task to-do task do
10:             plan ← SELECTPLAN(task, belBase, planLib)
11:             CREATENEWINTENTION(plan, task)
12:         end for
13:     end if
14:     actList ← []
15:     for each ongoing intention i do
16:         if TASKFULFILLED(i) then
17:             DROPINTENTION(i)
18:         end if
19:         actList ← actList + SELECTACTIONS(i, belBase)
20:     end for
21:                                                         ▷ ACT stage
22:     for each action act in actList do
23:         EXECUTE(act)
24:     end for
25: end while
```

In the sense stage, the state of the agent is updated with what has been perceived from the outside, fetching *one* event – if available – from the external event queue. Such a state includes agent *beliefs* – i.e., the informational part of the state, composed by private state variables, possibly keeping track of the observable state of the artifacts the agent is using – ongoing tasks and tasks to do. The event can concern either some change in the observable state of artifacts, or a new message sent by another agent, or the notification of the completion with success or failure of an action executed by the agent on the environment.

In the plan stage, if there are new tasks to do, then for each one a *plan* is selected from the agent *plan library* to handle the task and a new *intention* is instantiated—i.e., a new activity committed to the fulfillment of the task, keeping track of the plan in execution. Plans as programming abstraction will be described in the next section: they are module of *procedural knowledge* [RG+95], composed by a set of action rules that specify *what* to do and *when* to do it. Then, for each ongoing intention, all the actions that can be executed – according to the plans and current beliefs of the agent – are collected. Intentions that achieved their task are dropped.

Finally, in the act stage, all the collected actions are executed. Internal actions – i.e., actions

accessing/modifying the internal state of the agent – are executed atomically in this stage, in one cycle. External actions instead – i.e., actions that correspond to the execution of an operation provided by some artifact of the environment – are just started/triggered. Their completion will be notified later on by proper action completion events, which may be perceived asynchronously in the sense stage of future cycles.

Two main differences compared to the actor event loop are important here. First, an agent can execute a cycle even if *there are no external events to process*. This happens when the agent has one or more intentions about tasks to be executed – that have been previously assigned – and following the related plans some actions must be selected pro-actively. For instance: a plan stating that some action *a* must be continuously selected and executed, like a simple non terminating process. This is consistent with the idea that an agent follows the *pro-activity principle*. So, conceptually, an agent doing some task(s) is *never blocked*—always cycling until the task(s) have been achieved or failed. At the same time, an agent is reactive and event-driven: an event perceived in the sense stage can result in updating some agent beliefs and this could trigger the selection of some actions in the plan stage.

Second, intentions – i.e., plans in execution – *are not meant to be fully executed and completed in one cycle*: typically their execution require multiple cycles, each one selecting zero or one or multiple actions to be executed (depending on the plan). By doing an analogy between methods in the actor case and plans, this means that in the agent case the *macro-step semantics* (Section 2.1) is relaxed, or it has finer granularity, which is at the level of the actions composing a plan.

These features together allow to tackle the integration of the autonomous and reactive behaviors directly at the foundation level—however possibly raising performance issues, that will be discussed in Section 7.6.

7.1.3 A Computational Model for Artifacts

Artifacts have a simpler architecture w.r.t. agents' one, more similar to monitors as introduced in concurrent programming. The artifact model is inspired by our previous work on the CArtAgO technology, properly extended by introducing aspects and features – e.g., typing, polymorphism, etc. (see Section 7.3.2) – that are fundamental in the context of general purpose software development. Figure 7.3 shows a pictorial representation of an artifact in simpAL. As usual, similarly to artifacts used by humans, artifacts provide a *usage interface* which is exploited by agents to use and observe them. Such interface includes *(i)* a set of *operations*, that correspond to the set of actions available to agents for using artifacts; and *(ii) observable properties*, as variable-like information items storing those properties of an artifact which may be perceived and exploited by the agents using the artifact. Interactions among agents and artifacts are regulated by means of the action and perception model introduced in Section 4.1.

Artifacts are meant to be observed and used concurrently by multiple agents, automatically enforcing all the constraints that are necessary for avoiding interferences. To that purpose, operation execution in artifacts is *transactional*, in the sense that they are executed in a mutually

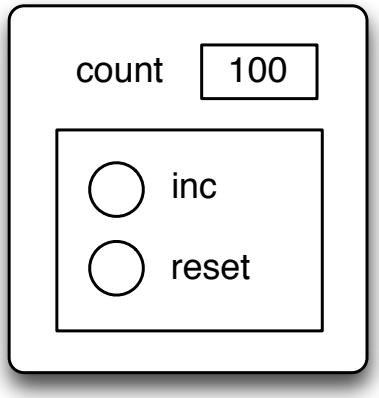


Figure 7.3: Abstract representation of a Counter artifact, with in evidence its usage interface: the operations (`inc` and `reset`) and one observable property (`count`).

exclusive way and the changes to the observable state of the artifact (properties) are done atomically. Changes are perceived by agents observing the artifact only when an operation completes (with success). The execution of an operation can fail: this causes the failure of the action on the agent side, while on the artifact side the observable state is rolled back to the value before executing the operation.

Then, it is often useful to design artifacts with long-term operations, that may eventually need to overlap in time. A main example is given by artifacts providing coordination functionalities, which typically provide operations (actions) whose execution must overlap in time in order to create the required synchronization. To that end, operations can be explicitly suspended waiting some conditions, allowing then other operations to be executed.

So artifacts strongly resemble monitors, with however some essential differences that are related to the agent-oriented model. In particular: *(i)* they have observable properties representing an observable state whose changes can be perceived as events by agents using them; *(ii)* the execution of operations triggered by agents (that are actions, from the agent viewpoint) is asynchronous with respect to agent execution: agents are notified of action completion or failure in terms of asynchronous events; *(iii)* on the agent side, by executing an operation over an artifact which is already busy with the execution of an operation, or an operation which suspends its execution, the agent cycle is not blocked but goes on regularly.

7.2 The simpAL Programming Language

After sketching the main features of the agent and artifact models, in this section we introduce the key elements of the simpAL programming language using a concrete example (Figure 7.4). We intentionally decided to use the same example rooted on the producer-consumer architecture

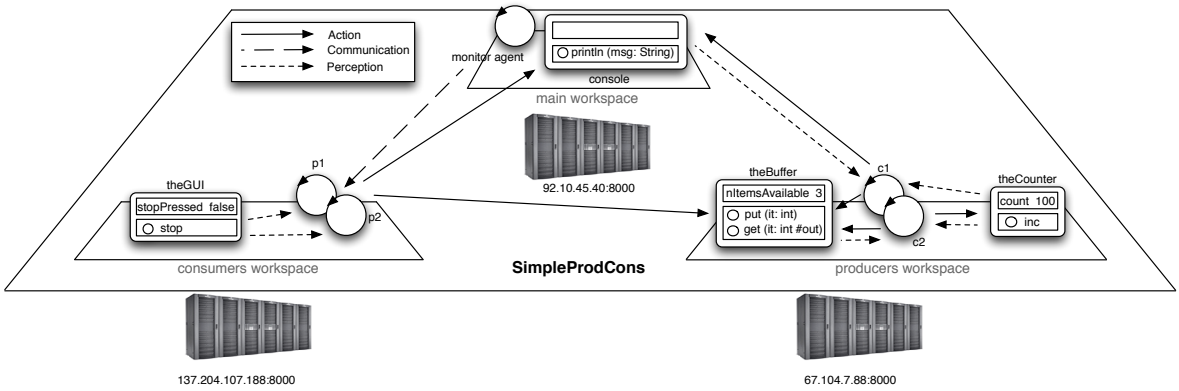


Figure 7.4: An abstract view of the producers-consumers example highlighting the agents and artifacts involved.

exploited in Section 4.2. However, in order to better point out and highlight the main features of simpAL, we introduced small modifications to the original example. The overall scenario, unchanged from the previous instance, is: *(i)* a set of producer agents (p1 and p2) have the task of continuously producing some items that must be consumed by a bunch of consumer agents (c1 and c2); *(ii)* consumer agents must stop their activities as soon as the total number of items processed is greater than a certain value; *(iii)* both consumer and producer agents must stop as soon as the user presses the `stop` button available in a GUI (theGUI). Differently from the previous example, here we have that:

- A monitor agent – observing the overall activities – may communicate directly to the producer agents that more items need to be produced.
- Instead of using a single artifact, multiple ones are exploited to support agent work and coordination: *(i)* a bounded buffer artifact (theBuffer), with the obvious functionality; *(ii)* a counter (theCounter), used by consumers to keep track of the overall number of items processed; and *(iii)* a GUI used by producers to observe user inputs (theGUI).
- The overall application must be physically distributed among three different network nodes, as depicted in Figure 7.4.

In the following we proceed bottom-up, first introducing the programming of agents (Section 7.2.1) and artifacts (Section 7.2.2) as basic components of a simpAL program, and then presenting the programming of the organization, defining the overall structure of the system (Section 7.2.3).

7.2.1 Programming the Agents

Following the basic principle of separation between interface and implementation, in simpAL the agent programming model is characterized on the one side by the notion of *role* and on the other side by the notion of *agent script*. The former is used to explicitly define an agent *type*, by grouping together the *types* of *tasks* that all the agents that declare to play a certain role are capable to do. The latter instead is the basic programming construct used to define the implementation of concrete *plans* useful to accomplish the tasks related to a specific role.

Referring to our example, Figure 7.6 shows the definition of Producer and Consumer roles, while Figure 7.7 and Figure 7.8 show the definition of the agent scripts SimpleProducer and SimpleConsumer implementing such roles.

Defining Agent Types: Roles and Tasks

From a software engineering perspective, a type defines a contract about what one can expect by some computational entity. In the case of objects, this concerns its interface, i.e. what methods can be invoked (and with which parameters) or – in a more abstract view – what messages can be handled by the object [Kay96, Kay69]. Conceptually, messages are the core concept of objects: receiving a message is the reason why an object moves and computes something—i.e., the *reactivity principle*. This is also true for active objects and actors Section 2.1.

Differently from objects and actors, agents are not based on the reactivity principle, since they do something not necessarily only when receiving a message, but because they have one or multiple tasks to accomplish. In that sense, we can say that they are based on a *pro-activity principle*, autonomously choosing and executing actions in order to fulfill some tasks. It is quite intuitive then to define an agent's type as its contract w.r.t. the organizational environment where it is immersed. Following this idea we introduce: *(i)* the notion of *task type* to describe a type of job that has to be done, and *(ii)* the notion of *role* to explicitly define the type of an agent as the set of the possible types of *tasks* that any agent playing that role is able to do. So, a role is the specification of *what* agents playing the role are capable of (not how they do it). In Appendix A is reported the EBNF syntax of role and task definition in simpAL. A role is identified by a name and includes the definition of a set of task types. A task type defines a *contract* between the task *assigner* and the task *assignee*, which is defined in terms of:

- An `input-params` block, defining the information that must be specified when the task is assigned to the assignee agent.
- An `output-params` block, defining the information that must be specified when the task has been completed by the assignee agent.
- An `understands` block, containing the definition of messages that can be understood by the task assignee, *in the context of that particular task*.
- A `talks-about` block, containing the definition of messages that can be sent by the task assignee to the assigner, *in the context of that particular task*.

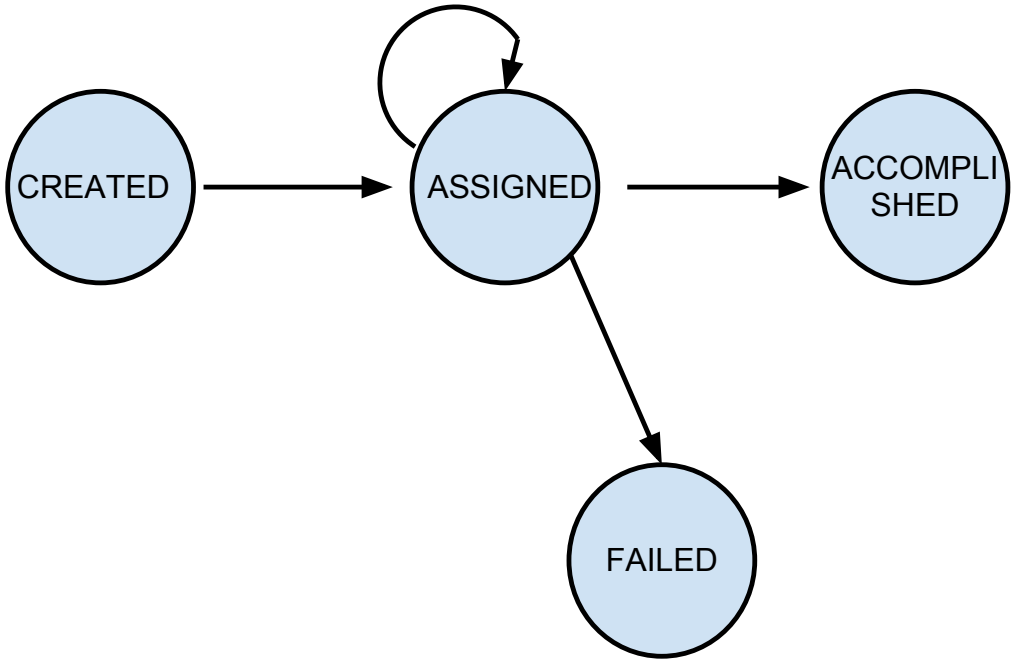


Figure 7.5: The task life cycle in simpAL, with in evidence the different execution stages.

Tasks have a specific life cycle (Figure 7.5), including the following stages: *created*, *assigned*, *accomplished* and *failed*. A task is first created by some agent (as an instance of some type of task). The task is then assigned to some other agent (it could be also self-assigned or assigned at boot by the programmer), that becomes the *assignee* of the task, while who assigned the task becomes the *assigner*. Finally, a task can be completed with success or can fail, on the basis of the outcome of the actions performed by the assignee for its achievement. Built-in predicates are available to check the tasks states: *(i)* `is-ongoing`, which returns true if the specified task is defined and it has been assigned but it is not completed; *(ii)* `is-done`, which returns true if the specified task has been completed; and *(iii)* `is-failed`, which returns true if the specified task is defined and its execution has failed.

In Figure 7.6, the roles Producer and Consumer have only one task type each, respectively Producing and Consuming. The former has three input parameters: *(i)* the maximum number of items to produce (`numItems`), *(ii)* the GUI that users can use to stop the production of new items (`gui`), and *(iii)* the bounded buffer used to store produced items (`buffer`). While doing the Producing task, Producer agents can be told about the value of the `newItemsToProduce` belief. The Consuming task type instead is characterized by only input parameters: *(i)* the maximum number of items to consume (`maxItemsToProcess`), *(ii)* the bounded buffer artifact used to retrieve items (`buffer`), and the artifact used to keep track of the items consumed so far (`counter`).

```
role Producer {

  task Producing {

    input-params {
      numItems: int;
      gui: GUI;
      buffer: Buffer;
    }

    output-params { }
    understands {
      newItemsToProduce: int;
    }
    talks-about { }
  }
}
```

```
role Consumer {

  task Consuming {

    input-params {
      maxItemsToProcess: int;
      buffer: Buffer;
      counter: Counter;
    }

    output-params { }
    understands { }
    talks-about { }
  }
}
```

Figure 7.6: (*left*) Definition of the Producer role. (*right*) Definition of the Consumer role.

So, roles define the type of agents, to be used to define the type of an agent's reference or identifier in beliefs and task parameters on the agent side, in observable properties and variables on the artifact side. This allows for doing a set of error checking controls at compile time, as will be better clarified in Section 7.3.2.

Defining Agent Structure and Behavior: Scripts and Plans

A script represents a module of agent behavior, containing both the definition of a set of *plans* useful to accomplish the tasks of the role declared to be implemented by the script, and a set of beliefs that can be accessed by all the plans declared into that script. By loading a script, an agent adds the declared beliefs into its belief base, and the script's plans to its plan library. The script SimpleProducer shown in Figure 7.7 has a global belief – `item` (line 4), used to keep track of the current item produced by the agent – and a plan, for handling the Producing task.

Plan definition includes two basic parts: the definition of the plan's attributes and body. The set of plan's attributes are:

- `task:` — to specify the task type for which the plan can be used.
- `name:` — to unambiguously identify the plan inside the script (optional, if omitted a default value is generated).
- `context:` — to specify the precondition over the belief base that makes the plan applicable (optional, if omitted the context is `true` by default).

The plan body contains the specification of the procedural knowledge that agents can use in order to accomplish the task associated to a plan. Such knowledge can be specified in terms of *action rules*, that are ECA-like rules, each specifying an *action* to do, along with the event and condition specifying *when* the action must be done. An action rule block – which constitutes

the body of a plan, denoted by { . . . } – is a set of action rules, possibly including the definition of local beliefs, i.e. beliefs whose scope is the block, as a kind of short-term memory. Action rule blocks can be nested, making it possible to structure the behavior inside a plan—this point will be discussed extensively in next sub-sections.

Recalling the agent execution cycle described in Section 7.1.2, in the plan stage, when an agent perceives that a new task has to be done, it selects from its plan library an applicable plan given the type of the task and the context condition. Once a plan has been found, the agent instantiates a new intention, representing the plan in execution. The intention is kept until the task is accomplished or failed.

Action blocks can have attributes to declaratively specify aspects related to actions and their execution inside the block. Among these, the main ones are:

- `#using:` — to specify the list of the artifact identifiers used inside the block (e.g., Figure 7.7 lines 9 and 21, and Figure 7.8 lines 8, 12 and 24). An artifact inside a block can be used/observed only if explicitly declared. At runtime, when entering a block where an artifact is used, the artifact's observable properties are automatically perceived, in a continuous fashion, and their value is stored in corresponding beliefs in the belief base—updated in the sense stage of the agent execution cycle.

- `#completed-when:` — to specify the condition for which the action rule block execution can be considered completed (e.g., Figure 7.7 line 8). As soon as the condition holds, the block is terminated with success.

- `#atomic` — to specify that the action rule block must be executed as a single action, without being interrupted or interleaved with blocks of other plans in execution (when the agent is executing multiple tasks at a time).

Other attributes will be described in next sub-sections.

Actions Rules: Events, Conditions, Actions

The action rule model has been specifically devised to ease the definition of behavioral blocks which may need to integrate and mix the execution of some workflow of actions, along with the reactions to some events or conditions over the state of the agent (see Section 7.3.1 for full details). An example of such behavior is given by the plan for the Producing task in the SimpleProducer script (Figure 7.7, lines 7-37). It accounts for repeatedly generating and inserting integer items in the buffer artifact by executing the `put` action – which is part of the Buffer interface, as will be shown in Section 7.2.2 – until the number of items inserted in the buffer is greater than the maximum specified. While doing this, the agent must also react to events coming from the environment and other agents, in particular: when the stop button is pressed in the GUI the agent must stop and successfully terminate the plan; every time the agent is informed by other agents about new items to be produced, the number of items to produce must be updated.

```
agent-script SimpleProducer implements Producer in ProdConsOrgModel {

  /* global beliefs */
  item: int = 0

  /* plans */
  plan-for Producing {
    #completed-when: is-done jobDone || is-done stopNotified
    #using:  console@main, this-task.gui

    /* plan's beliefs */
    noMoreItemsToProduce: boolean = false
    nItemsProduced: int = 0
    nItemsToProduce: int = numItems

    println(msg: "num items to produce: "+nItemsToProduce) on console@main;
    {
      /* active/autonomous part */
      #to-be-rep-until: nItemsProduced >= nItemsToProduce || stopPressed
      #using: this-task.buffer

      item = item + 1;
      put(item: item) on this-task.buffer;
      nItemsProduced = nItemsProduced + 1
    };
    println(msg: "job done") #act: jobDone

    /* reactive part */
    when changed stopPressed in this-task.gui=> {
      println(msg:"stopped.")
    } #act: stopNotified

    every-time told this-task.newItemsToProduce => {
      println(msg: "new items to produce: "+this-task.newItemsToProduce);
      nItemsToProduce = nItemsToProduce + this-task.newItemsToProduce
    }
  }
}
```

Figure 7.7: Implementation of the SimpleProducer script used in the producer-consumer example.

In the remainder of this sub-section we show how this behavior can be achieved, by describing in detail the action rule model. The EBNF syntax that defines an action rule in simpAL is reported in Appendix A. Practically, in the most general case an action rule is of the kind:

Ev : *Cond* => *Act* #act: *Tag*

Ev is an event template, specifying what kinds of event can trigger the rule; *Cond* is a boolean expression, specifying the condition over the agent's belief base that must hold for a triggered rule to be applicable; finally *Act* is the action to be executed if the rule is triggered and is applicable, and *Tag* is a label which may be optionally specified to identify the action executed by the rule. So, informally, an action rule states that the specified action *Act* labeled as *Tag* can be executed every time the event *Ev* occurs and the specified condition *Cond* holds.

Some syntactic sugar is provided then to ease the implementation of frequently used patterns of actions. One example is the *sequence* of actions. A sequence or a *chain* of actions is defined by a list of actions a_i, where: *(i)* the action a_k can be executed only when the completion of action a_{k-1} is perceived, and *(ii)* any action a_i must be executed only once. This can be specified as a simple list of actions (with no event or condition specified) using '`;`' as separator. The end of the sequence is then determined by the first rule without a semicolon. If the sequence is composed by a single action, then the action can be selected and executed immediately, but only once.

An example of sequence of actions is shown in the plan for the Producing task, in Figure 7.7. The plan first prints a message on a console (line 16), then a nested block (lines 17-25) is executed and when the block has been completed then a last message is printed (line 26). The block (lines 17-25) contains a sequence (lines 22-24), in which first a new item is generated, then the item is inserted in the buffer by means of the `put` action and only when the `put` succeeds the number of items produced is incremented. A block can contain also multiple independent sequences, that are executed conceptually in parallel.

Some syntactic sugar is provided also for coding reactions. Rules that need to be triggered *only once* can be coded by specifying the `when` keyword at the beginning of the rule, before the event/condition (that can be both specified or just one of them):

```
when Ev :  Cond => Act #act:  Tag
when Ev => Act #act:  Tag
when Cond => Act #act:  Tag
```

when rules are translated in flat ones by simply adding a further condition in `cond` at compile time, which make the rule applicable only if the action `Tag` is still todo. An example is shown in the plan for the Producing task of the SimpleProducer script (lines 29-31). When a change of the observable property `stopPressed` in the GUI artifact is perceived, then the agent must react, printing a message.

Besides `when` rules, rules which need to be triggered *every-time* some event/condition hold can be implemented by using the keyword `every-time`:

```
every-time Ev :  Cond => Act #act:  Tag
every-time Ev => Act #act:  Tag
every-time Cond => Act #act:  Tag
```

An example is provided again in the plan for the Producing task (lines 33-36). Every-time a message about the new threshold is told, the agent must react and update the total number of items to produce.

Events — Events specified in ECA rules concern percepts related to either the environment, or messages sent by other agents, or action execution. In any case, all events are uniformly

modeled as changes to some belief belonging to the agent belief base, given the fact that observable properties, messages sent and action state variables are all represented as beliefs. The syntax for specifying events related to a change of an observable property is `changed ObsProp`, for instance (Figure 7.7, line 29):

```
1 when changed stopPressed in this-task.gui => {...}
```

In this case the rule is selected as soon as the belief about the current value of the observable property `stopPressed` of the GUI artifact has changed. It is worth remarking that, thanks to *smart* controls performed by the simpAL compiler, when referring to a general task element (inputs/outputs/messages) the prefix `this-task.` can be omitted if there is no ambiguity w.r.t. other beliefs declared in the script (e.g., in this case if the agent does not declare any belief named `newItemsToProduce`).

The syntax provided to specify the update of a belief about an information told by another agent in the context of a task is `told this-task.what`, for instance (e.g., Figure 7.7, line 33):

```
1 every-time told this-task.newItemsToProduce => {...}
```

In this case the action is selected every time the belief `newItemsToProduce` representing a message sent by the task assigner in the context of the Producing task has been updated. Given the simpAL task model, only the assignee can react to the reception of messages defined in the `understands` block of a task type.

Other events are related to changes of beliefs used to keep track of the execution state of those actions that have been explicitly labeled. In this case the event is syntactically represented by one of the two following form:

```
1 when is-done actLabel => ...
2 when is-failed actLabel => ...
```

where `actLabel` is the label that has been previously used to tag an action rule.

Internal Actions — Actions can be either *internal* – i.e., affecting only the agent internal state – or *external*—i.e., actions affecting the environment or other agents. Internal actions are predefined, and include:

- Basic actions to work with beliefs (i.e., belief assignment, to update the value of a belief (e.g., Figure 7.7 lines 12-14 and 24)).
- Actions to create a new task (`new-task`), and manage the current set of the agent's ongoing tasks and plans included in the plan library: *(i)* `assign-task myTaskBel` which succeeds as soon as the task is successfully self-assigned; *(ii)* `do-task myTaskBel` (Figure 7.8, line 17) which instead waits for the completion of the self-assigned task; *(iii)* `drop-task`, `suspend-tasks` and `resume-task`, with the obvious functionalities; *(iv)* `drop-all-tasks` which drops all the agent's ongoing tasks,

and related intentions, but the current one; and *(v)* `forget-all-plans`, which acts on the current intention's rule stack, removing all the action rule blocks but the top level one.

- Actions to work with and manipulate local (Java) objects, to instantiate local objects and to invoke methods (adopting a Java-like syntax).
- Actions to realize classical control structures – e.g., if, for, etc. – which have as arguments action rule blocks.
- Actions to support the nesting of action rule block ({ . . . } and => { . . . }, the difference among the two will be described in a following sub-section). An example is given by the block of rules described in Figure 7.7, lines 17-25. The utility of nesting blocks is to modularize the behavior of a plan in terms of parts, each specifying both what actions and reactions the agent can do and specific execution attributes.

External Actions — External actions are of two main types. The first one is given by operations provided of some artifact. The complete syntax for external action execution is `OpName(Params) on ArtifactId`. For instance (Figure 7.7 line 23):

```
1 put(item: item) on this-task.buffer
```

triggers the execution of the `put` operation (action) on the artifact whose identifier is stored in the `this-task.buffer` belief. The artifact identifier can be directly a literal, e.g. `console@main` including the workspace name (see Appendix A). Recalling the agent execution cycle, it is worth noting here that external actions are executed in the act stage without blocking the control loop: the completion with success or failure of an action is eventually perceived by an agent as an event, which will be enqueued in the event queue in the future.

The repertoire of actions that an agent can do is dynamic and depends on the set of artifacts available in the environment. Besides this dynamic set, three external actions are always available to the agents by means of a special syntax (Appendix A). These actions are the `new-artifact`, `dispose-artifact` and `new-agent`, which are provided by the WorkspaceArtifact, an artifact available by default in each workspace (see Section 7.2.3). The former allows to create a new artifact. It requires in input the artifact's template name, initial parameters and a belief in which the artifact reference will be stored. To destroy an existing artifact, the `dispose-artifact` operation is provided instead, which requires in input a belief storing the id of the artifact to be disposed of. Usage examples of these two default external actions follows:

```
1 newArt: CounterUI;
2 new-artifact Counter (startValue:0) ref: newArt;
3 ...
4 dispose-artifact newArt;
```

Finally, the last action (`new-agent`) can be used to create a new agent. The action requires in input the script name, a belief in which will be stored the reference of the created agent and, if needed, the agent initial task. An example of usage follows:

```
1 newAg: Producer;
2 new-agent SimpleProducer init-task: new-task Producing (...) ref: newAg;
```

Thanks to specific compiler controls, the explicit specification of an external action's target artifact (i.e., `on artifact`) can be omitted in agent code (as it happens in some points of the SimpleProducer script shown in Figure 7.7 (e.g., line 30 and 34)) when there is no ambiguity w.r.t. the artifacts that are currently used by the agent (i.e., there is only one artifact providing the specified operation among the ones currently used).

The second type of external action is given by *communicative actions*, to support direct communication among agents. In simpAL, direct communication is based on asynchronous message passing plus a predefined set of communication actions with a built-in semantics, similar to speech acts. These communication actions can be roughly separated in two main classes:

- The ones concerning the assignment and management of tasks execution of an external assignee agent (`assign-task`, `do-task`, `drop-task`, `suspend-task`).
- The ones concerning pure exchange of information (`tell`).

The actions of the first group manage task assignment and execution when the task's assignee is an external agent. From a syntactical point of view, they are equals to the internal actions that can be used in a script for managing agent's ongoing tasks—i.e., those tasks that have been assigned to the agent. However, when an explicit assignee is specified, the intended semantics of these actions is different. In this case these are communicative actions meant to be used by the assigner, to assign (as before `assign-task` succeeds as soon as the task is assigned, `do-task` instead waits for task completion), suspend (`suspend-task`) or cause the abortion of the execution (`drop-task`) of a task assigned to an external agent. Referring to our example, the monitor agent (whose source code is not shown for sake of simplicity) assigns the execution of two instances of the Producing task to producers p1 and p2 as follows:

```
1 ...
2 tForP1: Producing = new-task Producing (numItems:20000, gui:theGUI, buffer:theBuffer)
3 tForP2: Producing = new-task Producing (numItems:20000, gui:theGUI, buffer:theBuffer)
4 assign-task tForP1 to: p1AgRef;
5 assign-task tForP2 to: p2AgRef;
6 ...
```

The task's input parameters are specified in a keyword-based fashion, in any order.

Exchanges of information are always contextualized to tasks. That is: if there are no tasks in execution, no messages (but those for assigning tasks) can circulate; if an agent is not doing any task, it does not make sense for it to receive any message (but those for assigning tasks). So, an agent A can send an information to another agent B only referring to a task T, without explicitly referring to B. To make a concrete example, let's consider the case of a task instance T, which has been assigned to B by agent A. Two cases hold:

- A can send an information in the context of the task T to B by issuing a `tell` action, specifying one of the messages declared into the `understands` block of task T type definition.
- Being B the assignee of the task T, it can send an information to the assigner (A) by issuing a `tell` action, specifying one of the messages declared into the `talks-about` block of task T type definition.

Going back to our example, being the monitor agent the assigner of the `tForP1` and `tForP2` Producing tasks, it can tell producers a message about new items to produce as follows:

```
...
tell tForP1.newItemsToProduce = 100
tell tForP2.newItemsToProduce = 100
...
```

This goes toward a more data-driven approach to message-passing, where communication accounts for asserting news about task parameters (i.e., the task data). We believe that this approach will have some nice impacts also on how complex communication protocols will be specified in simpAL.

Action Rule Block Management: Nesting, Interruption, Completion and Repetition

As already mentioned, internal actions include also the instantiation of a new action rule block, to support nested blocks. At runtime, for each intention – i.e., plan in execution – a stack of action rule blocks is managed. At the beginning, the stack contains only the body of the plan. Then, as soon as an internal action instantiating a new sub-action rule block is executed, the block is pushed on the top of the stack—i.e., it is nested over a current one, which becomes its parent. That internal action is then considered completed as soon as the action rule block is completed, and then the block is removed from the stack.

Besides being useful to structure the set of rules, block nesting makes it possible to realize an interrupt behavior. For instance:

```
counter : Counter;
{
  #using: counter
  nInterrupts : int = 0
  ...
  println (msg : " this ");
  println (msg : " can be");
  println (msg : " interrupted ")

  when changed count in counter => {
    println (msg : " interruption !");
    nInterrupts ++
  }
}
```

In this example, the sequence of printing actions can be interrupted in any point as soon as the agent perceives that the observable property `count` has changed. When (if) this occurs, the

```
1  agent-script SimpleConsumer implements Consumer in ProdConsOrgModel {
2
3    /* global beliefs */
4    consumed: int
5
6    /* plans */
7    plan-for Consuming  {
8      #using: console@main
9
10     consumed = 0;
11     {
12       #using: this-task.counter, this-task.buffer, theGUI@producers
13       #to-be-rep-until:  (count >= this-task.maxItemsToProcess) || stopPressed
14
15       item: int;
16       get(item: item);
17       do-task new-task ProcessItem(item: item);
18       inc()
19     };
20     println(msg: "consumer done - num items processed: "+consumed)
21   }
22
23   plan-for ProcessItem {
24     #using: console@main
25     consumed = consumed + 1;
26     if (consumed % 100 == 0){
27       println(msg: "processed "+item)
28     }
29   }
30
31   /* private task definition */
32   task ProcessItem {
33     input-params {
34       item: int;
35     }
36   }
37 }
```

Figure 7.8: Implementation of the SimpleConsumer script used in the producer-consumer example.

block at lines 10-13 is pushed on the stack. Blocks pushed by reactions – like in this case – are tagged by default as *hard-blocks*. This means that when selecting actions in the plan stage, if an hard-block is at the top of the stack, only the rules of this block are considered, and the rules of other blocks below in the stack are ignored. In other words, hard-blocks cannot be interrupted by rules not belonging to the block.

Blocks pushed on the stack by pure actions (rules without the event/condition) are by default tagged as *soft-blocks*. In that case, when selecting actions in the plan stage, if a soft-block is at the top of the stack, also the other blocks in the stack are considered. For instance:

```
1  counter : Counter;
2  {
3    #using:counter
4    nInterripts : int = 0
5    condition : boolean
```

```
6     ...
7     println (msg : " this ");
8     if (condition){
9       println (msg : " can be");
10      println (msg : " interrupted ")
11    }
12
13    when changed count in counter => {
14      println (msg : " interruption !");
15      nInterrupts ++
16    }
17  }
```

Here `if` is a predefined simpAL internal action, pushing on the stack the block specified in the *"then"* arm if the condition holds. In this case the block specified in lines 8-11 is soft and can be interrupted. In the case of need, the attribute hard/soft can be explicitly specified using the predefined `#hard-block` and `#soft-block` attributes.

As already mentioned, the completion of a block is defined by the `#completed-when:` attribute. If this attribute is not explicitly specified by the programmer, then some different cases are considered by default, depending on the content of the block. If the block contains only one or multiple sequences of actions – i.e., no reactions – then the condition implicitly defined in `#completed-when:` is the completion with success of the last action of every sequence. In other words, the block completes when all the sequences of actions complete. Instead, if the block contains at least one reaction, i.e. an action rule with the event/condition specified, then the default value for `#completed-when:` is false. In this case the block is meant to be never completed—this is useful, for instance, in maintenance tasks.

Finally, pro-active tasks typically account for repeatedly executing some set of actions. In simpAL, besides `while` and `for` internal actions, this can be expressed declaratively, by means of some attributes of an action rule block: `#to-be-repeated` and `#to-be-rep-until: Cond`. The former says that once completed, the action rule block should be re-instantiated on the stack. The latter is a variant in which the block is re-instantiated until the specified condition holds. In the example, this attribute is used both in the plan for Producing and Consuming tasks. In the former case (line 19), it is used to specify that the block should be repeated until all the items have been produced or a stop command on the GUI has been issued. In the latter case (line 13), the block is repeated until the count observable property of the counter shared by the consumers achieved the desired value or, again, the GUI issued a stop.

Structuring Complex Plans

As a structuring mechanism, complex plans can be structured by breaking a task in internal (private, not visible by other agents) sub-tasks, defined in the context of the script.

An example is reported in the SimpleConsumer script shown in Figure 7.8. The plan for the Consuming task accounts for repeatedly consuming items from the buffer and process them, until the total number of items processed by the overall set of consumers is greater than a certain threshold (stored in the `maxItemsToProcess` belief, which is an input parameter of

the Consuming task) or the stop button has been pressed. A counter artifact is then used to keep track of such total number, incremented by every consumer after an item has been processed. So, the processing of the item is represented by a private task type (ProcessItem) defined in the script (lines 32-36), as well as a plan for handling it (lines 23-29). The sub-task is assigned at line 17, after having successfully retrieved an item from the buffer. Being this a simple sequence of actions, every action is executed implicitly when the completion event related to the previous one is received. So, the counter is incremented (line 18) only after the sub-task assigned by the `do-task` action is completed.

7.2.2 Programming the Environment

The programming model of artifacts is definitely simpler than the agents' one, more similar to the model used for classic passive entities, such as monitors or objects. Artifacts are simple modules encapsulating the implementation and execution of sets of operations as actions that the artifact makes it available to agents, and a set of observable properties that agents using the artifact may perceive. Besides observable properties, an artifact can contain also hidden (not observable) state variables, useful for implementing artifact functionalities.

```
1  interface Buffer {
2
3    obs-prop nAvailItems: int;
4
5    operation put (item: int);
6    operation get (?item: int);
7  }
```

```
1  interface Counter {
2
3    obs-prop count: int;
4
5    operation inc();
6    operation reset();
7  }
```

Figure 7.9: Implementation of the artifact usage interfaces used in the in the producer-consumer example: Buffer usage interface (*left*), and Counter usage interface (*right*).

Analogously to the agent case, also for artifact programming we separate the abstract description of the artifact functionalities from their concrete implementation, defining artifact structure and behavior. The former is specified in *usage interfaces*. Figure 7.9 shows the source code of both the Buffer and Counter usage interfaces. The definition of an usage interface includes the name of the interface, a set of observable properties and the declaration of a set of operations. Observable properties are similar to variables, characterized by a name, a value and a type. The parameters declared by operations are keyword based—for instance, `put` has a parameter called `item`. On the agent side when invoking the operation (i.e., executing an action), the parameters must be specified with the keyword, in any order. A parameter can be declared to be an *action feedback* – i.e., an output parameter which is computed by the operation and returned to the agent when the action (operation) has completed – by prefixing a `?` to its name (e.g., `?item` parameter in `get` operation). An operation can include multiple output parameters.

Usage interfaces define the type of artifacts, used for instance to type beliefs on the agent side keeping track of the artifacts to be used. This allows for doing a set of error checking controls at compile time, as will be better clarified in Section 7.3.2.

```
1  artifact SimpleBuffer implements Buffer {
2
3    /* hidden state variables */
4
5    int maxNumElems;
6    java.util.LinkedList<Integer> elems;
7
8    /* constructor */
9
10   init (maxElems: int) {
11     count = startValue;
12     nAvailItems = 0;
13     maxNumElems = maxElems;
14     elems = new java.util.LinkedList<Integer>();
15   }
16
17   /* operations implementation */
18
19   operation put (item: int) {
20     await nAvailItems < maxNumElems;
21     elems.add(item);
22     nAvailItems = nAvailableItems + 1;
23   }
24
25   operation get (?item: int) {
26     await nAvailItems > 0;
27     nAvailItems = nAvailableItems - 1;
28     item = elems.remove();
29   }
30 }
```

```
1  artifact SimpleCounter
2            implements Counter {
3
4    /* hidden state variables */
5
6    c0: int;
7
8    /* constructor */
9
10   init (startCount: int) {
11     count = startCount;
12     c0 = startCount;
13   }
14
15   /* operations */
16
17   operation inc() {
18     count = count + 1
19   }
20
21   operation reset(){
22     count = c0;
23   }
24 }
```

Figure 7.10: SimpleBuffer and SimpleCounter artifact templates implementing the usage interfaces shown in Figure 7.9.

The implementation of artifacts instead is defined in *artifact templates*. Figure 7.10 shows the source code of the SimpleBuffer and SimpleCounter artifact templates implementing the usage interfaces shown in Figure 7.9. Like classes in object-oriented programming, artifact templates are a blueprint for creating instances of artifacts. On the agent side, a predefined WorkspaceArtifact available by default in each workspace (see next sub-section) provides actions for artifact creation (`new-artifact`) and disposal (`dispose-artifact`). On the environment side, when the `new-artifact` operation is executed, the `init` operation (Figure 7.10 on the left at lines 10-15, and on the right at lines 10-13) of the artifact template to be created – i.e., the template's *constructor* – is invoked. Such operation is responsible of getting the initial parameters and setting up the initial artifact state.

The definition of an artifact template includes a name, the declaration of the implemented usage interface, the concrete implementation of operations and the definition of internal variables (non observable) that can be accessed by operations. In templates, the observable properties are not re-declared, being already declared in the usage interface.

Operation behavior is given by a simple sequence of statements, in pure imperative style, using classic control flow constructs, assignment operators, etc. As previously mentioned, Java is used as the language for defining data structures. So, objects as well as primitive values can be

used in expressions and as value of variables and observable properties, and method invocation appears among the statement of the language. For instance, in the SimpleBuffer implementation, a `LinkedList` Java class is used to keep track internally (`elems` variable) of the elements stored in the buffer.

Besides classic statements, specific primitives are introduced to control operation execution. For instance, the `await` statement – used in the `get` and `put` operations of the SimpleBuffer (Figure 7.10) – allows for suspending an operation until the specified condition is met (allowing then other operations to be executed). As in the case of monitors, only one operation can be in execution: so if multiple suspended operations can be resumed a certain time, only one is selected for being executed. This feature is useful in particular to implement coordination artifacts, i.e. artifacts explicitly designed to provide also coordinating/synchronizing functionalities to the agents sharing and concurrently using them.

Operations may complete with success or fails. Correspondingly, the agent who issued the operation will eventually receive an action completion event with success or an action failure event.

7.2.3 Programming the Organization

The global structure of a simpAL program and its initial configuration are specified by the notion of organization (recalling the human organization metaphor). Also for this aspect we separate the model part from the concrete implementation one. To this end, on the one side we have the notion of *organization model*, which is introduced for specifying the abstract structure of the overall program. On the other side, the notion of *concrete organization* allows to define a concrete application instance, referring to an existing *organization model*.

An *organization model* is identified by a name, and it is used to define the workspace-based logic structure of the organization, in which, for each workspace, it is possible to define, statically, the name (identifier) and the type of agents and artifacts that, for that workspace, will be automatically instantiated and initialized at each launch of the organization. By default, each workspace contains a predefined set of artifacts created at boot time:

- A console artifact, providing functionalities for printing on standard output.
- A WorkspaceArtifact artifact, providing functionalities for dynamically: *(i)* creating new artifacts and disposing existing ones (`new-artifact` and `dispose-artifact` operations), and *(ii)* spawning new agents (`new-agent`) operation.

Figure 7.11 shows the definition of the organization model used in our producer-consumer example. As requested, the topology of the application has been structured in three workspaces: `producers`, `consumers`, and `main`. The workspace `producers` hosts the producer agents (p1, p2) and the GUI artifact (theGUI). The workspace `consumers` hosts the consumer agents (c1, c2), the counter and the buffer artifacts (theCounter, theBuffer). Finally, the `main` workspace contains the monitor agent (monitor) and the console artifact that will be used for purely logging functionalities (not declared since is available by default).

```
org-model ProdConsOrgModel {

  workspace producers {
    p1: Producer
    p2: Producer
    theGUI: GUI
  }

  workspace consumers {
    c1: Consumer
    c2: Consumer
    theCounter: Counter
    theBuffer: Buffer
  }

  workspace main {
    monitor: Monitor
}}
```

Figure 7.11: Organization model for the producer-consumer example.

```
org SimpleProdCons implements ProdConsOrgModel {

  workspace producers {
    p1 =  new-agent SimpleProducer()
    p2 =  new-agent SimpleProducer()
    theGUI = new-artifact SimpleGUI (title: "Simple GUI")
  }

  workspace consumers {
    c1 =  new-agent SimpleConsumer()
            init-task: new-task Consuming (maxItemsToProcess: 20000, buffer: theBuffer,
                                           counter: theCounter)
    c2 =  new-agent SimpleConsumer()
            init-task: new-task Consuming (maxItemsToProcess: 20000, buffer: theBuffer,
                                           counter: theCounter)
    theCounter = new-artifact SimpleCounter(startValue: 0)
    theBuffer = new-artifact SimpleBuffer(maxElems: 10)
  }

  workspace main {
    monitor = new-agent SimpleMonitor() init-task: new-task Monitoring()
}}
```

Figure 7.12: The SimpleProdCons concrete organization implementing the organization model shown in Figure 7.11.

```
/* SimpleProdCons organization deployment file */
workspace-addresses {
  main = 92.10.45.40:8000
  producers = 67.104.7.88:8000
  consumers = 137.204.107.188:8000
}
```

Figure 7.13: Deployment configuration file for the concrete organization shown in Figure 7.12.

An organization can contain further agents/artifacts instances besides those statically declared in the organization model—since both agents and artifacts can be dynamically created by agents by means of specific actions (see above). The static case is useful anyway to specify the identifier of those components whose name and type must be known at the organization level, at *compile time*. In other words to define global symbols – i.e., agents and artifacts *literals* – that can be resolved and checked in scripts that *explicitly declared* to play a role inside an organization of this type (e.g., SimpleProducer and SimpleConsumer scripts which declare to play their role inside the ProdConsOrgModel). By doing so, the symbols and identifiers declared in the organization model can be referred as literals also in the script (e.g., `theGUI@producers` in Figure 7.8, line 12) and then checked at compile time.

Then, the definition of a concrete organization accounts for specifying the concrete instances of agents and artifacts declared in the organization model. For artifacts, the artifact template's name is provided, also possibly including the value of some parameters required by the artifact initialization operation. For agents, the initial script to be loaded must be specified, and, if needed, could be also specified the initial task to do, along with its initial parameters. Figure 7.12 shows an example of a SimpleProdCons organization for the case of our producer-consumer example, where the agents/artifacts declared into the ProdConsOrgModel organization model are properly initialized in order to characterize the specific SimpleProdCons organization instance.

So, in simpAL a physically distributed multi-agent system is programmed as an *organization* whose logical structure is defined in terms of workspaces distributed among different network nodes. In this perspective, the notion of workspace represents the key to conceive and model the logical structure of the MAS, by properly grouping, on the basis of both application requirements and decisions made at design time, agents and artifacts in the set of workspaces that define the organization. This grouping is meant to define only the logical structure of the application, abstracting from all the details concerning its deployment, which is instead managed by the simpAL distributed runtime infrastructure (Section 7.3.4), through the definition of run/deploy file configurations. Such files are used to specify the binding of the logical workspace-based structure of an organization, into the proper set of simpAL *nodes* targeted for hosting its execution. A simpAL node is a generic machine available on the network, on top of which the simpAL kernel has been properly installed and launched. An example of configuration file for the SimpleProdCons organization is reported in Figure 7.13. In this case we distribute the organization on top of three different simpAL nodes (i.e., `137.204.107.188:8000`, `92.10.45.40:8000` and `67.104.7.88:8000` (Figure 7.4)).

The model easily allows to swap one configuration file with another giving hence the opportunity to change, even radically, the whole deployment model of a distributed application—e.g., moving, for the same application, from a deployment configuration with all the workspaces hosted on localhost, to another one in which each workspace is hosted on a different and remote simpAL node. It is up to developers decide how to organize both the logical and physical topology of the application, by taking into account both available hardware resources and application requirements—e.g., an application in which a set of independent tasks are managed by

a group of agents and artifacts located in separated and dedicated workspaces can be deployed, to exploits concurrency at its best, either on a set of normal desktops machines or on a server with high computing capabilities.

Applications that do not require to be distributed over the network can simply avoid to specify a deployment configuration file. In this case the simpAL runtime will execute the MAS as a (local) standalone application.

7.3 Focus on Main Features

In this section we focus on some of the main features of simpAL, namely: *(i)* the seamless integration of both autonomous and event-driven behaviors (Section 7.3.1), thanks to the plan and action rule models; *(ii)* the support for typing (Section 7.3.2) and polymorphism (Section 7.3.3); and *(iii)* the simpAL distributed runtime infrastructure (Section 7.3.4).

7.3.1 Integrating Autonomous and Event-Driven Behaviors

The integration of autonomous and reactive behaviors is a relevant problem in the context of concurrent programming, related to the integration of thread-based and event-driven programming. Many interesting problems and applications call for developing software components capable of integrating a process-oriented autonomous behavior with an event-driven, reactive one. A simple example is a web crawler that has to search pro-actively information over the Internet and at the same time must be able to react to asynchronous inputs generated by the user through a GUI, either to interrupt the crawler or to dynamically refine its search.

Another example is given by cooperative distributed algorithms like Ricart-Agrawala (see Section 7.4.2 for an implementation in simpAL of the Ricart-Agrawala's algorithm) or Token-Ring [RA81, BA05], for distributed mutual exclusion and critical section. In these algorithms, the behavior of each distributed node/peer typically includes both an autonomous part (e.g., periodically entering in critical section (CS)), and a reactive part, which needs to receive and send messages aside to the first one to ensure the correct coordination among the nodes. The two parts then need to cooperate, since the behavior of the reactive part can depend on the state of the pro-active one (e.g., being in CS or not). The same situation can be found in producers-consumers architectures in which producers and consumers, while doing their job, need to be reactive to some kind of asynchronous events—e.g., producers should stop producing as soon as some condition occurs (we presented this example both in Section 4.2 and in Section 7.2).

In concurrent programming literature, the problem of integrating autonomous and reactive behaviors is strongly related to the one contrasting thread-based and event-based programming [vBCB03, Ous96], and to those works that look for unifying the approaches [HO07, CMM09]. Being based on a pure *reactivity principle* [BGL98, Kay69] (Section 2.1), actors – as objects as well – do not provide native means to effectively integrate also pro-activity, so actor-based solutions to this problem – as will be clarified in the next sub-section – suffer in

general of a weak abstraction and modularity.

Here, we discuss how in simpAL it is possible to integrate these two aspects at the conceptual and foundation level, going beyond the reactivity principle found in actors [RS12a]. Such a synergistic integration is made possible thanks to both the the agent control architecture (Section 7.1.2), and the specific plan and action rule models adopted in simpAL (Section 7.2.1).

Problem Statement

Both actors and objects are based on reactivity and the *reactivity principle*. As discussed in Section 2.1, the only way to activate an object or an actor is by sending to it a message. In [BGL98] this is opposed to the idea of a *process*, or a pure autonomous behavior, that starts processing as soon as it is created.

The integration of object with process (the concept of active object) raises the issue of whether reactivity will be preserved or shadowed by the autonomous behavior of the process. Two broad families are identified[BGL98]: *(i) reactive active objects* – these approaches adhere to the reactivity principle (such as actors); *(ii) autonomous active objects* – in these approaches the active entity may compute before being sent a message. Although the models are opposite they can easily simulate each other [BGL98]. On the one side, a reactive active object can have a method whose body contains an endless loop, turning it into an autonomous active object after receiving a corresponding message. On the other side, an autonomous active object whose activity is to keep accepting messages actually models a reactive active object.

However, this is not useful when dealing with problems that call for exploiting both of them in an *integrated* way. Abstracting from the details, all these problems have some kind of process doing pro-actively actions to accomplish some (possibly long-running) task that requires also to react to some asynchronous events from their environment. In the remainder we analyze the problem by considering an abstract example that captures some core issues. We will use actors as reference model and related state of the art programming technologies. Let's consider a task T which is supposed to be long-term, articulated in a sequence of three sub-tasks: Ta, Tb, Tc—for sake of simplicity we suppose initially that these sub-tasks are fully computational, without interactions. This constitutes the autonomous/pro-active part of the job. Then, the task requires to promptly react to a message `react` that could be sent in any moment while doing T, and upon receiving the message the actor must suspend the execution of T for printing in standard output the `react!` message. As soon as the message has been printed, the execution of T can be resumed.

Figure 7.14 on the left shows a first solution in ActorFoundry [KA], which we will consider as the reference technology for implementing pure actor solutions. Note that we could use any framework/language *strictly* implementing the actor model Section 2.1. The only peculiarity that we exploited of ActorFoundry is the `call` primitive, which realizes a request-reply message exchange pattern. The problem with this solution is that given the macro-step semantics adopted by the actor model, the actor can react to the the `react` message only after fully executing the body of the method `doTask` executed when receiving the corresponding message.

```
1  public class TestActor0 extends Actor {
2    private int c = 0;
3
4    @message
5    public void doTaskT() {
6      ta();
7      tb();
8      tc();
9    }
10
11   @message
12   public void react() {
13     call(stdout,
14              "println","react! "+c);
15   }
16
17   private void ta(){ c = c + 1; }
18   private void tb(){ c = c + 1; }
19   private void tc(){ c = c + 1; }
20 }
```

```
1  public class TestActor1 extends Actor {
2    private int c = 0;
3
4    @message
5    public void doTaskT() {
6      send(self(), "doingTa");
7    }
8
9    @message
10   public void doingTa() {
11     send(self(), "doingTb");
12     ta();
13   }
14
15   @message
16   public void doingTb() {
17     send(self(), "doingTc");
18     tb();
19   }
20
21   @message
22   public void doingTc() { tc(); }
23
24   @message
25   public void react()
26               throws RemoteCodeException {
27     call(stdout, "println","react! "+c);
28   }
29   ...
30 }
```

Figure 7.14: Different solutions to the abstract problem used to investigate the integration of active and event-driven behaviors in ActorFoundry. (*left*) A first solution that shadows reactivity, (*right*) a solution that does not.

So the message printed on standard output is always `react! 3`. In this case reactivity is shadowed.

In order to be reactive while doing the tasks, the autonomous behavior of the actor must necessarily be broken in sub-behaviors so as to allow the actor event loop to consider the receipt of the `react` message. An example is shown in Figure 7.14 on the right. The problem in this case is the fragmentation of the code in handlers, which does not necessarily corresponds to a good modularization from the point of view of the organization of the autonomous behavior. One is forced to artificially break the behavior so that the event loop can take the control and check the availability of messages possibly sent by other actors. Besides, self-sending messages is needed to proceed the computation, in the case that no messages are available, not to get stucked. This is clearly a programming trick, decreasing the level of abstraction used to describe the strategy identified at the design level.

No substantial improvements can be obtained if we consider actor approaches based on explicit acceptance of messages (following the classification discussed in [BGL98]), i.e. providing

```
test_actor() ->
  self() ! doTaskT, loop(0).

loop(C) ->
  receive
    doTaskT ->
      C1 = ta(C),
      self() ! doingTb,
      loop(C1);
    doingTb ->
      C1 = tb(C),
      self() ! doingTc,
      loop(C1);
    doingTc ->
      C1 = tc(C),
      loop(C1);
    react ->
      io:format("react! ~w~n", [C]),
      loop(C)
    end.

ta(C) -> C+1.
tb(C) -> C+1.
tc(C) -> C+1.
```

```
controller() ->
  ActA = spawn(test1, actorA, []),
  ActB = spawn(test1, actorB, []),
  ActC = spawn(test1, actorC, []),
  self() ! doTaskT,
  loop(0,ActA,ActB,ActC).

loop(C,ActA,ActB,ActC) ->
  receive
    doTaskT ->
      ActA ! {doTa, C, self()},
      loop(C,ActA,ActB,ActC);
    {doneTa, C1} ->
      ActB ! {doTb, C1, self()},
      loop(C1,ActA,ActB,ActC);
    {doneTb, C1} ->
      ActC ! {doTc, C1, self()},
      loop(C1,ActA,ActB,ActC);
    {doneTc, C1} ->
      loop(C1,ActA,ActB,ActC);
    react ->
      io:format("react! ~w~n", [C]),
      loop(C,ActA,ActB,ActC)
  end.

ta(C) -> C + 1.

actorA() ->
  receive
    {doTa, C, Controller} ->
      C1 = ta(C),
      Controller ! {doneTa, C1},
  end.
```

Figure 7.15: Different solutions to the abstract problem used to investigate the integration of active and event-driven behaviors in Erlang. (*left*) A first basic solution, (*right*) a solution based on a network of cooperative actors for improving reactivity.

a receive primitive to explicitly retrieve messages from the mailbox (and not encapsulating then the event-loop). Figure 7.15 on the left shows the solution in Erlang [Arm10], which is a main representative case. Here the fragmentation occurs by splitting the autonomous behavior into the arms of the receive primitive.

No improvements can be obtained neither if we consider the programming abstractions that have been proposed in literature upon the basic actor model – e.g., local synchronization constraints, synchronizers (Section 2.2). This because all such extensions are finally targeted to ease the management of messages, so improving the programming of the reactive part. The same applies for extensions introducing mechanisms to overcome handler/callback fragmentation—by means, for instance, of join continuations or promises (Section 2.2). These mechanisms are effective to improve the organization of the callbacks handling asynchronous events, avoiding obscure nesting.

A radically different approach to overcome the problem could be using a network of cooperating actors instead of one. In particular, a *controller* actor can be used to delegate the execution of the sub-tasks to other sub-actors. Figure 7.15 on the right shows an implementation of the example in Erlang. The controller is responsible to receive the `react` message and print the message, besides coordinating the sequential execution of the sub-tasks by interacting with the other sub-actors. In the case that the `react` message should have an effect also on the ongoing sub-tasks, then we would need to make also the sub-actors reactive to that message. For doing so, we need to decompose sub-tasks further. Sub-actors become controllers themselves, creating a tree of controllers where the leaves represent actors executing *atomic* tasks, that cannot be interrupted. Some specific message-based protocol must be introduced then among controllers in order to coordinate their execution, in particular to propagate the `react` message. So in this case we model pro-activity by a tree of purely reactive controllers, using in the leaves a set of actors playing the roles of atomic uninterruptible tasks or actions. While this pattern can be useful to deal with the problem from a pragmatic point of view starting from a pure actor model, it is apparent that it enforces the programmer to manage a further level of accidental complexity given by the set of controllers and their coordination.

Integrating Autonomous and Reactive Behaviors in simpAL

Figure 7.16 shows a simpAL implementation of the abstract example introduced in the previous sub-section. In the solution, the plan for fulling the task TaskT breaks it in three private sub-tasks Ta, Tb, Tc, to be fulfilled in sequence. The body of the plan is a sequence of `do-task` actions, which self-assign the sub-tasks to the agent and complete as soon as the task instance specified as parameter has been executed up to completion (Section 7.2.1). The script includes also the plans to manage the private sub-tasks, where a global belief `c` is incremented.

Then, as usual, reactivity is managed by introducing an action rule that specifies the reaction related to the specific event/condition of interest. The rule at lines 12-15 says that when the agent receives a message about reacting, it eventually interrupts the sequential course of action of the current intention and executes a new action rule block. In the reaction block the agent prints the `react` message on standard output using the usual `console` artifact available in the `main` workspace.

We argue that the solution showed in Figure 7.16 proofs in a concrete manner how it is possible to tackle the issue of integrating autonomous and reactive behaviors in simpAL programs, directly at the foundation level, without resorting to programming tricks. This is made possible on the one side by the specific plan model adopted in the language: through proper syntactical sugar, developers can easily program any arbitrary combination of sequence of actions – i.e., a block of autonomous/pro-active behavior – mixed with reactions—i.e., actions that must be taken if/when/every time some condition holds. On the other side, the agent control architecture is the one that makes it possible the concrete execution of such combinations of action rules, as it has been described in detail in Section 7.1.2.

```
1  agent-script TestScript implements RoleR {
2
3    c: int = 0;
4
5    plan-for TaskT {
6      #completed-when: is-done tc
7
8      do-task new-task Ta();
9      do-task new-task Tb();
10     do-task new-task Tc() #act: tc
11
12     when told this-task.react => {
13       #using: console@main
14       println(msg: "react! "+c);
15     }
16   }
17
18   plan-for Ta { c = c + 1 }
19   plan-for Tb { c = c + 1 }
20   plan-for Tc { c = c + 1 }
21
22   task Ta {}
23   task Tb {}
24   task Tc {}
25 }
```

```
1  role RoleR {
2    task TaskT {
3      input-params {
4        react: java.lang.String;
5      }
6    }
7  }
```

Figure 7.16: A solution to the abstract problem used to investigate the integration of active and event-driven behaviors insimpAL: (*left*) the implementation of the TestScript which integrates both autonomous and reactive behaviors as requested, and (*right*) definition of the RoleR implemented by the script.

Managing Tasks as First-class Entities — The capability of managing tasks in execution flexibly as first-class entity of the language is an important feature for programming more structured and complex behaviors, in particular for those cases in which the reactive part can influence the execution of the autonomous one. To this end, we consider here a simple extension of the previous example where the reaction is meant to produce a different future behavior depending on what the agent was doing. In particular: if the `react` message is received when doing the Ta sub-task, then the sub-task has to be interrupted and a task Td must be executed to complete TaskT. instead, if the `react` message is received when doing Tb, then the sub-task must be carried on until the end and after that, instead of doing a Tc sub-task, a new sub-task Te must be executed. All the other cases are not relevant for the agent.

The solution in simpAL is depicted in Figure 7.17 and it is a good example to show: *(i)* what kind of flexibility is possible by having tasks as first-class abstractions, and again *(ii)* the level of abstraction and modularity provided by the simpAL plan and action rule models. Differently from previous case, beliefs are used (`ta`, `tb`, `tc`, `te`) to track the tasks as soon as they are instantiated and assigned. The rule reacting to the event (lines 14-30) is triggered only if either the task `ta` or `tb` are ongoing. Then, by inspecting the state of the tasks, the future course of action for the plan is decided. In particular, if the task `ta` is ongoing (lines 23-26), then it is immediately dropped and a new task `tc` is assigned; otherwise, if the task `tb` is ongoing

```
1  agent-script TestScript implements RoleR {
2    c: int = 0;
3
4    plan-for TaskT {
5      #completed-when: is-done tc
6
7      ta: Ta = new-task Ta();
8      do-task ta;
9      tb: Tb = new-task Tb();
10     do-task tb;
11     tc: Tc = new-task Tc();
12     do-task tc
13
14     when told this-task.react : is-ongoing ta || is-ongoing tb => {
15       #using: console@main
16       #completed-when: is-done te || is-done newTc
17
18       te: Te; newTc: Tc;
19       forget-old-plans;
20       {
21         #atomic
22         println(msg: "react! "+c);
23         if (is-ongoing ta) {
24           drop-task ta;
25           newTc = new-task Tc();
26           assign-task newTc;
27         } else if (is-ongoing tb) {
28           te = new-task Te(prev: tb);
29           assign-task te;
30   }}}}
31
32   plan-for Te {
33     taskToWait: Tb = this-task.prev;
34     {
35       #completed-when: is-done te
36       when is-done taskToWait => using: console@main {
37         c = c * 100 ;
38         println(msg: "done te: "+c) ;
39       } #act: te
40     };
41   }
42
43   task Te {
44     input-params {
45       prev: Tb;
46   }}
47   ...
48 }
```

Figure 7.17: A more complex example of reactive behavior, that inspects ongoing tasks.

(line 27-29), a new sub-task `te` is instantiated, without dropping `tb` that can proceeds until completion. The plan handling the task type `Te` (lines 32-41) waits for the completion of the ongoing task `tb` – which has been passed as input parameter of the sub-task – before proceeding and doing its job, which accounts to update `c` and print a message on the console.

It is worth remarking the use of the `#atomic` attribute in the block at line 21 (introduced in Section 7.2.1) when the agent is reacting and deciding what to do, depending on the state of

the specific sub-task in execution. The attribute allows to specify that the agent must execute the actions defined in the block as a single action, without interleaving with other plans in execution that may interfere.

Event Handling Without Inversion of Control — A well-known problem in literature which is strongly related to the one discussed in this sub-section is the capability of doing event-oriented programming without the problems that typically are found when adopting callbacks and inversion of control [HO08, MTS05]. In simpAL there is no inversion of control, since events are managed by the agent control loop—which is a single control flow, from a conceptual point of view.

As an example, Figure 7.18 shows a possible implementation in simpAL of the *observer* pattern [GHJV95]. simpAL natively provides a support for publish/subscribe and observation-based kind of interactions. Observed objects can be directly modeled as artifacts with some specific observable properties. Observers can be modeled as agents, simply declaring to use those artifacts. The example shows the script of an agent observing a counter, reacting each time the observable property `count` changes, because of the execution of the `inc` operation by some (other) user agents, and the source code of the artifact implementing such Counter usage interface. Since this kind of interaction is part of the simpAL programming model, no specific code for managing observer registration / notification is necessary.

```
role Observer {
  task Observing {
    input-params {
     sharedCounter: CounterUI;
}}}
```

```
agent-script ObserverScript
                    implements Observer {
  plan-for Observing {
    #using: console@main, sharedCounter
    println(msg: "start observing...");
    every-time changed count => {
      println(msg: "new count perceived!")
    }
}}}
```

```
usage-interface CounterUI {

  obs-prop count: int;

  operation inc();
}
```

```
artifact Counter implements CounterUI {

  init() { count = 0; }

  operation inc () { count = count + 1; }
}
```

Figure 7.18: Role and script for observer agents (*left*). Usage interface and implementation of an observed counter artifact (*right*).

Every time the agent perceives a change of the observable property `count`, it prints a message on the console. The control flow executing this action is (conceptually) the agent control loop one. No race conditions and concurrency problems can occur even with multiple observers and users, thanks to the computation model of artifacts (in which the execution of operations is mutually exclusive) and of agents.

It is worth remarking that also in JaCa and JaCaMo events can be handled in a similar fashion, without introducing inversion of control.

Comparison with the support provided by BDI-based APLs — as introduced in Chapter 6 and showed in several concrete examples (e.g., see Section 4.2, Section 4.4.2, Section 4.6.2) the BDI control architecture makes it possible to realize an integration between autonomous (pro-active) and event-driven behaviors. However, this is just a first *basic* form of integration w.r.t. to the one provided by simpAL. Indeed, BDI-based state-of-the-art APLs do not provide adequate programming abstractions to seamlessly encapsulate inside a single computational block[1] any kind of possible behavior.

Taking ***Jason*** as a concrete example, it is not possible to mix, freely, a sequence of actions with reactions to particular events *inside the same plan*. Each event must be handled in a separated manner by writing a top-level plan of the kind `+my_event <- do_something`, *hence breaking the overall encapsulation of the plan construct*. To react to a particular event inside a plan, we are forced to define a new top-level reactive plan – i.e., *outside* the context of the original plan – that takes in charge the handling of the event of interest. Similar considerations also apply to other state-of-the-art APLs, in which events are always handled by means of top-level programming constructs. This could lead to several main drawbacks: *(i)* the main one is, as already mentioned, the breaking of the encapsulation of the basic programming construct provided by the language for coding agents' behavior, *(ii)* weak modularization, and finally *(iii)* poor code readability. All these issues in simpAL are solved thanks to the specific plan and action rule models adopted, which allow to freely combine reactive and event-driven behaviors with autonomous ones inside each action rule block.

To conclude the comparison, we consider now the capability of managing events without introducing inversion of control. As already mentioned, this can be done also in JaCa and JaCaMo thanks to the BDI agent control architecture and the specific action and perception model adopted (Section 4.1). However, the action rule model used in simpAL allows to introduce a set of important benefits when dealing with the management of asynchronous events:

- It is possible to do error checking on the rules that deal with events management. E.g., finding reactions to unknown events, type mismatch, etc.
- It is possible to make explicit, *at the block level*, the set of observable events of interest by specifying with the `#using:` attribute the set of artifacts to observe. This is a much more finer mechanism w.r.t the use of the explicit primitives `focus` and `stopFocus` available in JaCa and JaCaMo (see Section 4.2.3). Indeed, these primitives work at the global level, hence their use in the case of multiple active intentions could be troublesome— e.g., a plan can invoke the `stopFocus` action concerning an artifact that is still used by other parallel intentions.

[1] We intentionally used the term *computational block* in order to be general. Indeed, depending of the specific APL considered, the basic construct to define an agent behavior can be a plan, an action block, a rule, etc.

7.3.2 Typing Support

The introduction of an explicit notion of type for agents, artifacts and the overall programs (the organizations), give us the opportunity to: *(i)* conceptually characterize the different parts of an application, and *(ii)* perform several error checking controls at compile time [RS12b]. In Section 7.2 we introduced the definition and characterization of the types for the main first-class abstractions of the language, here instead we describe in detail the set of error checking controls available in simpAL, taking as a concrete example a SmartHome scenario.

A description of the example follows. An agent playing the role HomeAdmin manages the bedroom temperature by interacting with a Thermostat agent, which in turn uses a set of utility artifacts to do its job (see Figure 7.19 for an overview of the overall application).

```
org-model SmartHome{

  workspace mainRoom {
    majordomo: HomeAdmin
    userView: UserView
  }

  workspace bedRoom {
    thermostat: Thermostat
    conditioner: Conditioner
    thermometer: Thermometer
  }
}
```

```
org ACMESmartHome implements SmartHome{

  workspace mainRoom {
    majordomo = Majordomo()
                init-task: AdminHouse()
    userView = ACMEControlPanel()
  }

  workspace bedRoom {
    thermostat = ACMEThermostat()
    conditioner = ACMEConditioner()
    thermometer = ACMEThermometer()
  }
}
```

Figure 7.19: Organization model (*left*) and concrete organization (*right*) related to the SmartHome example.

Typing Agents

The concept of role defining the agent type allows us to do error checking on:

(a) The behavior of the agents implementing a role, checking that their implementation contained in scripts (the *how*) conforms to the role definition (the *what*).

(b) The behavior of the agents that aim at interacting with agents implementing a particular role, checking that: *(i)* they would request the accomplishment only of those tasks that are specified by the role, and *(ii)* they would send only those messages that the tasks' assignee can understand.

In simpAL, case *(a)* concerns performing two different checking controls when compiling agent scripts. The first control is responsible of validating the script's plans w.r.t. the task types defined in the role implemented by the script. The error checking rule states informally:

- For an agent script S, for each type of task T defined in the role R implemented by S, it must exist at least one plan P for T.

```
role Thermostat {

  task AchieveTemperature {
    input-params {
        targetTemp: double;
        threshold: double;
    }
  }

  task KeepTemperature {
    input-params {
      inputView: UserView;
    }
    understands {
      newThreshold: double;
    }
  }

  task DoSelfTest {
    talks-about {
      malfunctionDescr: MalfunctionInfo;
    }
  }
}
```

```
usage-interface Conditioner {

  obs-prop isHeating: boolean;
  obs-prop isCooling: boolean;

  operation startHeating(speed: double);
  operation startCooling(speed: double);
  operation stop();
}
```

```
interface Thermometer {
  obs-prop currentTemp: double;
}
```

```
usage-interface UserView {
  obs-prop desiredTemp: double;
  obs-prop threshold: double;
  obs-prop thermStatus:
     acme.ThermostatStatus;
}
```

Figure 7.20: Definition of the Thermostat agent role (*left*) and artifact interfaces Conditioner, UserView and Thermometer (*right*) related to the SmartHome example.

Given this rule, the ACMEThermostat script implementing the Thermostat role described in Figure 7.21 is correct, while instead a script like the following one:

```
agent-script UncompleteThermostatImpl implements Thermostat {
  plan-for AchieveTemperature { ... }
  plan-for DoSelfTest { ... }
}
```

would report an error message about missing a plan for a declared task, i.e. KeepTemperature. The second control concerns checking messages that an agent playing a certain role R can tell to its tasks assigners. This can be done by checking the set of messages included in the `talks-about` block of tasks definition. The checking rule in this case states:

- In a plan P related to a task type T, the messages sent by the *assignee* to the task *assigner* can be only the ones listed in T's `talks-about` block. In addition, the type of the messages sent must be compatible w.r.t. the message types defined in T.

Referring to our example, the only message that the thermostat agent can send to the majordomo is the message `malfunctionDescr` in the context of the task DoSelfTest, a task that can be assigned by the majordomo in order to check the correct functioning of the thermostat.

Case *(b)* concerns instead checking: *(i)* the assignment of tasks to agents playing a certain role R, and *(ii)* messages sent by a task assigner to the task assignee. As described in Section 7.2.1, task assignment (as well as task self-assignment) can be done using either the `do-task` or the `assign-task` action. In both cases, we can statically enforce that:

```
agent-script ACMEThermostat implements Thermostat in SmartHome {
  savedThreshold: double
  plan-for AchieveTemperature  {
    #using: console@mainRoom, thermometer@bedRoom, conditioner@bedRoom
    println(msg: "Achieving temperature "
      + this-task.targetTemp + " from " + currentTemp);
    savedThreshold = this-task.threshold;
    {
      #completed-when:
        java.lang.Math.abs(this-task.targetTemp - currentTemp) < savedThreshold

      every-time currentTemp > (this-task.targetTemp + savedThreshold)
        && !(isCooling in conditioner) =>  startCooling(speed: 1) on conditioner
      every-time currentTemp < (this-task.targetTemp - savedThreshold)
        && !(isHeating in conditioner) =>  startHeating(speed: 1) on conditioner
    };
    stop()
  }

  plan-for KeepTemperature {
    #using: console@mainRoom, thermometer, conditioner, userView@mainRoom
    quitPlan : boolean = false;
    {
      #completed-when: quitPlan

      achiveTempTask: AchieveTemperature =
        new-task AchieveTemperature(targetTemp: desiredTemp in userView,
                                    threshold: threshold in userView);
      assign-task achiveTempTask

      every-time changed desiredTemp => {
        drop-task achiveTempTask;
        achiveTempTask = new-task AchieveTemperature(targetTemp: desiredTemp,
                                                     threshold: threshold);
        assign-task achiveTempTask
      }
      every-time changed currentTemp : !is-doing-any AchieveTemperature => {
        assign-task new-task AchieveTemperature(targetTemp: desiredTemp,
                                                threshold: savedThreshold)
      }
      every-time changed thermStatus
       : thermStatus.equals(acme.ThermostatStatus.OFF) => {
        if (isCooling || isHeating){
          stop()
        };
        drop-task achiveTempTask;
        quitPlan = true
      }
      every-time told this-task.newThreshold => {
        #atomic
        savedThreshold = this-task.newThreshold
  }}}

  plan-for DoSelfTest {
    ...
    if (someCondition) {
      tell this-task.malfunctionDescr = new MalfunctionInfo( ... )
    }
    ...
}}
```

Figure 7.21: Source code of the ACMEThermostat script related to the SmartHome example.

- Given a belief *Id* of type *R*, storing the identifier of some agent playing the role *R*, then for any action `assign-task t to:Id` or `do-task t task-recipient:Id`, there must exist a task type *T* in *R* such that *t* is a value (instance) of *T*. In case of task self-assignment the belief *Id* storing the agent identifier is implicit (it refers to the current agent).

Then, given a script fragment with a belief `myThermostat: Thermostat`, we have the following list of the main errors that can be caught at compile time:

```
1  /* compilation ok */
2  assign-task AchieveTemperature(targetTemp:21, threshold:2) to: myThermostat
3
4  /*  error: no tasks matching CleanTheRoom in role Thermostat */
5  do-task CleanTheRoom() task-recipient: myThermostat
6
7  /*  error: no targetT param in AchieveTemperature */
8  /*  error: missing threshold param */
9  assign-task AchieveTemperature(targetT: 21) to: myThermostat
10
11 /*  error: wrong type for the param value targetTemp */
12 /*  error: missing threshold param */
13 do-task AchieveTemperature(targetTemp: "21") task-recipient: myThermostat
```

The definition of a task type *T* also includes the type of messages that the assigner can send to the task assignee. Given that, we can then check in agent scripts that the beliefs specified in the assigner's `tell` actions – those in which the task instance identifier is *not* `this-task.` – are among those listed in *T*'s `understands` block, and that the types of the beliefs are compatible. In the example, when doing the task KeepTemperature, the majordomo can tell to the thermostat agent (playing the Thermostat role) a message about the new threshold to adopt, which is represented by a belief `newThreshold` containing a value of type `double` (Figure 7.20 on the left, line 15). Examples of checks follow:

```
1  keepTempTask: KeepTemperature
2  /* compilation ok */
3  tell keepTempTask.newThreshold = 2
4
5  /* error: aMsg is not listed in KeepTemperature understands block */
6  tell keepTempTask.aMsg = "hello"
7
8  /* error: wrong type for the belief newThreshold
9   * told to an agent playing the role Thermostat */
10 tell keepTempTask.newThreshold = "2"
```

Finally, some other kinds of errors can be checked in scripts at compile time thanks to the explicit declaration of beliefs (and their types): finding errors in plans about beliefs that are not declared neither as beliefs at the script level, nor as local beliefs of plans, nor as parameters of the task; or about beliefs that are assigned with expressions of the wrong type.

Typing the Environment

The introduction of an explicit notion of type for artifacts allows us to define a way to address two main issues:

(a) On the agent side, checking errors about the actions (i.e. artifacts operations) and percepts (related to artifacts observable state).

(b) On the environment side, checking errors in artifact templates (i.e. the implementation), controlling that they conform to the implemented usage interfaces (i.e the type specification).

Referring to our example, Figure 7.20 on the right shows the definition of the artifacts types used by the thermostat agent, namely: *(i)* Conditioner, representing the interface of a conditioner device modeled as an artifact, used to heat or cool; *(ii)* Thermometer, used to be aware of the current temperature; and *(iii)* UserView, representing the interface of a GUI artifact used to interact with the human users, in particular to know what is the desired temperature. Figure 7.22 shows instead the skeleton of the definition of an artifact template implementing the Conditioner interface.

The case *(a)* concerns checking the action (rules) in plan bodies, so that for each action `OpName(Params) on Target`, specified in an action rule, meaning the execution of an operation *OpName* over an artifact identifier *Target* whose type is *I*:

- There must exist an operation defined in the interface *I* matching the operation requested.
- The action rule must appear in an action rule block (or in any of its parent block) where *Target* has been explicitly listed among the artifact used by the agent through the `#using:` attribute.

Examples of checks, given a fragment of a script with e.g. a belief `cond:Conditioner`:

```
1  /* compilation ok */
2  startCooling (speed: 1) on cond
3
4  /*  error: unknown operation switchOn */
5  switchOn () on cond
6
7  /*  error: unknown parameter time in startCooling operation */
8  startCooling (speed: 2 time: 10) on cond
9
10 /*  error: wrong type for the param value speed */
11 startCooling (speed: "fast") on cond
```

On the event/percept side, we can check beliefs representing artifact observable properties in the event template of rules and in any expression appearing either in the context or in action rule body, containing such beliefs. For what concerns event templates, given an action rule: `updated Prop in Target : Context => Action`, where the event concerns the update of the belief about an observable property *Prop* in the artifact of type *I* denoted by the belief *Target*, then the following checks apply:

- There must exist an observable property defined in *I* which matches *Prop*.

```
artifact ACMEConditioner implements Conditioner {
  nTimesUsed: int;

  init(){
    isCooling = false; isHeating = true; nTimesUsed = 0;
  }

  startCooling(speed: double){
    nTimesUsed++;
    isCooling = true;  isHeating = false;
    ...
  }

  startHeating(speed: double){...}

  stop(){
   isCooling = false; isHeating = false;
   ...
  }
}
```

Figure 7.22: Definition of the ACMEConditioner artifact template related to the SmartHome example.

- The action rule must appear in an action rule block (or in any of its parent block) where *Target* has been explicitly listed among the artifacts used by the agent through the `#using:` attribute.

Examples of checks follow, supposing to have a fragment of a script with beliefs `cond:Conditioner` and `therm:Thermometer` about a conditioner and thermometer artifact:

```
/* compilation ok */
changed currentTemp => println(msg: "the temperature has changed")
changed currentTemp : isHeating
       => println(msg: "the temperature has changed while heating...")
sum: double = currentTemp in therm + 1

/* error: unknown obs property isHeating  in Thermometer type */
changed isHeating in therm => ...

/* error: wrong type */
bak: boolean = currentTemp in therm
```

On the environment side (case *(b)*) the definition of the interface as a type allows for checking the conformance of artifact templates that declare to implement that interface, so that:

- For each operation signature *Op* declared in the interface *I* implemented by the template, the template must contain the implementation of the operation.
- For any observable property *Prop* that appears in expressions or assignments in operation implementation, then the declaration of the observable property must appear in the

interface implemented by the template and the corresponding type expression must be compatible.

Finally, the explicit declaration of observable properties (in interfaces) and (hidden) state variables in artifact templates – the latter can be declared also as local variables in operations – allow for checking errors in the implementation of operations about the use of unknown observable properties/variables or about the assignment of values with a wrong type. assignment of values with a wrong type.

Typing the Organization

The notion of organization model, defining the type for a simpAL organization, allows us to::

(a) Perform additional error checking controls in scripts *explicitly declared* in the context of an organization of a certain type.

(b) Control that an organization specification (i.e., the implementation) conforms to its type specification (i.e., organization model).

The case *(a)* allows to check, in those scripts sources declared inside an organizational context (e.g., Figure 7.21 line 1), that all the used literals refer to existing symbols defined in the related organization model.

On the organization side (case *(b)*), the definition of the organization model $OrgModel$ as a type allows for checking the conformance of a concrete organization instance Org that declares to implement that model, so that:

- Each workspace Wsp declared in $OrgModel$ must be defined also in Org.
- Each artifact literal $ArtLit$ of type I defined inside a workspace Wsp in $OrgModel$ must be correctly instantiated in the concrete organization Org. In particular such literal must be instantiated in Wsp, specifying an artifact template $ArtTempl$ implementing the usage interface I and, if needed, providing the initial parameters required by $ArtTempl$.
- Each agent literal $AgLit$ of type R defined inside a workspace Wsp in $OrgModel$ must be correctly instantiated in the concrete organization Org. In particular such literal must be instantiated in Wsp, specifying an agent script $AgScript$ implementing the type R and, if needed, providing an initial task to the agent.

It is worth remarking that in the definition of a concrete organization Org implementing an organization model $OrgModel$, additional workspaces and agent/artifact instances can be added to the ones initially declared in $OrgModel$. Examples of static checks that can be done are reported in Figure 7.23, supposing to have a fragment of an organization that declares to implement the SmartHome organization model defined in Figure 7.19.

```
1  org DummyHome implements SmartHome {
2    /* compilation ok: new workspace */
3    workspace newWsp {
4      otherConsole = Console()
5    }
6    workspace bedRoom {
7       /* error: missing instantiation of thermostat agent */
8      conditioner = ACMEConditioner() /* compilation ok */
9      /* error: wrong type. ACMEConditioner does
10        not implement the required Thermometer role */
11     thermometer = ACMEConditioner()
12   }
13 }
14 /* error: missing mainRoom workspace */
```

Figure 7.23: Example of a concrete organization instance with several compilation errors.

7.3.3 Polymorphism

simpAL supports various forms of polymorphism, at the agent, environment and organization level. This is possible thanks to the strong separation of concerns between the interface and the implementation that we operated in the definition of the main programming abstractions of the language.

On the agent side, we are able to support polymorphism at the role and at the script level. In the former case, given a role R, we can have multiple scripts $S1$, $S2$, etc. that provide different plans implementations to achieve R's tasks. However, despite implementation differences, all the scripts $S1$, $S2$, etc. must adhere to the contract defined by R, specified in terms of task types definition. So, in order to interact with agents implementing a certain role R, an agent only needs to know the expected behaviors of such agents as specified by R, abstracting from all the implementation details related to the actual scripts used by the target agents to play the role R. In the latter case instead, given a script S implementing a role R, in S it is possible to define different plans implementations to achieve a certain task type T defined in R, which can be applied on the basis of actual contextual conditions. Recalling the agent reasoning cycle, this selection is done during the plan stage when searching for an applicable plan for T, by evaluating the plans' context condition (`#context:` attribute, see Section 7.2.1). Like in BDI-based languages (Chapter 6), this gives developers the opportunity to define different implementations – and finally different behaviors – for the achievement of tasks, given the particular states in which an agent can be.

Polymorphism can be exploited also on the environment side. That is: different artifact templates $ArtT1$, $ArtT2$, etc. implementing the same artifact usage interface I, can provide different implementations and finally behaviors for the same operations P_i defined in I. So, on the agent side, different kind of artifacts implementing the same model are used in the same way, without the need of knowing the specific implementation of the artifact, yet possibly obtaining different specialized behaviors depending on the specific template.

Finally, on the organization side, polymorphism can be exploited thanks to the notion of or-

ganization model. An organization model *OrgModel* can be implemented by different concrete organizations *Org*1, *Org*2, etc., which can add new workspaces, and new agents and artifacts instances to the ones defined in *OrgModel*. On the one side, the definition of an organization model gives the opportunity to define a sort of *template* for a specific kind of applications—i.e., by characterizing the application's logic structure in terms of required workspaces and agents/artifacts that have to reside in such workspaces. On the other side, it can be used then as the blueprint to define concrete applications – i.e., organization instances – that can freely extend the template on the basis of specific needs.

7.3.4 Distributed Runtime Infrastructure

The deployment, execution and life-cycle management of programs written in simpAL are in charge of a distributed runtime infrastructure, developed in Java, which has been explicitly devised for managing all these issues transparently with respect to distribution [SR12]. The background idea that guided the development process of this infrastructure is: the execution and management of distributed MASs should be, from a developer perspective, as simple as the case of centralized, not distributed ones. To this end the notions of simpAL kernel and simpAL node have been introduced. The former is in charge of the concrete execution of simpAL programs or parts of them (in case of distributed execution). The latter instead is a generic network node in which the simpAL kernel is installed and executed – typically like a demon running in background, launched when the machine boots – used as the basic building block for providing a robust and flexible distributed runtime infrastructure for executing MASs—i.e., a simpAL node can be considered as a generic network node on top of which a user may want to host the execution of simpAL programs, or parts of them.

Once the kernel is up and running in all the interested network nodes, a user can start the execution of a simpAL application by launching it from a generic simpAL node that assumes the role of launch manager. Once the launch starts, the manager, using the information specified into the application deployment configuration file (Section 7.2.3), properly distributes the simpAL program, in its compiled version, among the other target nodes. It is worth remarking that the simpAL runtime infrastructure guarantees the consistency and the alignment of programs' sources either in the case of multiple local launches and in the case of distributed ones—i.e., only the up-to-date version of the compiled sources is loaded or distributed among the interested nodes. Since this phase does not involve any kind of logical dependency, it is done in a concurrent manner to speed up the boot process. Then, when the simpAL kernels installed in the interested nodes receive their part of the application, each one of them autonomously starts the booting by properly initializing the workspaces that need to be hosted in that node. During this process, the simpAL kernels communicate with each other by using a simple handshake protocol in order to keep track of the current status of the application and to guarantee a proper initialization of the MAS. Finally, only when all the workspaces have been deployed and all the static artifacts contained in them properly created, the agents are spawned and then the MAS can start its execution.

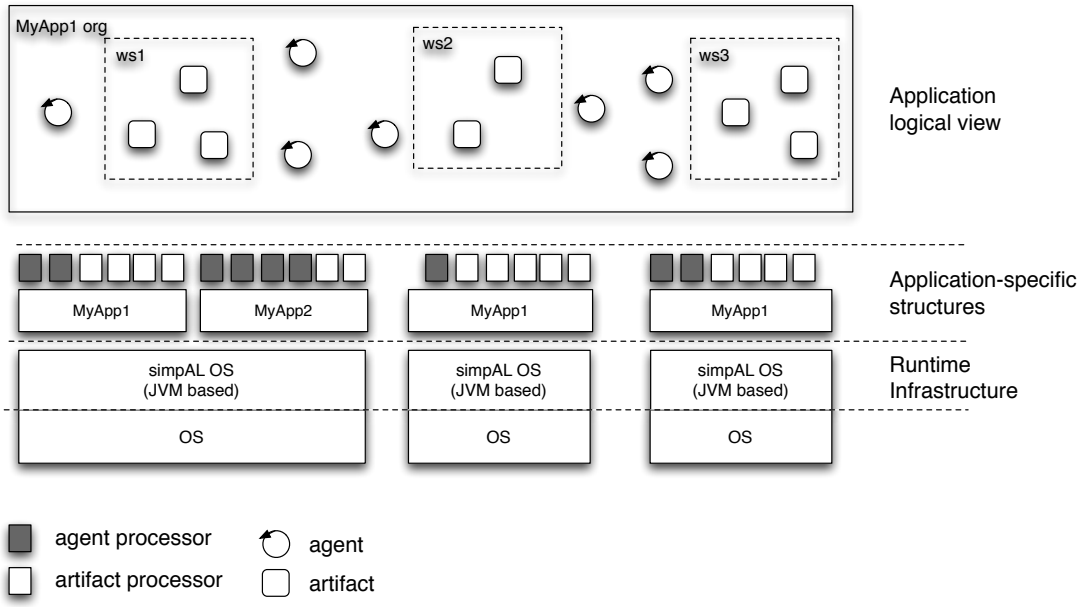


Figure 7.24: Abstract representation of a distributed simpAL program in execution on top of the simpAL distributed runtime infrastructure.

The termination of a running application can be triggered from any of the simpAL nodes in which it is hosted. Once triggered, the termination is managed, exploiting a proper shutdown protocol, in a coordinated manner by all the kernels involved in the shutdown process.

7.4 Concrete Case Studies

In this section we provide a concrete evaluation of the simpAL programming language through the implementation of some relevant programming examples. In particular, the examples have been purposely selected in order to stress, in real world case studies, the effectiveness of the support provided by the simpAL plan and action rule models for integrating autonomous and event-driven behaviors.

7.4.1 A Reactive File Searcher

This example concerns the realization of an active software component (or system) that searches and then prints in standard output the list of all the files of a certain directory whose size is greater than a threshold provided in input. The file size threshold can be changed dynamically

during the search process[2].

Given the problem, we tried to express an algorithm in pseudo-code solving the problem in the most *"natural"* manner (see below). The algorithm simply accounts for recursively checking each file starting from the directory provided in input, keeping track of all those that have a size which is greater than the reference threshold. While doing this, if a new threshold is communicated and it is greater than the previous one, then the search process must be interrupted and the list of found files so far must be filtered, according to the new threshold. If instead the new threshold is lower than the previous one, then the search process must restart from scratch. When the search process is completed, the list of files is printed on standard output.

Algorithm 3 Description of the reactive file searcher behavior in abstract pseudo-code

```
 1: dir ←get inputPath from user
 2: threshold ←get inputSizeThreshold from user
 3: fileList ←[ ]
 4: for-each element E found in dir do
 5:     if (IsFile(E) && Size(E) > threshold) then
 6:         fileList ← E + fileList
 7:     else if (IsDir(E)) then
 8:         recursively perform the search in E
 9:     end if
10: end for
11: In the meanwhile,
12: every time a new threshold newThr is communicated, do
13:     suspend searching
14:     if (newThr > threshold) then
15:         fileList ← Filter(dir,threshold)
16:         threshold ← newThr
17:         resume searching
18:     else if (newThr < threshold) then
19:         threshold ← newThr
20:         restart from scratch the search process
21:     end if
22: when the search process is finished, print the files in fileList
```

The implementation of this strategy in simpAL is quite straightforward (Figure 7.25). For sake of simplicity, the implementation reported here abstracts from non relevant technical details. The interested reader can find the full sources of the Searcher script both in Section B.1 and in the `examples` folder of the standard simpAL distribution.

The search process is triggered by an Assignator agent (Figure 7.26 (*top*)), which creates a new Searcher agent and assigns to it the SearchFiles task with the designed input parameters

[2]For sake of simplicity threshold changes that occurs when printing in standard output the results, so when the search process is already finished, are discarded—i.e., we consider the threshold updates arrived too late.

```
1  agent-script Searcher implements ReactiveSearcher {
2
3    foundFiles: java.util.List
4    currThr: int
5
6    plan-for SearchFiles {
7      #completed-when: is-done printRes
8
9      currThr = this-task.threshold;
10     searchTask: SearchFilesInDir = new-task SearchFilesInDir(dir: this-task.dir);
11     assign-task searchTask
12
13     when is-done searchTask => { /* print results */ } #act: printRes
14
15     every-time told this-task.newThr => {
16       #atomic
17       if (newThr > currThr){
18         currThr = newThr;
19         foundFiles = filter(foundFiles, currThr)
20       } else if (newThr < currThr) {
21         currThr = newThr;
22         drop-task searchTask;
23         foundFiles.clear();
24         searchTask = new-task SearchFilesInDir(dir: this-task.dir);
25         assign-task searchTask
26       }
27     }
28   }
29
30   plan-for SearchFilesInDir{
31     for-each (elem in this-task.dir) {
32       if (isDir(elem)) {
33         assign-task new-task SearchFilesInDir(dir: elem)
34       } else-if (size(elem)>currThr) {
35         #atomic
36         foundFiles.add(foundFiles)
37       }
38     }
39   }
40
41   task SearchFilesInDir {
42     input-params {
43       dir: String
44     }
45   }
46 }
```

Figure 7.25: Implementation of the reactive file searcher in simpAL.

(lines 4-10). Then, it exploits a clock artifact (line 12) for waiting a specified amount of time (input parameter `delay` of task AssignDynamicSearch) before notifying to the Searcher agent an update of the file threshold (line 14).

The implementation of the Searcher script is shown in Figure 7.25. Its role (ReactiveSearcher, Figure 7.26 (*center-right*)) defines only one task type, the task SearchFiles. This task is characterized by `dir` and `threshold` input parameters, and `newThr` representing the message(s) that can be told for notifying new thresholds. The plan for the task (lines

```
agent-script SearchAssignatorScript implements SearchAssignator in FileSearcherOrgModel {
  plan-for AssignDynamicSearch {
    #using: clock, console
    searcher: ReactiveSearcher;
    searchFiles: SearchFiles = new-task SearchFiles(dir: this-task.dirPath,
        threshold: this-task.startThreshold);
    println(msg: "[Assignator:] Assigning task: startThreshold "
        + this-task.startThreshold + ", waitTime " + this-task.delay
        + " ms, newThreshold " + this-task.newThreshold);
    new-agent Searcher() init-task: searchFiles ref: searcher;
    println(msg: "[Assignator:] Task assigned.. waiting for telling new threshold");
    wait(howLong:this-task.delay);
    println(msg: "[Assignator:] Wait done, telling new threshold");
    tell searchFiles.newThr= this-task.newThreshold;
    println(msg: "[Assignator:] New threshold told")
  }
}
```

```
role SearchAssignator {
  task AssignDynamicSearch {
    input-params{
      dirPath: String;
      startThreshold: int;
      delay: long;
      newThreshold: long;
    }
  }
}
```

```
role ReactiveSearcher {
  task SearchFiles{
    input-params{
      dir: String;
      threshold: long;
    }
    understands{
      newThr: long;
    }
  }
}
```

```
usage-interface ClockInterface {
  obs-prop time: long;
  operation switchOn();
  operation switchOff();
  operation setRate(rate: int);
  operation tick(currentTime: long);
  operation wait(howLong: long);
}
```

Figure 7.26: Some other relevant sources related to the reactive file searcher example: (*top*) SearchAssignatorScript implementation, (*center*) ReactiveSearcher and SearchAssignator roles definitions, (*bottom*) ClockInterface artifact usage interface.

```
org-model FileSearcherOrgModel{
  workspace main{
    Assignator: SearchAssignator;
    Clock: ClockInterface;
  }
}
```

```
org FileSearcherOrg implements FileSearcherOrgModel{
  workspace main {
    Clock = ClockArtifact()
    Assignator = SearchAssignatorScript() init-task:
        AssignDynamicSearch(dirPath: "...",
                            startThreshold:10000,
                            delay:60,
                            newThreshold: 1000)
  }
}
```

Figure 7.27: (*left*) Definition of the FileSearcherOrgModel organization model, and (*right*) of the concrete organization instance FileSearcherOrg implementing it.

```
[Assignator:] Assigning task: startThreshold 1000, waitTime 60 ms, newThreshold 10000
[Searcher:] Found relevant file: File1
[Searcher:] Found relevant file: File2
...
[Assignator:] Wait done, telling new threshold
[Searcher:] New threshold found. Was 1000 now is 10000 so far have been found 31 files
[Searcher:] Threshold greater than the previous one, we can simply filter the results...
[Searcher:] After filtering the files are 4
[Searcher:] Found relevant file: File56
...
[Searcher:] Search ended, the found files are:
 - File1
 - File56
 - ...
[Searcher:] Found 12 files in total with the desired size
```

Figure 7.28: Execution trace of the reactive file searcher example with: `startThreshold` 1000 bytes, Assignator `waitTime` 60ms and `newThreshold` 10000 bytes. From the trace it is possible to see how, once received the new threshold the Searcher – since it is the case of a threshold increase – filters the list of files found so far (from 31 to 4) before resuming the search process and printing on the console the result.

```
[Assignator:] Assigning task: startThreshold 10000, waitTime 60 ms, newThreshold 1000
[Searcher:] Found relevant file: File1
[Searcher:] Found relevant file: File2
...
[Assignator:] Wait done, telling new threshold
[Searcher:] New threshold found. Was 10000 now is 1000 so far have been found 3
[Searcher:] Threshold lower than the previous one, we need to restart from scratch...
[Searcher:] searchTask dropped
[Searcher:] searchTask re-assigned
[Searcher:] Found relevant file: File1
[Searcher:] Found relevant file: File2
[Searcher:] Found relevant file: File7
...
[Searcher:] Search ended, the found files are:
 - File1
 - File2
 - File7
 - ...
[Searcher:] Found 194 files in total with the desired size
```

Figure 7.29: Execution trace of the reactive file searcher example with: `startThreshold` 10000 bytes, Assignator `waitTime` 60ms and `newThreshold` 1000 bytes. From the trace it is possible to see that, once received the new threshold the Searcher – since it is the case of a threshold decrease – is forced to drop the current `searchTask` and re-instantiate a new one with the new size threshold.

6-28) has an autonomous part and a reactive one. The autonomous behavior accounts for instantiating and self-assigning the sub-task `searchTask` of type SearchFilesInDir (line 10) first, providing as input parameter the starting directory which corresponds to the `dir` parameter of the `SearchFiles` task, and then printing the results when the sub-task is completed (line 13).

The plan for handling the SearchFilesInDir private sub-task (lines 30-39) collects recursively, instantiating a new SearchFilesInDir sub-task for each sub-directory found (lines 32-33), the set of files that meet the desired size.

The reactive behavior is given by an action rule (lines 15-27) which handles the updates of the threshold on the basis of the strategy described above. The threshold value can be updated by the agent that has assigned the SearchFiles task to the ReactiveSearcher by sending to it a `newThr` message. In the case of a threshold increase (lines 17-19) the action rule simply filters the list of files found so far. In the case of a threshold decrease instead (lines 20-26), the task `searchTask` is first dropped (line 22) and then re-instantiated (lines 24-25) in order to restart the search from scratch since there is no guarantee that we have not already discarded files that have become relevant given the new threshold. The action rule block is executed atomically so that all the SearchFilesInDir sub-tasks and related intentions are suspended until the action rule is executed up to completion, giving hence the opportunity to realize the required integration between autonomous and reactive behaviors.

So, the solution in simpAL makes it possible to essentially keep the same structure of the algorithm in pseudo-code. This has clear advantages in term of readability of the code, which is not polluted by idiosyncrasies of the computing/programming model.

Launching the application specifying different values for the input parameters `startThreshold`, `newThreshold` and `delay` during the creation of the AssignDynamicSearch task, it is possible to test the different reactive and adaptive behaviors of the searcher agent. Examples of execution traces are reported in Figure 7.28 and Figure 7.29.

7.4.2 Implementation of the Ricart-Agrawala's algorithm

In this section we describe a simpAL implementation of the Ricart-Agrawala's algorithm [RA81] for realizing distributed mutual exclusion among `N` peers. Like many other concurrent and distributed algorithms that imply the coordination of independent processes, it calls for *superimposing* to the autonomous behavior of the individual processes some behavior which is functional to achieve some coordination objective.
A simple description of the Ricart-Agrawala's algorithm follows, inspired by the one provided in the original paper [RA81]. Each peer that takes part in the distributed coordination algorithm has a unique ID. A peer enters its critical section (CS) only after all the other peers have been notified of its request and have sent a reply granting their permission. A peer making an attempt to invoke mutual exclusion sends a request message to all other peers. Upon receipt of the request message, the other peer either sends a reply immediately or defers a response until after it leaves its own critical section. The algorithm is rooted on the fact that a peer receiving a request message can immediately determine whether the requesting peer or itself should be allowed to enter its critical section first. A reply message is returned immediately if the originator of the request message has priority; otherwise, the reply is delayed. The priority order decision is made by comparing a sequence number included in each request message. The numbers chosen

Algorithm 4 Implementation in pseudo-code of a process-based peer following the Ricart-Agrawala's algorithm adapted from [BA05]

```
1:                                                          ▷ SHARED VARIABLES
2:  myID ← peerID
3:  mySeqNum ← 0
4:  deferredList ← []
5:  maxSeqNum ← 0
6:  reqCS ← false
7:  otherPeers ← listOfOtherPeers
8:  ---------------------------------------------------------------------------
9:                                                          ▷ MAIN PROCESS
10: while true do
11:     DOJOBINNCS( )
12:     reqCS ← true
13:     mySeqNum ← maxSeqNum + 1
14:     for (each peer P in otherPeers) do
15:         SEND(request, P, myID, mySeqNum)
16:     end for
17:     awaits replis from all otherPeers
18:     DOJOBINCS( )
19:     reqCS ← false
20:     for (each peer P in deferredList) do
21:         SEND(reply, P, myID)
22:     end for
23:     deferredList ← [ ]
24: end while
25: ---------------------------------------------------------------------------
26:                                          ▷ PROCESS MANAGING THE SENDING OF REPLIES
27: while true do
28:     RECEIVE(request, source, reqID)
29:     maxSeqNum ← MAX(reqID, maxSeqNum)
30:     if (reqCS && (reqID > mySeqNum
31:                                  || (reqID == mySeqNum && peerID > myID))) then
32:         deferredList ← deferredList + peerID
33:     else
34:         SEND(reply, source, myID)
35:     end if
36: end while
```

by the peers must be *monotonic*, in the sense that a peer will choose a number that is higher than all other sequence numbers it knows about. Ties concerning sequence numbers are broken by comparing the peers IDs: a peer with a lower ID has priority over a peer with a higher ID.

The algorithm in pseudo-code is reported above, inspired to the one described in [BA05]. It

requires the implementation of two different macro-behaviors for each peer, one for managing the pro-active part – i.e., cyclically enter the CS once all the replies have been collected, do some job in CS, then leave the CS and send deferred replies (if needed) – and another one for managing the reactive part—i.e., handle the reception of incoming request messages. These two parts need to interact in a synergistic manner in order to guarantee the correct functioning of the algorithm.

To preserve the required level of reactivity while repeatedly entering/leaving the CS – i.e., reply in a timely fashion when is received a request for entering the CS from a peer that has priority over the current one – classical solutions (see the pseudo-code above) require: *(i)* the introduction of two active entities (processes/threads or actors) for every peer, which are in charge of one macro-behavior each; and *(ii)* the coding of specific coordination protocols between such computing entities, to guarantee their correct functioning w.r.t. the algorithm's logic—e.g., in process/thread based solutions is required the use of shared variables and locks to guarantee the absence of race conditions. So, there is an apparent abstraction gap between the strategy upon which the algorithm is rooted and the actual implementations.

The availability of abstractions integrating autonomy and reactivity makes it possible in this case to provide a simpler solution. The solution in simpAL shown below is composed by a single agent, which encapsulates both the pro-active and reactive parts.

```
1  agent-script SimplePeer implements Peer {
2
3    /* global beliefs */
4    myID: int; myCommManager: CommManager; othersCommManagers: CommManager[]; rd: ReqData;
5    reqCS: boolean = false; timesInCS: int = 0; maxSeqNum: int = 0; mySeqNum: int;
6    deferredList: ArrayList = new-object ArrayList(); nReplies: int = 0; numPeers: int;
7
8    plan-for DoJob {
9      #using: console@main, myCommManager, othersCommManagers
10     ... /* initialization of global beliefs */
11     {
12       #to-be-repeated
13       do-task new-task doJobInNCS();
14       do-task new-task Prologue();
15       do-task new-task doJobInCS();
16       do-task new-task Epilogue();
17       timesInCS++;
18       println(msg: "[Peer"+myID+"]" + timesInCS + " iteration(s) done")
19     }
20
21     every-time changed request in myCommManager as: rd => {
22       #atomic
23       if (rd.reqID> maxSeqNum) {
24         maxSeqNum = rd.reqID
25       };
26       if (reqCS && (rd.reqID > mySeqNum || (rd.reqID == mySeqNum && rd.peerID > myID))){
27         deferredList.add(rd.peerID); /* the peer has to wait */
28       } else {
29         placeReply() on othersCommManagers[rd.peerID] /* the peer can proceed */
30   }}}
31
32   plan-for doJobInNCS { ... }
33
34   plan-for Prologue {
```

```
35      #using: console, myCommManager, othersCommManagers
36      #completed-when: nReplies==numPeers-1
37
38      do-task new-task ChooseTicketForCS();
39      for-each (commManager in othersCommManagers) {
40        placeRequest(rd: new ReqData(myID, mySeqNum)) on commManager
41      }
42
43      every-time changed numReplyReceived in myCommManager => {
44        nReplies++
45    }}
46
47    plan-for doJobInCS { ... }
48
49    plan-for Epilogue {
50      #using: console@main, othersCommManagers
51      reqCS = false;
52      for-each (elem in deferredList) {
53        placeReply() on othersCommManagers[deferredList.get(elem)];
54      };
55      /* reset variables for the next iteration */
56      deferredList.clear(); nReplies = 0
57    }
58
59    plan-for ChooseTicketForCS {
60      #atomic requestCS = true; mySeqNum = highestSequenceNumber + 1
61    }
62
63    task doJobInNCS{}
64    task Prologue{}
65    task doJobInCS{}
66    task Epilogue{}
67    task ChooseTicketForCS{}
68  }
```

The peer behavior is managed by the plan reported at lines 8-30, related to the task DoJob. The plan encapsulates both the pro-active and reactive behaviors described above. The pro-active one is in charge of the soft action rule block reported at lines 11-19. Through this block the agent cyclically executes the following steps: *(i)* firstly, it does some job outside the critical section (doJobInNCS sub-task line 13, managed by the plan at line 32); *(ii)* then, as soon as it requires to enter its CS, it executes a sub-task that manages the pre-protocol for entering it, following what defined by the Ricart-Agrawala's algorithm (Prologue task line 14, managed by the plan at lines 34-45); *(iii)* when the Prologue task has been completed – i.e., all the other peers' replies have been received (line 36) – the agent enters its CS and executes some job inside it (doJobInCS task line 15, managed by the plan at line 47); *(iv)* finally, it executes a sub-task for managing the algorithm's epilogue (Epilogue task line 16, managed by the plan at lines 49-57) and prints a log message indicating the number of times it has been in CS so far `console` (line 18).

Given the current limits of simpAL direct communication model[3], a communication artifact (Figure 7.30) is used for each agent, functioning as a message box. In particular communication

[3]Currently simpAL does not support peer-to-peer message passing. Being the communication protocols specified on a task basis, messages can be exchanged only among the agent who assigned a task and the task assignee.

```
usage-interface CommManager {

  obs-prop request: ReqData;
  obs-prop numReplyReceived: int;

  operation placeRequest(req: ReqData);
  operation placeReply();
}
```

```
public class ReqData {
  public int reqID;
  public int peerID;
  public ReqData(int peerID, int reqID) {
    this.reqID = reqID;
    this.peerID = peerID;
  }
}
```

```
artifact SimpleCommMngr
                implements CommManager{

  init(){
    numReplyReceived = 0;
  }

  operation placeRequest(req: ReqData){
    request = req;
  }

  operation placeReply(){
    numReplyReceived++;
  }
}
```

Figure 7.30: Implementation of: *(i)* the CommManager usage interface (top-left), *(ii)* the SimpleCommMngr artifact implementing it (right), and *(iii)* the ReqData class used to model a request for entering the distributed critical section (bottom-left).

artifacts are used to: *(i)* send to a peer a request for entering the CS (`placeRequest` operation, which updates the artifact's `request` observable property with the request provided in input), and *(ii)* to send a reply related to a previously received request (`placeReply` operation, which increments the artifact's observable property `numReplyReceived` counting the number of replies received so far). The request message is modeled via the Java class ReqData (Figure 7.30, bottom-left), storing as public fields both the sequence number associated to the request (`reqID`) and the ID of peer that has sent it (`peerID`).

The plans that manage the Prologue and the Epilogue sub-tasks are quite simple. The former first computes a new ticket (sub-task ChooseTicketForCS, line 38), then it sends the requests for entering the CS (lines 39-41). A reaction is used to count the number of replies received (lines 43-45). The block completes when all the replies are received (line 36). The Epilogue instead (lines 49-57), sends the replies related to request messages sent by other peers that have been deferred (if any).

The reactive behavior in the main plan (reaction rule at 21-30) allows for managing incoming requests following the strategy defined by the algorithm, through the help of a CommManager artifact. For each `request` message received through the myCommManager (line 21) – i.e., the personal communication artifact of the peer – a reply is either immediately sent back (lines 28-29) or delayed (lines 26-27)—i.e., added to the deferredList and sent only later on, when the agent is executing the Epilogue sub-task.

The action rule blocks of the reaction rule in the main plan and of the ChooseTicketForCS sub-task must be tagged as atomic, since when executing those blocks of actions the agent can not carry on other concurrent activities that could interfere with these ones.

Concrete solutions that adopt two processes/threads for each peer typically need to use low level synchronization mechanisms to coordinate them, such as semaphores. This is not neces-

sary in the simpAL solution, which allows for keeping a structure quite similar to the abstract one described in the beginning.

7.5 The simpAL Integrated Development Environment

The availability of an adequate set of tools – starting from a powerful Integrated Development Environment (IDE) – able to support and help programmers during the entire development process of an application is, in general, an important requirement in oder to make a programming language usable and adopted outside the mere academics contexts. Accordingly, in the context of this thesis, some efforts have been put in the engineering of an adequate IDE for simpAL. The rationale that guided the engineering of the simpAL IDE is quite simple: trying to provide developers the best means for coding, debugging, executing and inspecting simpAL applications.

In the remainder of this sub-section we present the results of our efforts in exploiting the Eclipse [Ecla] ecosystem for the realization of an Eclipse-based IDE for simpAL [SR11b]. First, we outline the requirements of the IDE, and then we describe its architecture and main features, w.r.t such requirements.

7.5.1 IDE Requirements

The requirements of the simpAL IDE can be split in two different groups. On the one side it is possible to identify a first group of requirements related to features that can be found on a big majority of modern integrated development environments: *(i)* creation and management of multiple projects related to different applications and the opportunity to explore the content of such projects; *(ii)* proper file editors providing features such as context-assist, code completion, template proposals, cross-referencing, etc.; *(iii)* a set of useful views that help programmers during their development experience—output views for inspecting the outputs produced by the application, views for representing the outline of structured files, etc. We need *at least* all these features in our simpAL IDE.

On the other side, being simpAL applications inherently concurrent, distributed and rooted on specific agent-oriented abstractions, a set of more advanced and specific IDE requirements are needed in order to give programmers a seamless development experience. The first one concerns the definition of a proper organization of the sources inside a project in the IDE, given the specific abstractions of the simpAL programming model (roles, agents, artifacts, workspaces, etc.). There is the need to provide developers the best project organization in order to make it easy and immediate the browsing among the different sources of the project, and the understanding of the overall structure of the application.

A second requirement concerns the transparent management of the deployment and execution of distributed applications on top of the simpAL distributed runtime infrastructure. To this end, a properer integration between the IDE and the runtime infrastructure must be engineered. Another requirement, which is related to the previous one, concerns the management of the

simpAL runtime infrastructure in the different network nodes in which it has been installed—i.e., having proper means to start/stop/inspect the infrastructure in both local and remote nodes directly from the IDE.

Finally, there is the need of a powerful debugger for inspecting and monitoring simpAL applications. Such debugger should allow to: *(i)* inspect transparently both local and remote agents/artifacts, *(ii)* inspect all the agents' ongoing tasks, possibly choosing one of them for step-by-step execution while other ones are carried out in background smoothly and without side-effects, *(iii)* suspend one or more tasks assigned to an agent, *(iv)* debug on a step-by-step basis the execution of artifact operations, etc.

7.5.2 IDE Overview

Starting from the requirements described in the previous section, we engineered a working prototype of an Eclipse-based IDE for simpAL. The IDE has been engineered on top of the Eclipse ecosystem by exploiting in synergy the Eclipse Plugin Development Environment (PDE) [Eclb], part of the Eclipse SDK, and the Xtext language development framework [Ite]. For its realization we used as a starting point our previous work concerning the development of an Eclipse-based IDE for the JaCa platform, which has been presented in [SLNR11].

Figure 7.31 depicts the plugin-based IDE architecture, focusing on the dependencies among the various plugins introduced for its realization. The `simpal.ide` plugin represents the core of the IDE. It has been realized using PDE for defining a set of Eclipse extensions – relying on existing Eclipse extension points[4] – able to cover most of the requirements outlined in the previous section.

In particular, this plugin covers almost entirely the first group of requirements by providing:

- A set of wizards for managing the creation of new simpAL projects, new agents and new artifacts.
- The definition of a custom Eclipse nature and structure for simpAL projects (Figure 7.32 *(a)*).
- The definition of a specific Eclipse perspective for simpAL (Figure 7.32 *(b)*) that shows a proper set of views: a custom outline view (Figure 7.32 *(c)*), a problems view (Figure 7.32 *(d)*) and a custom navigator (Figure 7.32 *(e)*, see below for further details about the navigator).
- The implementation of an incremental project builder (Figure 7.32 *(f)*) that is in charge of invoking the simpAL compiler as soon as some relevant sources are changed. Possible errors found during the compilation process are listed in the problem view contained in

[4]Extension and extensions points are two mechanisms introduced by the Eclipse platform to enable the customization and the extension of functionalities provided by plugins. For more details about these mechanisms see [CR08].

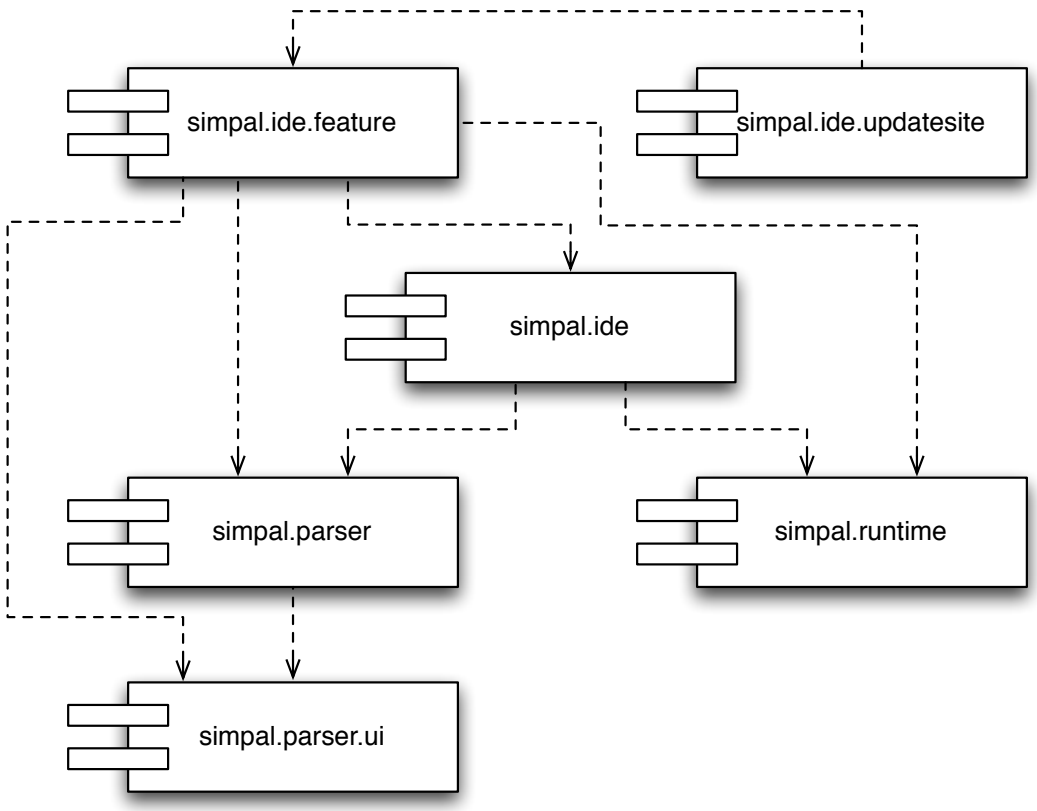


Figure 7.31: Plugin-based architecture of the simpAL IDE.

the simpAL perspective, by exploiting the usual support available in Eclipse for managing error/warning markers.

- The implementation of the required integration between the IDE and the simpAL distributed runtime infrastructure for managing the execution of (possible distributed) simpAL applications. Currently this integration only allows to run distributed applications on top of an already up and running simpAL infrastructure—i.e., there is no support yet to start/stop the simpAL kernel on remote nodes from the IDE.
- The implementation of a set of new commands and handlers for triggering the deploy/run-/termination of simpAL applications (Figure 7.32 *(g)*).
- The implementation of a custom console view to show the output of running simpAL programs (Figure 7.32 *(h)*).

Moreover, for what concerns the second group of requirements this plugin provides: *(i)* a custom debugger for inspecting both local and remote application parts (i.e., agents, artifacts, work-

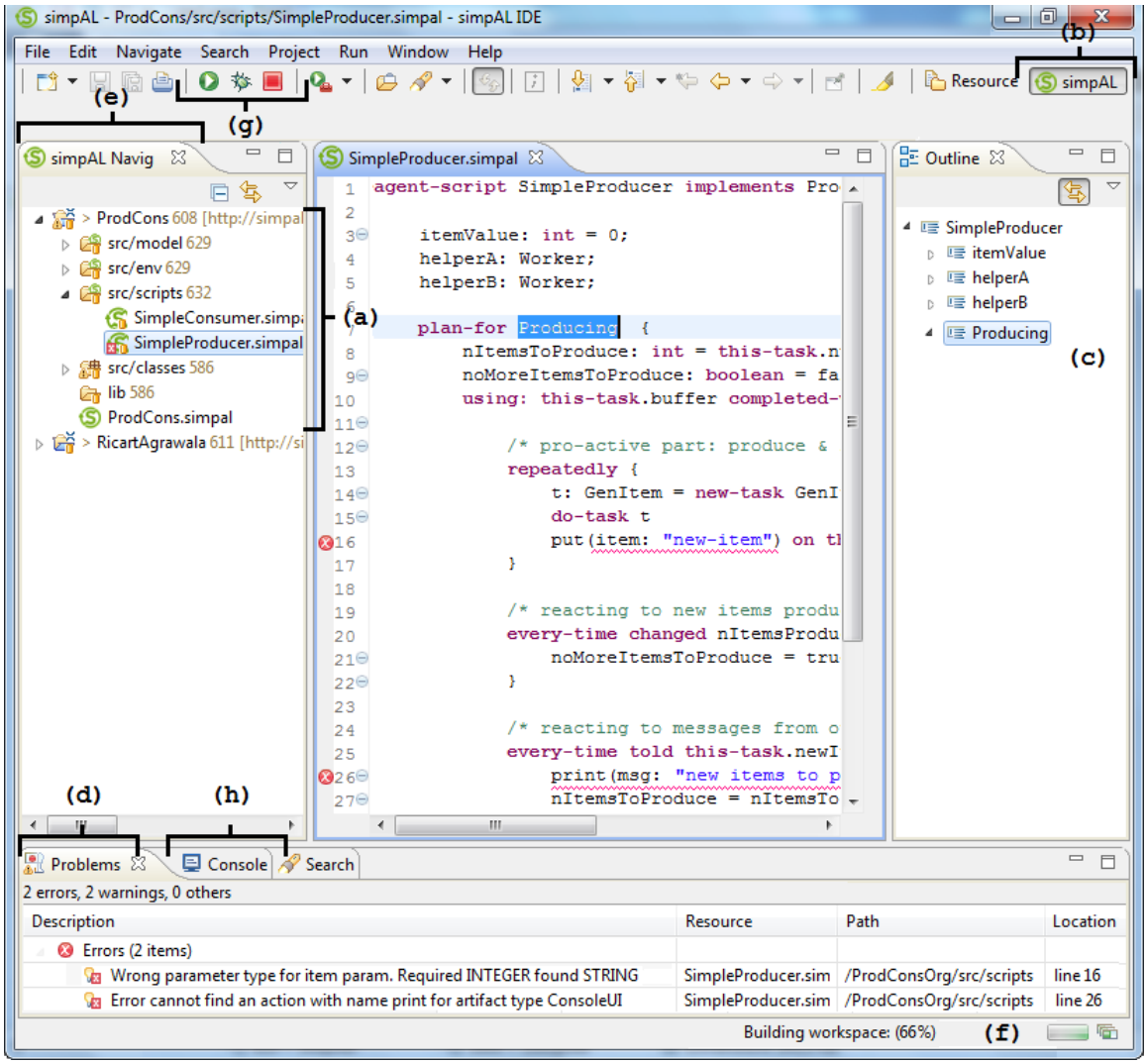


Figure 7.32: simpAL IDE Overview.

spaces), and *(ii)* the definition of a custom navigator view which shows the content of simpAL projects with the desired organization, hiding all the details that are not useful when coding a simpAL application—i.e., the custom navigator defines proper filters to hide non-relevant resources (e.g., the `build` folder in which the compiled sources are stored), and custom content providers in order to display the most immediate and intuitive project organization.

As reported by Figure 7.31 the `simpal.ide` plugin logically depends from two other plugins: the `simpal.parser` and the `simpal.runtime`. The former represents the Xtext-based plugin used to describe the grammar that defines the syntax of the simpAL language. Starting from the grammar specification, Xtext automatically generates an Eclipse-based file

editor that can be used for writing programs in the language defined by such a grammar. This auto-generated file editor is implemented by the `simpal.parser.ui` plugin (since it is generated from the `simpal.parser` plugin it logically depends from it in Figure 7.31). The editor exploits the full power of the Xtext language development framework, in synergy with some slight ad-hoc customizations, in order to provide developers a full-blown file editor able to ease their coding experience. Currently, it supports: syntax highlighting, code completion and context assist.

The `simpal.runtime` plugin instead contains the implementation of both the simpAL distributed runtime infrastructure and virtual machine. It is exploited by the `simpal.ide` plugin to easily deploy and run, from the IDE, distributed simpAL applications on already up and running simpAL infrastructural nodes.

The last two plugins `simpal.feature` and `simpal.updatesite` are used to manage the distribution of the IDE. The former is used to define an Eclipse feature that keeps track of all the plugins introduced for realizing the IDE (see the logical dependency in Figure 7.31). The latter instead exploits the feature defined by the former plugin in order to configure a classic Eclipse update site[5] from which it is possible to download the simpAL IDE.

Besides being distributed through the update site, the simpAL IDE is also available as a standalone Eclipse product for all the main platforms (Windows, Linux, Mac), which can be downloaded from the reference simpAL website [Ric].

7.6 Remarks on Performance

The importance of abstraction in programming is well-known. However, often a higher-level of abstraction comes along with a *price* in terms of performance, which could be either acceptable or not depending on the specific application domain considered. In this perspective, in the abstraction layer introduced by simpAL there are two aspects that could be critical for efficient programs execution.

The first aspect concerns the strategy adopted for coupling agent and artifact execution w.r.t. the physical level of concurrency provided by the Operating System (OS). It is apparent that the simplest strategy in which there is one raw OS thread for each agent and each artifact does not scale, as soon as the efficient execution of programs with hundreds/thousands of computing entities is considered. This is also true in the case of actor-oriented technologies, in which using one physical thread for managing the execution of each actor is not a feasible solution. To solve this issue simpAL introduces a logical level of concurrency, as found in reference implementation of state-of-the-art actor oriented framework and languages (e.g., Erlang and Scala, in which thousands of actors can be in execution on the same machine). Then at runtime, the execution of all the agents and artifacts on a simpAL node is managed by a pool of threads, whose size is dimensioned on the basis of the number of processors available on that node. So, programmers can freely write simpAL programs with hundreds/thousands of agents and artifacts

[5] `http://simpal.sourceforge.net/update-site/`

which could be executed – with different performances, given the number of CPUs available – on machines ranging from mono-processor PCs to highly-capable many-core servers. However, also this logical level of concurrency has a price, which in this is related to the scheduling of agents – so as to carry on their execution cycle – and the execution of operations over artifacts.

The second aspect is related to the overhead introduced by the agent control architecture. We identified the following critical points:

- Useless cycling — the sense-plan-act cycle described in Section 7.1.2 is executed continuously even in the case of – for instance – a simple sequence of actions that must be executed without reacting to any external events (e.g., Figure 7.25 lines 17-26). This brings a penalty on performance when considering – in particular – the execution of pure computational blocks compared to e.g. actor technologies implementing a macro-step semantics, in which once a message is received the corresponding handler is executed up to completion, without further involving the event control loop.

- The process of action selection at each agent cycle — currently, this process consists in evaluating each action rule given the current set of blocks in each intention stack.

- The management of beliefs, observable properties and artifacts private variables — currently all these are managed through maps, accessed via a string-based key. This naive strategy results in a severe overhead indeed, compared to e.g. classic techniques used in compiled languages where access to variables is index-based.

To start investigating the performance issue related to this second aspect, we made two tests comparing the performance of simpAL with reference representatives of both actor-oriented and agent-oriented technologies. In particular, Erlang and ActorFoundry are considered on the actor side. To the best of our knowledge, Erlang is the fastest actor based technologies available in the state-of-the-art [Arm10]. Erlang virtual machine has been developed in C with a particular clue on efficiency and scalability—in fact, it allows the concurrent execution of a very large number of interacting processes (i.e., actors) that can exchange up to millions of messages per second. ActorFoundry is a Java-based actor technology, i.e. the runtime runs on top of the JVM – like simpAL – implementing a strict actor semantics. Recently, it was updated to include optimizations that make it one of the fastest Java-based actor technologies [KSA09]. On the agent side, we consider ***Jason*** [BHW07], which is one of the most well known and mature BDI-based agent-oriented programming language available in the state-of-the-art.

These three platform provide then a good spectrum to understand where simpAL currently is. On the one side, given simpAL computation model and its Java-based runtime, we expect to have better performance in general with respect to ***Jason***, which is Java-based too but it strongly relies on logic programming. This because ***Jason*** has been primarily introduced to tackle problems in distributed artificial intelligence contexts, and not for general-purpose computer programming. On the other side, we expect to not be as efficient as ActorFoundry, given the differences between the agent and actor computation model. But, being based on the JVM

Language	N=10K	N=100K	N=1M	N=10M
Erlang (V1)	0,31s	0,47s	2,1s	18,2s
Erlang (V2)	0,38s	1.13s	8,48s	81,24s
ActorFoundry (V1)	0,34s	0,95s	6.33s	59,36s
ActorFoundry (V2)	0,36s	1,72s	14,78s	145,88s
Jason (with tell)	7,93s	1185s	>1h	>1h
Jason (with achieve)	6,94s	1069s	>1h	>1h
simpAL	2,05s	9,33s	82,32s	804,46s

Table 7.1: Execution time in seconds of the first test program in Erlang, ActorFoundry, ***Jason*** and simpAL. The time reported in each cell refers to the average of twenty different runs.

too, the performance of ActorFoundry could be for us a good target to look for. Erlang is useful instead to understand what is the current upper bound of performance.

The benchmarks have been executed using Erlang version 5.8.3, ActorFoundry 1.0, Jason 1.3.8 and simpAL version 0.3, on a PC with a Intel Core 2 Duo P8400 2.26GHZ (dual core) and 3GiB RAM. The full source code of the test programs have been included in the appendix (Section B.2).

The first test program is a slightly extended version of the first example described in Section 7.3.1, where the execution of the task T and the sending of the `react` message have been repeated for 10K, 100K, 1M and 10M of times respectively. Each program has been executed for twenty times. The average of the execution time experienced in the tests is reported in Table 7.1. Two slightly different versions of the test program are considered for what concerns ***Jason***, Erlang and ActorFoundry. In actor-oriented solutions, the first version (V1) is based on the first implementation of the example discussed in Section 7.3.1. The second one (V2) is rooted on the solution in which a network of actors is introduced in order to obtain the same level of reactivity available in agent-oriented implementations of the test. For what concern ***Jason*** instead, the two versions considered differ on the way in which the `react` message is sent. In the first version is used the `tell` performative – i.e., the performative for exchanging information between agents in ***Jason***. In the second version the `achieve` performative is used instead—i.e., the performative that assigns a new goal to an agent. The first solution is the correct one – from a conceptual point of view – w.r.t. the logic of the test—i.e., we want to exchange some information (the `react` message) and not assign a new goal. However, it is much slower compared to the second one, probably due to the different ways in which the ***Jason*** runtime manages the access to the belief and goal base. So both are considered.

The second test program is the so called *thread-ring* test[6]. 503 workers linked in a ring, have to pass a token through the ring for a fixed number of times (N). This is a quite well known performance test in the context of actor-oriented languages [KSA09], in particular to

[6] `http://benchmarksgame.alioth.debian.org/u32/performance.php?test=threadring`

Language	N=10K	N=100K	N=1M	N=10M
Erlang	0,32s	0,41s	1,43s	10.59s
ActorFoundry	0,63s	0,68s	2.21s	16,87s
Jason (with tell)	2,55s	38,53	>1h	>1h
Jason (with achieve)	2,1s	4,26s	26,2	244,92
simpAL	1,49s	3,37s	20,96s	202,29s

Table 7.2: Execution time in seconds of the thread-ring test in Erlang, ActorFoundry, ***Jason*** and simpAL. The time reported in each cell refers to the average of twenty different runs.

measure the performances related to the exchange of messages (i.e., the test measures pure reactivity). Table 7.2 shows the average execution time for N equals to 10K, 100K, 1M and 10M, with the same configuration described for the first test—i.e., same machine, same versions of the programming languages/frameworks used and same number of runs. Also in this case we developed two different versions of the test for ***Jason***. One in which is used the tell primitive to exchange the token among the workers, and another one in which the token is passed using the achieve primitive.

As expected, the results reported in the tables show that simpAL is currently positioned in the middle between ***Jason*** and actor technologies, however with a significance distance with respect to both ActorFoundry and Erlang. In particular, the first test shows that simpAL is currently about from 6 to 13 times slower than ActorFoundry (V2 and V1 versions) and from 10 to 45 times with respect to Erlang. In the second test, simpAL is about 12 times slower than ActorFoundry and 20 times with respect to Erlang. Instead, it performs as expected compared to ***Jason***.

7.7 Final Remarks

This section concludes the chapter by presenting: *(i)* a comparison with state-of-the-art agent-oriented programming approaches (Section 7.7.1), and *(iii)* a discussion about current weaknesses and limitations of the simpAL programming language (Section 7.7.2).

7.7.1 Comparison with State-of-the-Art Agent-Oriented Programming Approaches

In this sub-section we provide a brief comparison between simpAL and state-of-the-art agent-oriented programming approaches, focusing on the main similarities and differences among them.

We remarked several times that the simpAL programming model takes strong inspiration from previous work introduced in the context of the *"classical"* agent research community.

So, from a high-level point of view, conceptually the simpAL programming approach can be considered quite similar to other approaches in the state-of-the-art, in particular to the ones promoted by JaCa and JaCaMo: a software system is programmed by means of multiple agents situated inside an environment, which need to suitably interact both with the environment's resources and among themselves in order to achieve the application's objectives/goals. Nevertheless, there are several substantial differences. Besides being inspired by agent-oriented abstractions and computational models, simpAL founds its roots on well known general purpose features and programming principles that have been introduced in the context of mainstream software development and related programming languages: e.g., typing, polymorphism, strong separation of concerns between the *"interface"* and the implementation, etc. Along the chapter we discussed in detail the benefits related to the introduction of these features in the context of an agent-oriented programming language that targets general purpose computing.

In the literature it is possible to find few exceptions (e.g., [FSS05]) of APLs/frameworks that, at least conceptually, try to shift the perspective from the development of intelligent software systems in the distributed artificial intelligence context to programming in the large. However, we argue that besides sharing the same objectives, from a programming and software engineering point of view, these proposals, differently from simpAL, have not fully investigated the injection in the language/framework of those fundamental features that characterize, in our opinion, general-purpose programming.

A final main difference concerns the support provided for programming complex interaction protocols and social structures that norm and regulate the overall social behavior of a software system. Differently from agent-oriented programming approaches in the state-of-the-art – e.g., JaCaMo – for now simpAL does not provide first-class programming constructs to address directly these issues. However, application-specific coordination and interaction mechanisms, even if not regulated by underlying conceptual models (e.g., MAS organizational models Section 3.4.2), can be freely implemented in simpAL through the engineering of proper coordination artifacts.

7.7.2 Current Limitations

As already mentioned, the simpAL programming language is currently in its infancy. Therefore, not surprisingly, there are limitations and aspects that still need to be addressed, which here we logically divide in two main groups.

The first group is related to the current support for typing, polymorphism and inheritance. In particular, for what concerns typing, simpAL introduces an explicit notion of type for agents, artifacts and organizations, which are mainly used to: *(i)* properly characterize the different elements of a program, and *(ii)* perform static error checking controls (Section 7.3.2). On the one side, this is a clear advance w.r.t. what is available in state-of-the-art APLs. On the other side, this is just a first level support (e.g., w.r.t. the one provided by object-oriented languages), which still lacks an explicit formalization of both the type notions introduced and of the overall simpAL type system. These absences also hamper the formal definition of sub-typing relations

for agents, artifacts and organizations. Actually, from a practical and implementation point of view we discussed and investigated up to a good point the introduction of sub-typing relations in simpAL. However, we intentionally decided to defer the introduction of such relations in the language until their role and semantics are made clear by the study of proper formalizations, in order to avoid (possibly) heavy modifications on the language—i.e., in the case in which our *"practical"* implementation is not fully sound from a formal and semantic point of view.

Similar considerations to the ones just made for typing, also apply for the support provided for inheritance and polymorphism. For now, simpAL does not provide an explicit support for the former. Instead, basic forms of polymorphism are supported for agents, artifacts and organizations (see Section 7.3.3). As in the case of typing, from a practical point of view, we started to study more advanced forms of polymorphism – i.e., agents implementing multiple roles, artifacts implementing multiple interfaces – but we decided to defer their introduction in the language, until we are able to validate them also from a more formal point of view.

The second main group concerns the synergistic integration and exploitation in simpAL of the underlying object-oriented level. On the one side, as have been showed in various examples, in simpAL programs Java objects are used to define the data model, and it is quite straightforward the use of Java code in both agent scripts and artifact templates. On the other side, there are still some integration aspects that could be improved. In particular:

- The definition of arrays and collections (List, HashMaps, etc.) of agents and artifacts (used e.g. in Section 7.4.2) is possible, but relies on low level implementation mechanisms.

- Events related to updates of artifacts' observable properties bound to non-primitive Java objects, are generated only when the reference of the object stored by an observable property is changed—i.e., changes of the object's inner state (e.g., changing an object field) do not automatically generate an observable property update event. Indeed, method execution in simpAL is implemented under the hood via Java reflection, hence there are not explicit means to know when the execution of a method leads to the modification of an object's internal state. When needed, in artifacts implementation it is possible to use the `generateEvForObsProp ObsProp` statement, to force the generation of an update observable event, related to the specified observable property (`ObsProp`).

- Java generics are not supported in simpAL. As a consequence, there is still the need to operate several cast operations in simpAL code, e.g. when working with Java collections.

Part IV

Conclusion

8

Conclusion and Future Work

In this thesis we investigated the application of agent-oriented programming as a high-level general purpose programming paradigm, evolution of actor-oriented and object-oriented ones, as a possible answer to the challenges and complexities introduced by *"the free lunch is over call"* (Chapter 1). To this end, we first took active part in the study and definition of integrated – i.e., concerning the synergistic use of multiple programming dimensions – programming approaches and concrete technologies at support of the engineering of multi-agent systems for the (distributed) artificial intelligence context (part II of this thesis). The results obtained from these studies are one of the two macro-groups of contributions of this dissertation (Chapter 1).

Then, starting from the background and the expertise built with the previous work, we focused our efforts in the engineering of the simpAL programming language and its related ecosystem (part III of this thesis). We argue that the obtained results – we referred to them as the second macro-group of contributions of this dissertation in Chapter 1 – are promising. simpAL is the first programming language that makes it possible to exploit an agent-oriented level of abstraction for the development of concurrent and distributed software systems, which has its foundations rooted on all that key concepts and features that have been introduced and developed in the history of modern mainstream programming languages – e.g., typing, polymorphism, etc. – properly revisited for an agent-oriented language. In particular, we believe that, even if some aspects have not been fully explored yet – e.g., inheritance, sub-typing relations, etc. – with our work we have been able to delineate the backbone at the base of the development of an agent-oriented language targeting general purpose computing, rooted on the basic principles and practices of mainstream programming and software development.

The benefits of adopting of an agent-oriented level of abstraction for the development of concurrent and distributed systems have been discussed in the thesis by presenting a set of purposive examples. In particular, the main achievement has been the definition of proper abstractions – i.e., the simpAL agent architecture, the agent plan and action rule models – which allow to integrate and unify in a seamless manner autonomous and reactive (event-driven) behaviors (Section 7.3.1).

We envision several avenues for further research, divided in two main lines. The first one is related to the engineering of agent-oriented technologies for the (D)AI context:

- A first future contribution would be the investigation of the synergistic integration of the

interaction dimension in JaCaMo, with the other ones. Besides classical approaches, it would be interesting to explore the use of environment abstractions also to make direct communication support more flexible by introducing, for instance, appropriate personal communication artifacts, to provide agents means for communicating using different ACLs and managing complex conversations and ontologies.

- In JaCaMo, it would be possible to further explore the O-E (Organization-Environment) connection to implement more advanced institutional mechanisms such as the *count-as* relation [Sea69], thus providing, generally speaking, a direct semantic link between the execution of actions (operations) on environment artifacts and their meaning and effect at the organizational level. Initial investigations in this direction can be found in [PRBH09].
- There is a group of future work concerning both JaCa and JaCaMo:
  - Improving the integration with the object-oriented layer on the agent side, either by enriching the set of current ***Jason*** internal actions available to this end, or defining some sort of *"smart"* and automated translation mechanism for moving from an object-based representation to a logical one, and vice-versa.
  - The implementation (in case of JaCaMo) and improvement (in the case of JaCa) of integrated development environments that would facilitate the process of design, development, and execution of JaCa and JaCaMo applications, potentially reusing and integrating existing ***Jason***, CArtAgO, and *M* OISE tools and technologies.
  - Continuing the concrete evaluation of the two development platforms through the realization of new applications in relevant domains.

The second line concerns simpAL, its ecosystem, and more in general the overall investigation of the adoption of agent-oriented programming as a general purpose paradigm for developing concurrent and distributed software systems:

- A first main future contribution would address the formalization of the operational semantics of the language through the study of both proper formal models and a core calculus to define the semantics of the main programming constructs in a rigorous manner, and evaluate the main properties (and problems) of the language. To this end, these previous works [DGRV09, DGRV12] could be a good source of inspiration to start such investigations.
- A group of future work strictly related to the formalization of the language, concerns the formalization of the simpAL type system and in turn, once such a formalization has been realized, the finalization of all those aspects not fully addressed yet in the language (sub-typing relations, inheritance, etc.). We are confident that the injection of these new aspects should be carried out quite seamlessly, by exploiting the backbone of the simpAL programming language realized in the context of this thesis.

- A main future (and already ongoing) investigation would be the study of an extension of the current task model, in order to support also cooperative tasks besides individual ones. Such an extension is fundamental in oder to give developers proper means to program cooperative activities and also model the set of possible interactions among the different agents that take part in such activities. For the study of this important extension, the work concerning agent communication protocols [Sin11, CS11] and the recent studies on session types [HYC08, GVR03, DCMYD06] could be both an important background and source of inspiration.

- It would be possible to study optimizations of the language (at least) along the following main directions:

  - Minimize the time required to execute one sense-plan-act cycle. Given the centrality of the agent control loop – it governs entirely agent execution and computation – this is probably the most important optimization we have to work toward.
  - Optimize the process of action selection, avoiding as much as possible to do unnecessary stages of the sense-plan-act cycle. Indeed, in all those cases in which there are only actions that need to be executed in sequence – i.e., there are not rules with the event or the condition part (or with both) specified in the stack's active blocks – the execution of a full sense-plan-act cycle is a unnecessary overhead: the next actions to be chosen will always be the next ones in the sequences considered. Hence, such cases could be detected statically at compile time, giving the opportunity to generate optimized compiled code that could become, eventually, as efficient as the execution of message handlers in Java-based actor-oriented technologies.
  - The study and the implementation of a strategy allowing the retrieval of beliefs, observable properties and artifact private variables on the basis of an efficient index-based mechanism.

- It would be worth to continue with the concrete evaluation of the simpAL programming approach, by applying it for realizing applications in other relevant domains. It would be particularly interesting to stress its application in those domains in which we already exploited JaCa and JaCaMo, to investigate and compare the main benefits and drawbacks of the two approaches. To this end, we are currently working on a simpAL extension targeting the programming of Rich Internet Applications.

- To better stress and evaluate the performance of the language, a more advanced set of tests and performance analysis should be carried out. Current tests have mainly focused on the evaluation of the most critical aspect for the efficient execution of simpAL programs – i.e., the impact of the agent control architecture that governs agents execution – however, other kinds of tests are required as well, in order to evaluate other important aspects of the language (e.g., the efficiency of the logic level of concurrency implemented in simpAL, in particular w.r.t. the one available in actor-oriented languages and frameworks).

- Another main group of future work concerns improving the exploitation of the underlying object-oriented level in simpAL, in particular: *(i)* the engineering of a programming support for defining arrays and collections of agents and artifacts, at the right conceptual level, without resorting to low level implementation mechanisms; *(ii)* the engineering of a programming support for the *"smart"* definition of arrays and collections of observable properties in artifacts, enabling the generation of observable properties update events related to a single element of an array/collection (currently this is not possible Section 7.7.2); and *(iii)* the introduction of the support for generics.

- Improvement to the current implementation of the simpAL distributed runtime infrastructure, to make it more robust and mature for the execution and management of distributed applications. The seamless management of runtime faults – e.g., network delays, unexpected shutdown of network nodes, etc. – is a key issue in the general context of distributed systems and related infrastructures that support their execution. So far, in simpAL the only support provided for dealing with faults at runtime is at the programming level, where programmers can deal with the different kinds of network problems that can occur by properly reacting to action failures perceived by the agents. Future work would aim at improving the current basic support for handling faults at runtime, trying to shield programmers as much as possible from their management. The final objective of this enhancement would be the realization of a fault-tolerant runtime infrastructure, as robust as the ones available in reference actor-oriented frameworks and languages (e.g., akka and Erlang), able to manage in a quite seamless manner: temporary nodes unreachability, dynamic addition / shutdown of simpAL nodes, migration of workspaces, etc.

- Improvement to the current version of the simpAL IDE. A first improvement would be the integration of our custom debugger, which allows to inspect both local and remote agents/artifacts in execution, with the powerful Eclipse debugging framework, possibly extending it for supporting the debugging of distributed applications. A second improvement would be the enrichment of the current set of features provided by the IDE. E.g., enriching the current simpAL perspective with a proper view to show, besides file system content, the logical topology of a simpAL application in terms of workspaces and agents/artifacts contained in such workspaces; introducing the means to provide a full control over the life-cycle of (possibly distributed) simpAL architectural nodes—i.e., starting/terminating the simpAL kernel in such nodes directly from the IDE.

- Finally, we believe that this thesis could be a good source of inspiration for both those researchers and practitioners that, at least in part, share our vision and main research objectives; and for those people that reading it have become convinced, or at least intrigued, in the exploitation of a programming paradigm rooted on an agent-oriented level of abstraction for tackling the challenges and complexities introduced by *"the free lunch is over call"* (Chapter 1).

Part V

Appendix

A

EBNF Grammar of the simpAL Language

```
/*=============================================================================================*/
/* Top rule: manages simpaL compilation units                                                  */
/*=============================================================================================*/

CompilationUnit = (OrgModel | Org | UsageInterface | ArtifactTemplate | RoleDef
                   |  AgentScript | LaunchConfig) ;

/*=============================================================================================*/
/* OrgModel Rules                                                                              */
/*=============================================================================================*/

OrgModel = "org-model" ID "{" ("workspace" WorkspaceDecl)+ "}" ;

WorkspaceDecl = ID "{" (ArtifactOrAgentDecl)* "}" ;

ArtifactOrAgentDecl = ID ":" ID ";" ;

/*=============================================================================================*/
/* Org Rules                                                                                   */
/*=============================================================================================*/

Org = "org" ID "implements" ID "{" ("workspace" WorkspaceInstance)+ "}" ;

WorkspaceInstance = ID "{" (ArtifactOrAgentInstance)* "}" ;

ArtifactOrAgentInstance = ID "=" ID SimpalActualParameters
                            ("init-task:" SimpalType SimpalActualParameters)? ;

/*=============================================================================================*/
/* Role Rules                                                                                  */
/*=============================================================================================*/

RoleDef = "role" ID  "{" (BeliefDef | TaskDef)* "}" ;

TaskDef = "task" ID "{" ("input-params" "{" TaskFormalParameters "}")?
                        ("output-params" "{" TaskFormalParameters "}")?
                        ("undestands" "{" TaskFormalParameters "}")?
                        ("talks-about" "{" TaskFormalParameters "}")?
          "}" ;

TaskFormalParameters = (ID ":" SimpalType ";")* ;
```

```
/*=============================================================================================*/
/* Agent Script Rules                                                                          */
/*=============================================================================================*/

AgentScript = "agent-script" ID "implements" ID ("in" ID)? "{"
                    ((BeliefDef) | TaskDef | PlanDef)*
              "}";

BeliefDef = ID ":" SimpalType ("=" Expression)? ";"? ;

PlanDef = ("plan-for" | "plan" ID "task:") SimpalType
          ("context:" context=Expression)?
          ActionRuleBlock ;

ActionRuleBlock = "=>"? "{"
                            (ActionBlockAttribute)*
                            (ActionRuleDef | BeliefDef)*
                        "}" ;

ActionBlockAttribute = (
                          | ("#completed-when:" Expression)
                          | ("#atomic")
                          | ("#hard-block")
                          | ("#soft-block")
                          | ("#to-be-repeated")
                          | ("#to-be-rep-until:" Expression)
                          | ("#using:" Expression ("," Expression)*)
                       ) ;

ActionRuleDef = (
                   ("when" | "every-time" ) EventTemplate (":" Expression)? "=>"
                )?
                AgentAction ";"? ;

EventTemplate = (
                  |  ("done" IdentifierExpr)
                  |  ("failed" IdentifierExpr)
                  |  ("assigned" IdentifierExpr)
                  |  ("told" Expression)
                  |  ("changed" Expression  ("as:" Expression))?
                );

AgentAction = (
                    IfAction
                |   WhileAction
                |   TaskManagementAction
                |   PredefinedEnvAction
                |   EnvironmentAction
                |   AssignAction
                |   JavaAction
                |   ActionRuleBlock
              )
             (LabelActionAttribute)? ;

LabelActionAttribute = "#act:" ID ;

IfAction = "if" ParExpression ActionRuleBlock ("else-if" ParExpression ActionRuleBlock)*
                                              ("else" ActionRuleBlock)? ;

WhileAction = "while" ParExpression ActionRuleBlock ;
```

```
TaskManagementAction = (
                            AssignTaskAction
                          | DropTaskAction
                          | SuspendTaskAction
                          | ResumeTaskAction
                          | DoTaskAction
                          | DropAllTasksAction
                          | SuspendAllTasksAction
                          | ResumeAllTasksAction
                          | ForgetOldPlansAction
                        ) ;

AssignTaskAction = "assign-task" ("to:" Expression)? ;

DropTaskAction =  "drop-task" IdentifierExpr ;

SuspendTaskAction = "suspend-task" IdentifierExpr ;

ResumeTaskAction =  "resume-task" IdentifierExpr ;

DoTaskAction = "do-task" (IdentifierExpr | NewTaskExpr) ("task-recipient:" Expression)? ;

DropAllTasksAction = "drop-all-tasks" ;

SuspendAllTasksAction = "suspend-all-tasks" ;

ResumeAllTasksAction = "resume-all-tasks" ;

ForgetOldPlansAction = "forget-old-plans" ;

PredefinedEnvAction = (NewArtifactAction | DisposeArtifactAction
                        | SpawnAgentAction | TellAction) ;

NewArtifactAction = "new-artifact" ID SimpalActualParameters ("in" ID)? "ref:" Expression ;

DisposeArtifactAction = "dispose-artifact" Expression;

SpawnAgentAction = "new-agent" ID ("in" ID)? ("init-task:" Expression)? "ref:" Expression? ;

TellAction = "tell" (Expression "=" Expression) ;

EnvironmentAction = ID SimpalActualParameters ("on" res=Expression)? ;

JavaAction = Expression;

AssignAction = Expression "=" Expression ;

/*=============================================================================*/
/* Launch Configuration Rules                                                  */
/*=============================================================================*/

LaunchConfig = ("org" ID)
               ("org-id" orgId = ID)?
               "workspace-addresses" "{" (WspAddress)+ "}" ;

WspAddress = ID "=" INT_NUM "." INT_NUM "."
                    INT_NUM "." INT_NUM ":" INT_NUM ;

/*=============================================================================*/
/* Expression Rules: these rules are shared among agents and artifacts rules   */
/*=============================================================================*/
```

```
ParExpression = "(" Expression ")" ;

Expression = ConditionalAndExpr ("||" ConditionalAndExpr)* ;

ConditionalAndExpr = OrExpr ("&&" OrExpr)* ;

OrExpr = ExOrExpr ("|" ExOrExpr)* ;

ExOrExpr = AndExpr ("^" AndExpr)* ;

AndExpr = EqualityExpr ("&" EqualityExpr)* ;

EqualityExpr = InstanceOfExpr   ( ("=="|"!=") InstanceOfExpr)? ;

InstanceOfExpr = RelationalExpr ("instanceof" SimpalTypeReference)? ;

RelationalExpr = ShiftExpr (("<" | "<=" | ">" | ">=") ShiftExpr)* ;

ShiftExpr = AdditiveExpr (("<<"|">>>"|">>") AdditiveExpr)* ;

AdditiveExpr = MultiplicativeExpr (("+"|"-") MultiplicativeExpr)* ;

MultiplicativeExpr = UnaryExpr (("*"|"/"|"%") UnaryExpr)* ;

UnaryExpr = (
                 (("+"|"-") UnaryExpr)
                          |  UnaryExprNotPlusMinus
            ) ;

UnaryExprNotPlusMinus = (
                            CastExpr
                          |  CompositePrimary ("++"|"--")?
                        ) ;

CastExpr = "(" SimpalType ")" UnaryExprNotPlusMinus ;

CompositePrimary = Primary (
                               ("." ID (JavaActualParameters)?)
                             |  ("[" Expression "]")
                             |  ("@" ID)
                             |  ("in" CompositePrimary)
                           )* ;
Primary = (
               ParExpression
             |  Creator
             |  IdentifierExpr
             |  Literal
             |  ("is-defined" IdentifierExpr  ("." ID)?)
             |  ("is-doing-any" IdentifierExpr  ("." ID)?)
             |  ("is-done" IdentifierExpr  ("." ID)?)
             |  ("is-ongoing" IdentifierExpr  ("." ID)?)
             |  ("is-todo" IdentifierExpr  ("." ID)?)
             |  ("is-failed" IdentifierExpr  ("." ID)?)
          ) ;

Creator = (
               ObjCreator
             |  TaskCreator
             |  ArrayCreator
          ) ;

ObjCreator = "new-object" SimpalType JavaActualParameters ;
```

```
TaskCreator = "new-task" SimpalType SimpalActualParameters ;

ArrayCreator = "new-array" SimpalType ("[" Expression "]")* ;

IdentifierExpr = (PredefinedLiterals | ID) ;

PredefinedLiterals = (
                          "this-task"
                        | "script"
                        | "plan"
                        | "agent"
                        | "artifact"
                        | "params"
                        | "op"
                      ) ;

SimpalType = ((ID ("." ID)*)|PrimitiveType) ("[" "]")* ;

PrimitiveType = (
                       "boolean"
                     | "byte"
                     | "short"
                     | "int"
                     | "long"
                     | "float"
                     | "double"
                   ) ;

Literal = (
                  INT_LITERAL
                | LONG_LITERAL
                | FLOAT_LITERAL
                | DOUBLE_LITERAL
                | CHAR_LITERAL
                | STRING_LITERAL
                | BOOLEAN_LITERAL
                | NULL_LITERAL
                | WS_LITERAL /* Whitespace */
                | SL_COMMENT /* Single-line comment */
                | ML_COMMENT /* Multi-line comment */
              ) ;

ID = ("a".."z"|"A".."Z"|"_") ("a".."z"|"A".."Z"|"_"|"0".."9")* ;

/*=============================================================================*/
/* Usage Interface Rules                                                       */
/*=============================================================================*/

UsageInterface = "usage-interface" ID "{" ((ObsPropDef | OperationDecl )";")* "}" ;

ObsPropDef = "obs-prop" ID ":" SimpalType ;

OperationDecl = "operation" ID FormalParameters ;

/*=============================================================================*/
/* Artifact Template Rules                                                     */
/*=============================================================================*/

ArtifactTemplate = "artifact" ID "implements" ID
                       "{" InitDef ((VarDef | OperationDef)";")* "}" ;
```

```
InitDef =  "init" FormalParameters OpBlock ;

OperationDef = "operation" ID FormalParameters OpBlock ;

OpBlock = "{" ((EnvStatement | (VarDef";")))* "}" ;

VarDef = ID ":" SimpalType ("=" Expression)? ;

EnvStatement = (IfStat | WhileStat | AwaitStat | AwaitTimeStat
               FailStat | JavaStat| AssignStat | ForceObsEvent) ;

IfStat = "if" ParExpression OpBlock ("elseif" ParExpression OpBlock)* ("else" OpBlock)? ;

WhileStat =  "while" ParExpression OpBlock ;

AwaitStat = "await" Expression ";" ;

AwaitTimeStat = "await-time" Expression ";" ;

FailStat = "fail" ID SimpalActualParameters ";" ;

JavaStat = Expression ";" ;

AssignStat = Expression "=" Expression ";" ;

ForceObsEvent = "generateEvForObsProp" Expression ";" ;

/*=============================================================================*/
/* Actual/Formal Param Rules: these rules are shared among agents and artficats rules */
/*=============================================================================*/

JavaActualParameters = "(" (Expression ("," Expression)*)? ")" ;

SimpalActualParameters = "(" (SimpalActualParameter ("," SimpalActualParameter)* )? ")" ;

SimpalActualParameter : ID ":" Expression ;

FormalParameters = "(" (FormalParameter ("," FormalParameter)*)? ")" ;

FormalParameter = ID ":" SimpalType ("#out")? ;
```

B
Additional Sources

B.1 Reactive File Searcher Script

Implementation of the ReactiveSearcherScript in the context of the reactive file searcher example. This is the full source of the pseudo-implementation reported in Section 7.4.1.

```
1  agent-script ReactiveSearcherScript implements ReactiveSearcher in FileSearcherOrgModel{
2
3    filesFound: java.util.List = new java.util.ArrayList()
4    currThr: long
5
6    plan-for SearchFiles {
7      #using: console@main
8      #completed-when: is-done printRes
9
10     currThr = this-task.thr;
11     searchTask: SearchFilesInDir = new-task SearchFilesInDir(dir: this-task.dir);
12     assign-task searchTask;
13
14     /* Printing results when the searchTask is done*/
15     when is-done searchTask => {
16       println(msg: "Found " + filesFound.size() + " files with the desired size");
17       i: int = 0;
18       {
19         #to-be-rep-until: (i > filesFound.size()-1)
20         println(msg: " - " + ((java.io.File)filesFound.get(i)).getName());
21         i++
22       };
23     }#act: printRes
24
25     /* Reacts to threshold changes */
26     every-time told this-task.newThr => {
27       #atomic
28
29       newThr: int = this-task.newThr;
30       if (newThr > currThr){
31         /* We can filter the current results accordingly to the new threshold */
32         j: int=0;
33         {
34           #to-be-rep-until: (j> filesFound.size()-1)
35           currFile: java.io.File;
36           currFile = (java.io.File)filesFound.get(j);
37           if (currFile.length() < newThr){
38             /* The file is not ok considering the new threshold, we remove it */
39             filesFound.remove(j)
```

```
40            } else {
41              /* The file is ok also considering the new threshold, inspect the next one*/
42              j++
43            }
44          };
45          /* Set the currThr to the new one */
46          currThr = newThr
47        } else {
48          /* The threshold has been lowered, we have to restart from scratch */
49          /* The ongoing task is dropped */
50          drop-task searchTask;
51          /* The list of found files so far is cleared */
52          filesFound.clear();
53          searchTask = new-task SearchFilesInDir(dir: this-task.dir);
54          /* Set the currThr to the new one */
55          currThr = newThr;
56          assign-task searchTask
57        }
58      }
59    }
60
61    plan-for SearchFilesInDir {
62      #using: console@main
63      currFile: java.io.File;
64      files: java.util.List;
65      currDir: java.io.File = new java.io.File(this-task.dir);
66      if (currDir.isDirectory()){
67        files = java.util.Arrays.asList(currDir.listFiles());
68        i: int = 0;
69        {
70          #to-be-rep-until: (i >files.size()-1)
71          currFile = (java.io.File)files.get(i);
72          if (currFile.isDirectory()) {
73            /* New sub-dir, we span a new sub-task for searching the files recursively */
74            do-task new-task SearchFilesInDir(dir: currFile.getPath())
75          } else-if (currFile.length() > currThr) {
76            #atomic
77            filesFound.add(currFile)
78          };
79          i++
80        }
81      }
82    }
83
84    task SearchFilesInDir{
85      input-params{
86        dir: String;
87      }
88    }
89  }
```

B.2 Sources of the Test Programs

```
1  test_actor(N) ->
2    self() ! doTaskT,
3    loop(N,0,0).
4
5  loop(0,C,R) ->
6    trigger ! testDone;
7
8  loop(N,C,R) ->
9    receive
10     doTaskT ->
11       C1 = ta(C),
12       self() ! doingTb,
13       loop(N,C1,R);
14     doingTb ->
15       C1 = tb(C),
16       self() ! doingTc,
17       loop(N,C1,R);
18     doingTc ->
19       C1 = tc(C),
20       self() ! doTaskT,
21       loop(N-1,C1,R);
22     react ->
23       loop(N,C,R+1)
24   end.
25
26 ta(C) -> C+1.
27 tb(C) -> C+1.
28 tc(C) -> C+1.
29
30 trigger(Who,0) ->
31   receive
32     testDone ->
33       erlang:halt()
34   end;
35
36 trigger(Who,N) ->
37   Who ! react,
38   trigger(Who,N-1).
39
40 start() ->
41   PID = spawn(testloop, test_actor, [10000000]),
42   register(trigger, spawn(testloop, trigger, [PID, 10000000])).
```

Figure B.1: Implementation of the first version (V1) of the first test program in Erlang. The test_actor function has been used to implement the actor that has in charge the execution of the task T, while the trigger function has been used to implement the actor that sends the react messages.

```
controller(N, ActA, ActB, ActC) ->
  self() ! doTaskT,
  loop(N, 0, 0, ActA, ActB, ActC).

loop(0, _, 10000000, _, _, _) ->
  trigger ! testDone;

loop(N, C, R, ActA, ActB, ActC) ->
  receive
    doTaskT ->
      ActA ! {doTa, C, self()},
      loop(N, C, R, ActA, ActB, ActC);
    {doneTa, C1} ->
      ActB ! {doTb, C1, self()},
      loop(N, C1, R, ActA, ActB, ActC);
    {doneTb, C1} ->
      ActC ! {doTc, C1, self()},
      loop(N, C1, R, ActA, ActB, ActC);
    {doneTc, C1} ->
      self() ! doTaskT,
      loop(N-1, C1, R, ActA, ActB, ActC);
    react ->
      loop(N, C+1, R+1, ActA, ActB, ActC)
  end.

actorA() ->
  receive
    {doTa, C, Controller} -> Controller ! {doneTa, C + 1}, actorA()
  end.

actorB() ->
  receive
    {doTb, C, Controller} -> Controller ! {doneTb, C + 1}, actorB()
  end.

actorC() ->
  receive
    {doTc, C, Controller} -> Controller ! {doneTc, C + 1}, actorC()
  end.

trigger(_, 0) ->
  receive
    testDone ->
      erlang:halt()
  end;

trigger(Who, N) ->
  Who ! react,
  trigger(Who, N-1).

start() ->
  PIDA = spawn(testloop, actorA, []), PIDB = spawn(testloop, actorB, []),
  PIDC = spawn(testloop, actorC, []),
  CONTR = spawn(testloop, controller, [10000000, PIDA, PIDB, PIDC]),
  register(trigger, spawn(testloop, trigger, [CONTR, 10000000])).
```

Figure B.2: Implementation of the second version (V2) of the first test program in Erlang in which a network of cooperative actors is used to obtain the same level of reactivity available in agent-oriented implementations of the test.

```
public class TestActor extends Actor {
  private int c = 0; private int nTimes = 0; private int maxTimes;
  private int nReactions=0; private ActorName triggerActor;

  @message
  public void start(ActorName triggerActor, Integer maxTimes) throws RemoteCodeException {
    this.triggerActor = triggerActor; this.maxTimes = maxTimes;
    send(self(), "doTaskT");
  }
  @message
  public void doTaskT() throws RemoteCodeException {
    send(self(), "doingTa");
  }
  @message
  public void doingTa() throws RemoteCodeException {
    send(self(), "doingTb");
    ta();
  }
  @message
  public void doingTb() throws RemoteCodeException {
    send(self(), "doingTc");
    tb();
  }
  @message
  public void doingTc() throws RemoteCodeException {
    tc();
    nTimes++;
    if (nTimes < maxTimes){
      send(self(),"doingTa");
    } else {
      send(triggerActor, "testDone");
    }
  }
  @message
  public void react() throws RemoteCodeException {
    nReactions = nReactions + 1;
  }
  private void ta(){c = c + 1;}
  private void tb(){c = c + 1;}
  private void tc(){c = c + 1;}
}
```

```
public class TriggerActor extends Actor {

  @message
  public void start(ActorName testActor, Integer maxTimes) throws RemoteCodeException {
    for (int i=0; i < maxTimes; i++){
      send(testActor,"react");
    }
  }

  @message
  public void testDone() throws RemoteCodeException {
    exit()
  }
}
```

Figure B.3: Implementation of the first version (V1) of the first test in ActorFoundry. (*top*) Implementation of the TestActor that has in charge the execution of the task T. (*bottom*) Implementation of the TriggerActor that has in charge the sending of react messages to the TestActor.

```
public class TestActor extends Actor {
  private int nTimes = 0;       private int maxTimes;
  private int nReactions=0; private ActorName triggerActor;
  private ActorName actorA; private ActorName actorB; private ActorName actorC;

  @message
  public void start(ActorName triggerActor, ActorName actorA, ActorName actorB,
   ActorName actorC, Integer maxTimes) throws RemoteCodeException {
    this.actorA = actorA; this.actorB = actorB; this.actorC = actorC;
    this.triggerActor = triggerActor; this.maxTimes = maxTimes; send(self(), "doTaskT", 0);
  }

  @message
  public void doTaskT(Integer c) throws RemoteCodeException { send(actorA, "doTa", c); }
  @message
  public void doneTa(Integer c) throws RemoteCodeException { send(actorB, "doTb", c); }
  @message
  public void doneTb(Integer c) throws RemoteCodeException { send(actorC, "doTc", c); }

  @message
  public void doneTc(Integer c) throws RemoteCodeException {
    nTimes++;
    if (nTimes < maxTimes){
      send(self(), "doTaskT", c);
    } else {
      send(triggerActor, "testDone");
    }
  }

  @message
  public void react() throws RemoteCodeException {
    nReactions = nReactions + 1;
  }
}
```

```
public class TriggerActor extends Actor {
  @message
  public void start(ActorName testActor, Integer maxTimes) throws RemoteCodeException {
    for (int i=0; i < maxTimes; i++){ send(testActor,"react"); }
  }
  @message
  public void testDone() throws RemoteCodeException {
    exit()
  }
}
```

```
public class ActorA extends Actor {
  private ActorName controller;
  public ActorA(ActorName controller){ this.controller = controller; }
  @message
  public void doTa(Integer c) throws Exception { send(controller, "doneTa", c+1); }
}
```

Figure B.4: Implementation of the second version (V2) of the first test in ActorFoundry in which a network of cooperative actors is used to obtain the same level of reactivity available in agent-oriented implementations of the test. The missing implementation of ActorB and ActorC can be easily deduced from the source code of ActorA.

```
c(0).
reactions(0).

!loop(10000000).

+!loop(0) : true
 <- .send(trigger, tell, done).

+!loop(N)
 <- !tA;
    !tB;
    !tC;
    !!loop(N-1).

+!tA:c(Val)<--+c(Val+1).
+!tB:c(Val)<--+c(Val+1).
+!tC:c(Val)<--+c(Val+1).

@reactplan[atomic]
+react(N) : reactions(Val)
        <-  -+reactions(Val+1).
```

```
!loop(10000000).

+!loop(0).

+!loop(N)
 <- .send(test, tell, react(N));
    !!loop(N-1).

+done : true
 <- .stopMAS.
```

Figure B.5: Implementation of the first test in ***Jason*** in which is used the `tell` performative. (*left*) The test agent that has in charge the execution of the task T. (*right*) Implementation of the trigger agent that sends the react messages to the test agent.

```
c(0).
reactions(0).

!loop(10000000).

+!loop(0) : true
 <- .send(trigger, tell, done).

+!loop(N)
 <- !tA;
    !tB;
    !tC;
    !!loop(N-1).

+!tA:c(Val)<--+c(Val+1).
+!tB:c(Val)<--+c(Val+1).
+!tC:c(Val)<--+c(Val+1).

@reactplan[atomic]
+!react : reactions(Val)
        <-  -+reactions(Val+1).
```

```
!loop(10000000).

+!loop(0).

+!loop(N)
 <- .send(test, achieve, react);
    !!loop(N-1).

+done : true
 <- .stopMAS.
```

Figure B.6: Implementation of the first test in ***Jason*** in which is used the `achieve` performative. (*left*) The test agent that has in charge the execution of the task T. (*right*) Implementation of the trigger agent that sends the react messages to the test agent.

```
role RoleRLoop {
  task TaskT {
    input-params{
      maxTimes: int;
    }
    understands {
      react: boolean;
    }
  }
}
```

```
agent-script TestAgentScriptLoop implements RoleRLoop in agere12simpalOrgModel {

  c: int = 0
  maxTimes: int = 0

  plan-for TaskT {
    #using: console@main
    #completed-when: is-done trep

    nReactions: int = 0
    script.maxTimes = this-task.maxTimes;
    do-task new-task Trep() #act: trep

    every-time told this-task.react => {
      #atomic
      nReactions = nReactions + 1
    }
  }

  plan-for Trep {
    nTimes: int = 0
    while (nTimes < script.maxTimes) {
      do-task new-task Ta();
      do-task new-task Tb();
      do-task new-task Tc();
      nTimes = (nTimes + 1)
    }
  }

  plan-for Ta { c = c + 1 }
  plan-for Tb { c = c + 1 }
  plan-for Tc { c = c + 1 }

  task Ta {}
  task Tb {}
  task Tc {}
  task Trep {}
}
```

Figure B.7: Implementation of the first test in simpAL. Role and script of the agent in charge the execution of the task T in the context of the performance test.

```
role TriggerAgent {
  task TriggerTest {
    input-params{
      maxTimes: int;
    }
  }
}
```

```
agent-script TriggerAgentScriptLoop implements TriggerAgent in agere12simpalOrgModel {

  nTimes: int = 0
  finished: boolean = false
  testAgent: RoleRLoop
  taskT: RoleRLoop.TaskT

  plan-for TriggerTest {
    #completed-when: finished
    #using: console@main

    new-agent TestAgentScriptLoop() ref: testAgent;
    taskT = new-task RoleRLoop.TaskT(maxTimes: this-task.maxTimes);
    assign-task taskT to: testAgent;
    while (nTimes < this-task.maxTimes) {
      tell taskT.react = true;
      nTimes++
    }

    when is-done taskT => {
      finished = true
    } #act: doneT
  }
}
```

Figure B.8: Implementation of the first test in simpAL. Role and script of the trigger agent that has in charge the sending of react messages to the agent playing the role RoleRLoop in the context of the performance test.

```
start(Token) ->
    H = lists:foldl(
      fun(Id, Pid) -> spawn(threadring, roundtrip, [Id, Pid]) end,
      self(),
      lists:seq(?RING, 2, -1)),
    H ! Token,
    roundtrip(1, H).

roundtrip(Id, Pid) ->
  receive
    1 ->
      erlang:halt();
    Token ->
      Pid ! Token - 1,
      roundtrip(Id, Pid)
  end.

main() -> start(10000000).
```

Figure B.9: Implementation of the second test (the thred-ring test) in Erlang.

```
public class Worker extends Actor {

  private ActorName next;

  public Worker(){ }

  public Worker(Integer id, ActorName next){
    this.next = next;
  }

  @message
  public void start(Integer token, ActorName next) throws Exception {
    this.next = next;
    send(next, "PassToken", token-1);
  }

  @message
  public void PassToken(Integer token) throws Exception {
    if (token==0){
      System.out.exit(0);
    } else {
      send(next, "PassToken", token-1);
    }
  }
}
```

```
public class Initializator extends Actor {
  @message
  public void test() throws Exception {
    int nTimes = 10000000;
    int nWorkers = 503;
    ActorName firstWorker = create(Worker.class);
    ActorName nextWorker = firstWorker;
    for (int i=0;i<nWorkers-1;i++){
      System.currentTimeMillis();
      nextWorker = create(Worker.class, nextWorker);
    }
    send(firstWorker,"start", new Integer(nTimes), nextWorker);
  }
}
```

Figure B.10: Implementation of the second test (the thred-ring test) in ActorFoundry.

```
!init.

+!init :  .my_name(M) & .delete("thread", M, NS) & .term2string(N,NS) &
          Y = N mod 503 + 1 &
          .concat("thread",Y,X) <- +next(X);
          if (.my_name(thread503)) { .send(thread1, tell, token(1000000)) }.

+token(0) <- .stopMAS.

+token(N) : next(X) <- -token(N); .send(X, tell, token(N-1)).
```

```
!init.

+!init :  .my_name(M) & .delete("thread", M, NS) & .term2string(N,NS) &
          Y = N mod 503 + 1 &
          .concat("thread",Y,X) <- +next(X);
          if (.my_name(thread503)) { .send(thread1, achieve, token(10000)) }.

+!token(0) <- .stopMAS.

+!token(N) : next(X) <- .send(X, achieve, token(N-1)).
```

Figure B.11: Implementation of the second test (the thread-ring test) in ***Jason*** in which is used the `tell` performative (*top*). Implementation of the second test (the thred-ring test) in ***Jason*** in which is used the `achieve` performative (*bottom*).

```
role Initiator {
  task Setup {
    input-params{
      numAgents: int;
      nPasses: int;
    }
  }
  task PassToken {
    input-params {
      agentId: int ;
    }
    understands{
      token: int;
    }
  }
}
```

```
agent-script InitializatorScript implements Initializator in ThreadRingOrgModel {

  next: Worker
  totalPassToDo: int
  passTokenTask: Worker.PassToken

  plan-for Setup {
    #using: console@main
    totalPassToDo = nPasses;
    passTokenTask = new-task Worker.PassToken(agentId:this-task.numAgents-1);
    /* creating first worker */
    new-agent WorkerScript ref: next;
    /* task assignment to first worker */
    assign-task passTokenTask to: next
  }

  plan-for PassToken {
    #using: console@main
    /* start passing the token */
    tell passTokenTask.token=totalPassToDo-1;
    {
      every-time told this-task.token : this-task.token !=0 => {
        /* the token has come back, if not = 0 we go for another round */
        tell passTokenTask.token = this-task.token - 1
      }
      when told this-task.token : this-task.token ==0 => {
        println(msg: "[Initializator+]I was the last");
        exit()
      }
    }
  }
}
```

Figure B.12: Implementation in simpAL of the role and script of the initiator agent used in the second test (the thread-ring test).

```
role Worker {
  task PassToken {
    input-params {
      agentId: int ;
    }
    understands{
      token: int ;
    }
  }
}
```

```
agent-script WorkerScript implements Worker in ThreadRingOrgModel {

  next: Worker
  myId: int
  plan-for PassToken {
    #using: console
    if (agentId!=1){
      passTokenTask: PassToken;
      passTokenTask = new-task PassToken(agentId:this-task.agentId-1);
      new-agent WorkerScript ref: next;
      assign-task passTokenTask to: next
      every-time told this-task.token : this-task.token !=0 => {
        tell passTokenTask.token = this-task.token - 1
      }
      when told this-task.token : this-task.token ==0 => {
        println(msg: "["+agentId+"]I was the last")
        exit()
      }
    } else {
      /* last worker, it needs to interact with the initializator */
      passTokenTask: Initiator.PassToken
      passTokenTask = new-task Initiator.PassToken(agentId:this-task.agentId-1);
      assign-task passTokenTask to: firstWorker
      every-time told this-task.token : this-task.token !=0 => {
        tell passTokenTask.token = this-task.token - 1
      }
      when told this-task.token : this-task.token ==0 => {
        println(msg: "["+agentId+"]I was the last")
        exit()
      }
    }
  }
}
```

Figure B.13: Implementation in simpAL of the role and script of a generic worker agent used in the second test (the thread-ring test).

Bibliography

[Age] Agent-Oriented Software Group. Official JACK Website. Online document, available at: http://www.aosgrp.com.au/ – Last Retrieved: November 25, 2012.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[Agh90] Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, September 1990.

[AH87] Gul Agha and Carl Hewitt. Concurrent programming using actors. In Akinori Yonezawa and Mario Tokoro, editors, *Object-oriented concurrent programming*, pages 37–53. MIT Press, Cambridge, MA, USA, 1987.

[Alea] Alessandro, Ricci and Andrea, Santi. Official CArtAgO Website. Online document, available at: http://cartago.sourcefoge.net/ – Last Retrieved: November 15, 2012.

[Aleb] Alexander Pokahr and Lars Braubach and Winfried Lamersdorf. Official Jadex Website. Online document, available at: http://jadex.sourceforge.net – Last Retrieved: November 15, 2012.

[Alec] Alexandru Sorici. Reference Documentation for the JaCa-Arduino Project. Online document, available at: https://docs.google.com/document/d/1uIHHOk0DCXFIqWod05VrXMRZs4IFrj3pVz1fMeIKrNY – Last Retrieved: November 28, 2012.

[All02] J. Allaire. Macromedia flash mxa next-generation rich client (white paper). Technical report, Macromedia, 2002.

[AMST97] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, January 1997.

[Anda] Andrea Santi and Marco Guidi and Alessandro Ricci. JaCa-Android official website. Online document, available at: http://jaca-android.sourceforge.net/ – Last Retrieved: November 21, 2012.

[Andb] Andrea Santi and Mattia Minotti and Alessandro Ricci. JaCa-Web official website. Online document, available at: http://jaca-web.sourceforge.net/ – Last Retrieved: November 21, 2012.

[Ard] Arduino Gruop. The Arduino Project Official Website. Online document, available at: http://www.arduino.cc/ – Last Retrieved: November 28, 2012.

[Arm10] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[AT07] Luis Antunes and Keiki Takadama, editors. *Multi-Agent-Based Simulation VII, International Workshop, MABS 2006, Hakodate, Japan, May 8, 2006, Revised and Invited Papers*, volume 4442 of *Lecture Notes in Computer Science*. Springer, 2007.

[AUS75] J.L. Austin, J.O. Urmson, and M. Sbisà. *How to Do Things with Words: The William James Lectures Delivered at Harvard University in 1955*. Harvard University. Clarendon Press, 1975.

[BA05] Mordechai Ben-Ari. *Principle of Concurrent and Distributed Programming*. Addison-Wesley Longman, 2005.

[BBD+06] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah-Seghrouchni, Jorge J. Gómez-Sanz, João Leite, Gregory M. P. O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

[BBH+11] Olivier Boissier, Rafael H. Bordini, Jomi F. Hbner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, -(-):–, 2011.

[BBM10] Matteo Baldoni, Cristina Baroglio, and Elisa Marengo. Behavior-oriented commitment-based protocols. In *ECAI: European Conference on Artificial Intelligence*, pages 137–142, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.

[BCG07] F.L. Bellifemine, G. Caire, and D. Greenwood. *Developing multi-agent systems with JADE*. Wiley series in agent technology. John Wiley, 2007.

[BDDEFS11] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Special Issue: Multi-Agent Programming*, volume 23 (2). Springer Verlag, 2011.

[BDDFS05a] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications - Vol. I*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005.

[BDDFS05b] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications (Vol. I)*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005.

[BDDFS09] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications (Vol. II)*. Springer, 2009.

[BDEFSD09] Rafael H. Bordini, Mehdi Dastani, Amal El Fallah Seghrouchni, and Jürgen Dix, editors. *Multi-Agent Programming Languages, Platforms and Applications - Vol. II*. Springer, 2009.

[BDH+01] Jan Broersen, Mehdi Dastani, Joris Hulstijn, Zisheng Huang, and Leendert van der Torre. The boid architecture - conflicts between beliefs, obligations, intentions and desires. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 9–16. ACM Press, 2001.

[BFG+90] Howard Barringer, Michael Fisher, Dov M. Gabbay, Graham Gough, and Richard Owens. Metatem: A framework for programming in temporal logic. In *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, pages 94–129, London, UK, UK, 1990. Springer-Verlag.

[BGJ11] Tibor Bosse, Armando Geller, and Catholijn M. Jonker, editors. *Multi-Agent-Based Simulation XI - International Workshop, MABS 2010, Toronto, Canada, May 11, 2010, Revised Selected Papers*, volume 6532 of *Lecture Notes in Computer Science*. Springer, 2011.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[BH06] Rafael H. Bordini and Jomi F. Hübner. Bdi agent programming in agentspeak using jason. In *Proceedings of the 6th international conference on Computational Logic in Multi-Agent Systems*, CLIMA'05, pages 143–164, Berlin, Heidelberg, 2006. Springer-Verlag.

[BHD11] Tristan M. Behrens, Koen V. Hindriks, and Jürgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61(4):261–295, April 2011.

[BHLM02] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. In Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, editors, *Readings in hardware/software co-design*, chapter Ptolemy: a framework for simulating and prototyping heterogeneous systems, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[BHS07] Olivier Boissier, Jomi F. Hübner, and Jaime S. Sichman. Organization oriented programming: from closed to open organizations. In Gregory O'Hare, Oguz Dikenelli, and Alessandro Ricci, editors, *Engineering Societies in the Agents World VII (ESAW 06)*, volume 4457 of *LNCS*, pages 86–105. Springer Berlin / Heidelberg, 2007.

[BHW07] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using **Jason***. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[BIP07] M.E. Bratman, D.J. Israel, and M.E. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355, 2007.

[BPL06] L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the capability concept for flexible bdi agent modularization. In *Programming Multi-Agent Systems*, pages 139–155. Springer, 2006.

[BPML05] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal representation for bdi agent systems. *Programming Multi-Aent Systems*, pages 44–65, 2005.

[BPR99] F. Bellifemine, A. Poggi, and G. Rimassa. JADE: A FIPA-compliant agent framework. In *Proceedings of the fourth conference on the practical application of intelligent agents and multi-agent technology*, pages 97–108, London, UK, April 1999.

[Bra87] M. Bratman. *Intention, plans, and practical reason*. Harvard University Press, 1987.

[Bri89] J.P. Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *ECOOP–European Conference on Object-Oriented Programming*, volume 89, pages 109–129, 1989.

[BS08] S. Bromuri and K. Stathis. Situating cognitive agents in golem. In *International Workshop on Engineering Environment-Mediated Multi-Agent Systems*, pages 115–134. Springer, 2008.

[Cas94] Cristiano Castelfranchi. Guarantees for autonomy in cognitive agent architecture. In *ECAI Workshop on Agent Theories, Architectures, and Languages*, pages 56–70, 1994.

[CC95] Rosaria Conte and C. Castelfranchi. *Cognitive And Social Action*. Taylor & Francis Group, 1995.

[CFNS05] Francisco Curbera, Donald F. Ferguson, Martin Nally, and Marcia L. Stockton. Toward a programming model for service-oriented computing. In *Third International Conference on Service-Oriented Computing (ICSOC-05)*, volume 3826 of *Lecture Notes in Computer Science*. Springer, 2005.

[CG98] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*, pages 140–155. Springer, 1998.

[Chr] Christian Tismer. The stackless python framework official website. Online document, available at: `http://www.stackless.com/` – Last Retrieved: December 8, 2012.

[Cli81] William D Clinger. Foundations of actor semantics. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[CM03] W.F. Clocksin and C.S. Mellish. *Programming in PROLOG*. Springer, 2003.

[CMM09] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems & Structures*, 35(1):80 – 98, 2009.

[com00] The state of the art in agent communication languages. *Knowl. Inf. Syst.*, 2(3):259–284, August 2000.

[Cor12] Microsoft Corporation. TypeScript Language Official Website, 2012. Online document, available at: `http://www.typescriptlang.org/` – Last Retrieved: November 26, 2012.

[CR08] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins: Building Commercial-Quality Plug-ins*. Addison-Wesley Professional, 3 edition, 2008.

[CS11] A.K. Chopra and M.P. Singh. Specifying and applying commitment-based business patterns. In *The 10th International Conference on Autonomous Agents and Multiagent Systems*, pages 475–482, 2011.

[Das08] Mehdi Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[DCMYD06] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *ECOOP–European Conference on Object-Oriented Programming*, pages 328–352. Springer, 2006.

[DDM04] Virginia Dignum, Frank Dignum, and John-Jules Meyer. An agent-mediated approach to the support of knowledge sharing in organizations. *Knowledge Engineering Review*, 19(2):147–174, June 2004.

[Dem95] Yves Demazeau. From interactions to collective behaviour in agent-based systems. In *Proc. of the 1st European Conf. on Cognitive Science. Saint-Malo*, pages 117–132, 1995.

[DGLL00] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169, August 2000.

[DGMT09] Mehdi Dastani, Davide Grossi, John-Jules Ch. Meyer, and Nick Tinnemeier. Knowledge representation for agents and multi-agent systems. chapter Normative Multi-agent Programs and Their Logics, pages 16–31. Springer-Verlag, Berlin, Heidelberg, 2009.

[DGRV09] Ferruccio Damiani, Paola Giannini, Alessandro Ricci, and Mirko Viroli. Featherweight agent language - a core calculus for agents and artifacts. In *ICSOFT (1)*, pages 218–225, 2009.

[DGRV12] F. Damiani, P. Giannini, A. Ricci, and M. Viroli. Standard type soundness for agents and artifacts. *Scientific Annals of Computer Science*, 22(2):267–326, 2012.

[DMS08] M. Dastani, C. Mol, and B. Steunebrink. Modularity in agent programming languages. *Intelligent Agents and Multi-Agent Systems*, pages 139–152, 2008.

[DVCM+06] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. DHondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. *ECOOP–European Conference on Object-Oriented Programming*, pages 230–254, 2006.

[DvRM05] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Programming multi-agent systems in 3apl. In *Multi-Agent Programming: Languages, Platforms and Applications (Vol. I)*, pages 39–67. 2005.

[Ecla] Eclipse Software Foundations. Eclipse Official Website. Online document, available at: `http://www.eclipse.org/` – Last Retrieved: November 28, 2012.

[Eclb] Eclipse Software Foundations. Plugin Development Environment (PDE) Official Website. Online document, available at: `http://www.eclipse.org/pde/` – Last Retrieved: November 28, 2012.

[EFSS05] A. El Fallah Seghrouchni and A. Suna. Himalaya framework: Hierarchical intelligent mobile agents for building large-scale and adaptive systems based on ambients. *Massively Multi-Agent Systems I*, pages 575–575, 2005.

[Eps11] J.M. Epstein. *Generative social science: Studies in agent-based computational modeling*. Princeton University Press, 2011.

[ERARA04] Marc Esteva, Juan A. Rodríguez-Aguilar, Bruno Rosell, and Josep L. Arcos. AMELI: An agent-based middleware for electronic institutions. In *Prog. of the 3rd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'2004)*, pages 236–243, New York, 2004. ACM.

[Eri10] Marius Eriksen. Scaling scala at twitter. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 8:1–8:1, New York, NY, USA, 2010. ACM.

[ETS10] ETSI. Machine-to-machine communications (m2m), functional architecture. 2010.

[FFMM94] T. Finin, R. Fritzson, D. McKay, and R. McEntire. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

[FG98] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agents systems. In *Proc. of the 3rd Int. Conf. on Multi-Agent Systems (ICMAS'98)*, pages 128–135. IEEE Press, 1998.

[Fir10] D. Firesmith. Profiling Systems Using the Defining Characteristics of Systems of Systems (SoS). Technical Report Technical Note CMU/SEI-2010-TN-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2010.

[Fis94] Michael Fisher. A survey of concurrent metatem - the language and its applications. In *Proceedings of the First International Conference on Temporal Logic*, ICTL '94, pages 480–505, London, UK, UK, 1994. Springer-Verlag.

[FM96] J. Ferber and J.P. Müller. Influences and reaction: a model of situated multiagent systems. In *Proceedings of Second International Conference on Multi-Agent Systems (ICMAS-96)*, pages 72–79, 1996.

[FMBB04] Jacques Ferber, Fabien Michel, and José-Antonio Báez-Barranco. AGRE: Integrating environments with organizations. In *Environments for Multi-Agent Systems (E4MAS)*, pages 48–56, 2004.

[Foua] Apache Software Foundation. Axis2 Platform Official Website. Online document, available at: `http://axis.apache.org/axis2/java/core/` – Last Retrieved: November 18, 2012.

[Foub] Foundation of Intelligent Physical Agent. Fipa abstract architecture specification. Online document, available at: `http://www.fipa.org/repository/architecturespecs.html` – Last Retrieved: November 15, 2012.

[Fouc] Foundation of Intelligent Physical Agent. Fipa agent communication language specification. Online document, available at: `http://www.fipa.org/repository/aclspecs.html` – Last Retrieved: November 15, 2012.

[Foud] Foundation of Intelligent Physical Agent. Fipa agent management specification. Online document, available at: `http://www.fipa.org/repository/managementspecs..html` – Last Retrieved: November 15, 2012.

[Foue] Foundation of Intelligent Physical Agent. Fipa official website. Online document, available at: `http://www.fipa.org/` – Last Retrieved: November 8, 2012.

[FRSaF10] Piero Fraternali, Gustavo Rossi, and Fernando S andnchez Figueroa. Rich internet applications. *IEEE Internet Computing*, 14(3):9 –12, may-june 2010.

[FSS03] Amal El Fallah-Seghrouchni and Alexandru Suna. Claim: A computational language for autonomous, intelligent and mobile agents. In *Programming Multi-Agent Systems*, pages 90–110, 2003.

[FSS05] Amal El Fallah-Seghrouchni and Alexandru Suna. Claim and sympa: A programming environment for intelligent and mobile agents. In *Multi-Agent Programming: Languages, Platforms and Applications (Vol. I)*, pages 95–122. 2005.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, May 2002.

[GC04] D. Greenwood and M. Calisti. Engineering web service-agent integration. In *Proc. of IEEE Conf. on Systems, Man and Cybernetics*, 2004.

[GD94] G. Nigel Gilbert and J. Doran. *Simulating societies: the computer simulation of social phenomena*. UCL Press, 1994.

[Gel85] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[GF01] Olivier Gutknecht and Jacques Ferber. The madkit agent platform architecture. In *Revised Papers from the International Workshop on Infrastructure for Multi-Agent Systems: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, pages 48–55, London, UK, UK, 2001. Springer-Verlag.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Pattens*. Addison Wesley, 1995.

[GK94] M.R. Genesereth and S.P. Ketchpel. Software agents. *Communication of ACM*, 37(7):48–53, 1994.

[Gooa] Google Inc. Android Official Website. Online document, available at: `http://code.google.com/android` – Last Retrieved: November 18, 2012.

[Goob] Google Inc. Android Services Reference Documentation. Online document, available at: `http://developer.android.com/guide/components/services.html` – Last Retrieved: November 18, 2012.

[Gooc] Google Inc. Google Web Toolkit Official Website. Online document, available at: `http://code.google.com/webtoolkit/` – Last Retrieved: November 25, 2012.

[Good] Google Inc. The Looper API. Online document, available at: `http://developer.android.com/reference/android/os/Looper.html` – Last Retrieved: December 2, 2012.

[GPP+99] Michael P. Georgeff, Barney Pell, Martha E. Pollack, Milind Tambe, and Michael Wooldridge. The belief-desire-intention model of agency. In *Proceedings of the 5th International Workshop on Intelligent Agents, Agent Theories, Architectures, and Languages*, ATAL '98, pages 1–10, London, UK, UK, 1999. Springer-Verlag.

[Gup12] M.K. Gupta. *Akka Essentials*. Packt, 2012.

[GVR03] Simon Gay, Vasco T. Vasconcelos, and António Ravara. Session types for inter-process communication. TR 2003–133, Department of Computing, University of Glasgow, March 2003.

[HA79] Carl E Hewitt and Russell R. Atkinson. Specification and proof techniques for serializers. *IEEE Transactions on Software Engineering*, 5(1):10–23, 1979.

[HB77] Carl Hewitt and Henry G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977.

[HBB10] Jomi Fred Hübner, Olivier Boissier, and Rafael H. Bordini. From organisation specification to normative programming in multi-agent organisations. In *CLIMA XI*, pages 117–134, 2010.

[HBKR10] Jomi F. Hübner, Olivier Boissier, Rosine Kitio, and Alessandro Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous Agents and Multi-Agent Systems*, 20:369–400, May 2010.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[HDBVdHM99] K.V. Hindriks, F.S. De Boer, W. Van der Hoek, and J.J.C. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[Hew69] Carl Hewitt. Planner: a language for proving theorems in robots. In *Proceedings of the 1st international joint conference on Artificial intelligence*, IJCAI'69, pages 295–301, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[Hew93] G.W. Hewes. A history of speculation on the relation between tools and language. *Tools, language and cognition in human evolution*, pages 20–31, 1993.

[Hin08] K. Hindriks. Modules as policy-based intentions: Modular agent programming in goal. In *Programming Multi-Agent Systems*, pages 156–171. Springer, 2008.

[Hin09] Koen V. Hindriks. Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications (Vol. II)*, pages 3–37. Springer-Verlag, 2009.

[HL04] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316, December 2004.

[HLM94] Wiebe van der Hoek, Bernd van Linder, and John-Jules Ch. Meyer. A logic of capabilities. In *Proceedings of the Third International Symposium on Logical Foundations of Computer Science*, LFCS '94, pages 366–378, London, UK, UK, 1994. Springer-Verlag.

[HO07] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proceedings of the 9th international conference on Coordination models and languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[HO08] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

[HO09] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[HS12] P. Haller and F. Sommers. *Actors in Scala*. Artima Incorporation, 2012.

[HSB07] Jomi F. Hubner, Jaime S. Sichman, and Olivier Boissier. Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. *Int. J. Agent-Oriented Softw. Eng.*, 1(3):370–395, December 2007.

[Hüb03] J.F. Hübner. *Um modelo de reorganização de sistemas multiagentes*. PhD thesis, Universidade de ão Paulo, Escola Politcnica, 2003.

[Huh01] Michael N. Huhns. Interaction-oriented programming. In *First international workshop on Agent-oriented software engineering (AOSE 2000)*, pages 29–44, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[Huh02] Michael N. Huhns. Agents as web services. *IEEE Internet Computing*, 6(4):93–95, 2002.

[Hun06] Michael N. Hunhs. A research agenda for agent-based Service-Oriented Architectures. In Matthias Klusch, Michael Rovatsos, and Terry Payne, editors, *CIA 2006*, volume 4149 of *LNA*, pages 8–22. Springer-Verlag Berlin Heidelberg, 2006.

[HYC08] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *ACM SIGPLAN NOTICES*, 43(1):273, 2008.

[Inca] Google Inc. Dart isolates api official website. Online document, available at: `http://api.dartlang.org/docs/bleeding_edge/dart_isolate.html` – Last Retrieved: November 26, 2012.

[Incb] Google Inc. Dart language official website. Online document, available at: `http://www.dartlang.org/` – Last Retrieved: November 26, 2012.

[Incc] Google Inc. Gmail Official Website. Online document, available at: `http://mail.google.com/` – Last Retrieved: November 23, 2012.

[Incd] Google Inc. Google Maps Official Website. Online document, available at: `http://maps.google.com/` – Last Retrieved: November 25, 2012.

[Int] Internet Engineering Task Force. JavaScript Object Notation RFC. Online document, available at: `http://www.json.org/` – Last Retrieved: November 25, 2012.

[Ite] Itemis. Xtext Language Development Framework Official Website. Online document, available at: `http://www.eclipse.org/Xtext/` – Last Retrieved: November 28, 2012.

[Jac] Jacob Lee. The parley framework official website. Online document, available at: `http://osl.cs.uiuc.edu/parley/` – Last Retrieved: December 8, 2012.

[Jen01] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.

[KA] Rajesh Kumar Karmani and Gul Agha. The ActorFoundry Programming Language Official Website. Online document, available at: `http://osl.cs.uiuc.edu/af/` – Last Retrieved: December 8, 2012.

[KA11] Rajesh Kumar Karmani and Gul Agha. Actors. In *Springer's Encyclopedia of Parallel Computing*, 2011.

[Kaf90] Dennis Kafura. Act++: building a concurrent c++ with actors. *J. Object Oriented Program.*, 3(1):25–37, April 1990.

[Kam10] A. Kameas. Towards the next generation of ambient intelligent environments. In *Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), 2010 19th IEEE Int. Workshop on*, pages 1 –6, june 2010.

[Kay69] Alan Curtis Kay. *The reactive engine*. PhD thesis, The University of Utah, 1969. AAI7003806.

[Kay96] Alan C. Kay. History of programming languages—ii. chapter The early history of Smalltalk, pages 511–598. ACM, New York, NY, USA, 1996.

[Kay98] Alan Curtis Kay. An email on messaging in Smalltalk/Squeak, 1998. Online document, available at: `http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html` – Last Retrieved: November 28, 2012.

[Kim97] W. Kim. *ThAL: An actor system for efficient and scalable concurrent computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[KMS+04] Antonis Kakas, Paolo Mancarella, Fariba Sadri, Kostas Stathis, and Francesca Toni. The kgp model of agency. In *ECAI: European Conference on Artificial Intelligence*, pages 33–37. IOS Press, 2004.

[Koe] Koen V. Hindriks. Official GOAL Website. Online document, available at: `http://mmi.tudelft.nl/trac/goal` – Last Retrieved: November 15, 2012.

[KSA09] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *PPPJ–International Conference on the Principles and Practice of Programming in Java*, pages 11–20, New York, NY, USA, 2009. ACM.

[KSMA06] Y.M. Kwon, S. Sundresh, K. Mechitov, and G. Agha. Actornet: An actor platform for wireless sensor networks. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1297–1300. ACM, 2006.

[KSN00] M.T. Kone, A. Shimazu, and T. Nakajima. The state of the art in agent communication languages. *Knowledge and Information Systems*, 2(3):259–284, 2000.

[LAP01] João Alexandre Leite, José Júlio Alferes, and Luís Moniz Pereira. Multi-dimensional dynamic knowledge representation. In *Proceedings of the 6th International Conference on Logic Programming and Non-monotonic Reasoning*, LPNMR '01, pages 365–378, London, UK, UK, 2001. Springer-Verlag.

[LD12] Gary T. Leavens and Matthew B. Dwyer, editors. *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. ACM, 2012.

[LF11] Cristina Videira Lopes and Kathleen Fisher, editors. *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. ACM, 2011.

[LFP99a] Y. Labrou, T. Finin, and Y. Peng. Agent communication languages: The current landscape. *Intelligent Systems and Their Applications, IEEE*, 14(2):45–52, 1999.

[LFP99b] Yannis Labrou, Tim Finin, and Yun Peng. Agent communication languages: The current landscape. *IEEE Intelligent Systems*, 14(2):45–52, March 1999.

[Lie06] Henry Lieberman. The continuing quest for abstraction. In *ECOOP–European Conference on Object-Oriented Programming*, volume 4067/2006, pages 192–197. Springer Berlin / Heidelberg, 2006.

[LJR09] James A. Landay, Anthony D. Joseph, and Franklin Reynolds. Guest editors' introduction: Smarter phones. *IEEE Pervasive Computing*, 8:12–13, April 2009.

[LRL+97] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59–83, 1997.

[MB] Mark Miller and Dan Bornstein. The E language Official Website. Online document, available at: `http://erights.org/elang/index.html` – Last Retrieved: December 8, 2012.

[Meha] Mehdi Dastani. Official 2APL Website. Online document, available at: `http://apapl.sourceforge.net/` – Last Retrieved: November 25, 2012.

[Mehb] Mehdi Dastani and M. Birna van Riemsdijk and John-Jules Ch. Meyer. Official 3APL Website. Online document, available at: `http://www.cs.uu.nl/3apl` – Last Retrieved: November 25, 2012.

[Mica] Microsoft Corportaion. Await API Specification. Online document, available at: `http://msdn.microsoft.com/en-us/library/vstudio/hh156528.aspx` – Last Retrieved: December 8, 2012.

[Micb] Microsoft Corportaion. The axum programming language official website. Online document, available at: `http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx` – Last Retrieved: December 8, 2012.

[Mika] Mike Rettig. The jetlang framework official website. Online document, available at: `http://code.google.com/p/jetlang/` – Last Retrieved: December 8, 2012.

[Mikb] Mike Rettig. The retlang framework official website. Online document, available at: `http://code.google.com/p/retlang/` – Last Retrieved: December 8, 2012.

[Mit02] J.C. Mitchell. *Concepts in programming languages*. Cambridge University Press, 2002.

[ML10] N. Madden and B. Logan. Modularity and compositionality in jason. In *Programming Multi-Agent Systems*, pages 237–253. Springer, 2010.

[MRS11] M. Minotti, A. Ricci, and A. Santi. Exploiting agent-oriented programming for developing future internet applications based on the web: the jaca-web framework. *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, pages 76–94, 2011.

[MS04] G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274. ACM, 2004.

[MSR10] M. Minotti, A. Santi, and A. Ricci. Exploiting Agent-Oriented Programming for Building Advanced Web 2.0 Applications. In Omicini Andrea and Viroli Mirko, editors, *Proceedings of the 11th Workshop on Objects and Agents (WOA 2010)*, volume 621 of *CEUR Workshop Proceedings*. Sun SITE Central Europe, RWTH Aachen University, 2010.

[MTS05] Mark Miller, E. Tribble, and Jonathan Shapiro. Concurrency among strangers. In Rocco De Nicola and Davide Sangiorgi, editors, *1st international conference on Trustworthy global computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer Berlin / Heidelberg, 2005.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research directions in concurrent object-oriented programming*, chapter Analysis of inheritance anomaly in object-oriented concurrent programming languages, pages 107–150. MIT Press, Cambridge, MA, USA, 1993.

[Nar96] Bonnie A. Nardi, editor. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, 1996.

[NHSB05] Michael N. Huhns, Munindar P. Singh, and Mark et al. Burstein. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):69–70, November 2005.

[Nil94] Nils J. Nilsson. Teleo-reactive programs for agent control. *J. Artif. Int. Res.*, 1(1):139–158, January 1994.

[Nor91] Donald A. Norman. Cognitive artifacts. In John M. Carroll, editor, *Designing Interaction: Psychology at the Human-Computer Interface*, Cambridge Series On Human-Computer Interaction, pages 17–38. Cambridge University Press, New York, 1991.

[NRTV07] N. Nisan, T. Roughgarden, E. Tardos, and V.V. Vazirani. *Algorithmic game theory*. Cambridge University Press, 2007.

[Obj08] Object Management Group. Agent Meta-model and Profile (AMP) RFP, 2008. Online document, available at: `http://www.omg.org/cgi-bin/doc?ad/2008-09-05` – Last Retrieved: November 18, 2012.

[OD08] Daniel Okouya and Virginia Dignum. Operetta: a prototype tool for the design, analysis and development of multi-agent organizations. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers*, AAMAS '08, pages 1677–1678, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[Oli] Olivier Boissier and Rafael H. Bordini and Jomi F. Hbner and Alessandro Ricci and Andrea Santi. Official JaCaMo Website. Online document, available at: `http://jacamo.sourceforge.net/` – Last Retrieved: November 25, 2012.

[Ora] Orange Labs. Senscity Project Official Website. Online document, available at: `http://www.senscity-grenoble.com/` – Last Retrieved: November 28, 2012.

[ORV+04] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *Proc. of the 3rd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, volume 1, pages 286–293, New York, USA, 19–23July 2004. ACM.

[ORV08] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, December 2008.

[osftis06] OASIS (Advancing open standards for the information society). The WS-Security Specification, 2006. Online document, available at: `https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf` – Last Retrieved: November 26, 2012.

[osftis08a] OASIS (Advancing open standards for the information society). Official Website of the WS-AtomicTransaction Specification, 2008. Online document, available at: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-cd-02/wstx-wsat-1.2-spec-cd-02.html – Last Retrieved: November 26, 2012.

[osftis08b] OASIS (Advancing open standards for the information society). WS-Coordination Specification version 1.2, 2008. Online document, available at: `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-cd-02/wstx-wscoor-1.2-spec-cd-02.html` – Last Retrieved: November 26, 2012.

[Ous96] John Ousterhout. Why Threads Are A Bad Idea (for most purposes), 1996. Presented at USENIX Technical Conference.

[OVDPFB03] J. Odell, H. Van Dyke Parunak, M. Fleischer, and S. Brueckner. Modeling agents and their environment. In *AOSE–Agent-oriented Software Engineering*, pages 16–31. Springer, 2003.

[PBL05] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A bdi reasoning engine. In *Multi-Agent Programming: Languages, Platforms and Applications (Vol. I)*, pages 149–174. 2005.

[PPR11] C. Persson, G. Picard, and F. Ramparany. A multi-agent organization for the governance of machine-to-machine systems. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011 IEEE/WIC/ACM International Conference on*, volume 2, pages 421–424. IEEE, 2011.

[PRBH09] M. Piunti, A. Ricci, O. Boissier, and J.F. Hübner. Embodying organisations in multi-agent work environments. In *IEEE/WIC/ACM Int. Conf. on Web Intelligence and Intelligent Agent Technology (WI-IAT 2009)*, Milan, Italy., 2009.

[PRS09] Michele Piunti, Alessandro Ricci, and Andrea Santi. SOA/WS Applications using Cognitive Agents working in CArtAgO Environments. In *Proceedings of the 10th Workshop on Objects and Agents (WOA 2009)*, 2009.

[PRTG00] D.J. Povinelli, J.E. Reaux, L.A. Theall, and S. Giambrone. *Folk physics for apes: The chimpanzee's theory of how the world works*. Oxford University Press New York, 2000.

[PSR09] M. Piunti, A. Santi, and A. Ricci. Programming SOA/WS systems with BDI agents and artifact-based environments. In *Proceedings of MALLOW-AWESOME Joint Workshop on Agents, Web Services and Ontologies, Integrated Methodologies.*, Turin, Italy, 2009.

[PTCC99] David V. Pynadath, Milind Tambe, Nicolas Chauvat, and Lawrence Cavedon. Toward team-oriented programming. In *Agent Theories, Architectures, and Languages (ATAL)*, pages 233–247, 1999.

[PW02] S. Parsons and M. Wooldridge. Game theory and decision theory in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 5(3):243–254, 2002.

[RA81] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[Raf] Rafael H. Bordini and Jomi F. Hubner. Official Jason Website. Online document, available at: `http://jason.sourcefoge.net/` – Last Retrieved: November 15, 2012.

[Rao96] Anand S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away: agents breaking away*, MAAMAW '96, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[RDP10] A. Ricci, E. Denti, and M. Piunti. A platform for developing soa/ws applications as open and heterogeneous multi-agent systems. *Multiagent and Grid Systems*, 6(2):105–132, 2010.

[Rem] Rem W. Collier. Official Agent Factory Framework Website. Online document, available at: `http://www.agentfactory.com/index.php/Main_Page` – Last Retrieved: November 25, 2012.

[RG+95] A.S. Rao, M.P. Georgeff, et al. Bdi agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319, 1995.

[RG98] Anand S. Rao and Michael P. Georgeff. Readings in agents. chapter Modeling rational agents with a BDI-architecture, pages 317–328. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[Ric] Ricci, Alessandro and Santi, Andrea. simpAL Official Website. Online document, available at: `http://simpal.sourceforge.net/` – Last Retrieved: November 28, 2012.

[RJOC11] Sean Edward Russell, Howell R. Jordan, Gregory M. P. O'Hare, and Rem W. Collier. Agent factory: A framework for prototyping logic-based aop languages. In *Multiagent System Technologies (MATES)*, pages 125–136, 2011.

[RKL+05] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, January 2005.

[RN09] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice hall Englewood Cliffs, NJ, 3rd edition, 2009.

[ROD03] Alessandro Ricci, Andrea Omicini, and Enrico Denti. Activity Theory as a framework for MAS coordination. In Paolo Petta, Robert Tolksdorf, and Franco Zambonelli, editors, *Engineering Societies in the Agents World III*, volume 2577 of *LNCS*, pages 96–110. Springer Berlin / Heidelberg, April 2003.

[RPV09] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Externalisation and internalization: A new perspective on agent modularisation in multi-agent systems programming. In Mehdi Dastani, Amal El Fallah Seghrouchni, Joo Leite, and Paolo Torroni, editors, *Proceedings of MALLOW 2009 federated workshops: LAnguages, methodologies and Development tools for multi-agent systemS (LADS 2009)*, September 2009.

[RPV11] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems – an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, September 2011. Special Issue: Multi-Agent Programming.

[RPVO09] Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini. Environment programming in CArtAgO. In Rafael P. Bordini, Mehdi Dastani, Jurgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming II: Languages, Platforms and Applications*, Multiagent Systems, Artificial Societies, and Simulated Organizations, chapter 8, pages 259–288. Springer, June 2009.

[RS11a] Alessandro Ricci and Andrea Santi. Agent-oriented computing: Agents as a paradigm for computer programming and software development. In *Proc. of the 3rd Int. Conf. on Future Computational Technologies and Applications (Future Computing '11)*, pages 42–51, Rome, Italy, 2011. IARIA.

[RS11b] Alessandro Ricci and Andrea Santi. Designing a general-purpose programming language based on agent-oriented abstractions: the simpal project. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 159–170, New York, NY, USA, 2011. ACM.

[RS12a] Alessandro Ricci and Andrea Santi. From actors to agent-oriented programming abstractions in simpal. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, SPLASH '12, pages 73–74, New York, NY, USA, 2012. ACM.

[RS12b] Alessandro Ricci and Andrea Santi. Typing multi-agent programs in simpAL. In *Programming Multi-Agent Systems*, Valencia, Spain, 2012.

[RSP12] Alessandro Ricci, Andrea Santi, and Michele Piunti. Action and perception in agent programming languages: from exogenous to endogenous environments. In *Programming Multi-Agent Systems*, ProMAS'10, pages 119–138, Berlin, Heidelberg, 2012. Springer-Verlag.

[RVO07] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. CArtAgO: A framework for prototyping artifact-based environments in MAS. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 67–86. Springer, May 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.

[RVO08] A. Ricci, M. Viroli, and A. Omicini. The A&A Programming Model and Technology for Developing Agent Environments in MAS. In *Programming Multi-Agent Systems*, pages 89–106. Springer, 2008.

[RZ94] J.S. Rosenschein and G. Zlotkin. *Rules of encounter: designing conventions for automated negotiation among computers*. MIT press, 1994.

[SBPS11] A. Sorici, O. Boissier, G. Picard, and A. Santi. Exploiting the jacamo framework for realising an adaptive room governance application. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, pages 239–242. ACM, 2011.

[SCG98] Jaime Simão Sichman, Rosaria Conte, and Nigel Gilbert, editors. *Multi-Agent Systems and Agent-Based Simulation, First International Workshop, MABS '98, Paris, France, July 4-6, 1998, Proceedings*, volume 1534 of *Lecture Notes in Computer Science*. Springer, 1998.

[Sea69] J.R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

[Sen] Sencha Inc. Ext JS Official Website. Online document, available at: `http://www.sencha.com/products/extjs/` – Last Retrieved: November 25, 2012.

[SFS04] Alexandru Suna and Amal El Fallah-Seghrouchni. A mobile agents platform: Architecture, mobility and security elements. In *Programming Multi-Agent Systems*, pages 126–146, 2004.

[SFS07] Alexandru Suna and Amal El Fallah-Seghrouchni. Programming mobile intelligent agents: An operational semantics. *Web Intelligence and Agent Systems*, 5(1):47–67, 2007.

[SFT09] Tiberiu Stratulat, Jacques Ferber, and John Tranier. MASQ: towards an integral approach to interaction. In *International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 813–820, 2009.

[SGA10] Andrea Santi, Marco Guidi, and Ricci Alessandro. Exploiting Agent-Oriented Programming for Developing Android Applications. In Omicini Andrea and Viroli Mirko, editors, *Proceedings of the 11th Workshop on Objects and Agents (WOA 2010)*, volume 621 of *CEUR Workshop Proceedings*. Sun SITE Central Europe, RWTH Aachen University, 2010.

[SGR11] A. Santi, M. Guidi, and A. Ricci. Jaca-android: an agent-based platform for building smart mobile applications. *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, pages 95–114, 2011.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[Sil08] Jonathan Sillito. Stage: exploring erlang style concurrency in ruby. In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pages 33–40, New York, NY, USA, 2008. ACM.

[Sin96] Munindar P. Singh. Toward interaction-oriented programming. Technical report, North Carolina State University at Raleigh, Raleigh, NC, USA, 1996.

[Sin98] Munindar P. Singh. Agent communication languages: Rethinking the principles. *Computer*, 31(12):40–47, December 1998.

[Sin11] Munindar P. Singh. Information-driven interaction-oriented programming: Bspl, the blindingly simple protocol language. In *The 10th International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '11, pages 491–498, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue: Tomorrow's Computing Today*, 3(7):54–62, September 2005.

[SLNR11] Andrea Santi, Andrea Leardini, Antonio Natali, and Alessandro Ricci. Exploiting the eclipse ecosystem for agent-oriented programming. In *Proc. of The Sixth Workshop of the Italian Eclipse Community*, Milan, Italy, 2011.

[SM08] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. *ECOOP–European Conference on Object-Oriented Programming*, pages 104–128, 2008.

[Smi80] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transsactions on Computer*, 29:1104–1113, December 1980.

[Sor11] Alexandru Sorici. Agile governance in an AmI environment. Master's thesis, University "Politehnica" of Bucharest, September 2011.

[SR11a] A. Santi and A. Ricci. Exploiting intelligent agent-based technologies for programming smart mobile applications. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, pages 231–234. ACM, 2011.

[SR11b] Andrea Santi and Alessandro Ricci. Exploiting the eclipse ecosystem for agent-oriented programming. In *Proc. of The Seventh Workshop of the Italian Eclipse Community*, Milan, Italy, 2011.

[SR12] Andrea Santi and Alessandro Ricci. Programming distributed multi-agent systems in simpAL. In De Paoli Flavio and Vizzari Giuseppe, editors, *Proceedings of the 13th Workshop on Objects and Agents (WOA 2012)*, volume 892 of *CEUR Workshop Proceedings*. Sun SITE Central Europe, RWTH Aachen University, 2012.

[SSE86] L. Sterling, E. Shapiro, and M. Eytan. *The art of Prolog*, volume 94. Wiley Online Library, 1986.

[SUNa] SUN/ORACLE. Future API Specification. Online document, available at: `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html` – Last Retrieved: December 8, 2012.

[SUNb] SUN/ORACLE. Java Native Interface Specification. Online document, available at: `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html` – Last Retrieved: November 16, 2012.

[SUNc] SUN/ORACLE. LiveConnect Library Official Website. Online document, available at: `http://jdk6.java.net/plugin2/liveconnect/` – Last Retrieved: November 25, 2012.

[Sund] Sun/ORACLE. Official Website of Web Services Interoperability Technologies. Online document, available at: `http://wsit.java.net` – Last Retrieved: November 20, 2012.

[Sun05] Alexandru Suna. *CLAIM et SyMPA : Un environnement pour la programmation d'agents intelligents et mobiles*. PhD thesis, Universit Pierre et Marie Curie, 2005.

[Suta] Herb Sutter. The Free Lunch Is Over. Online document, available at: http://www.gotw.ca/publications/concurrency-ddj.htm – Last Retrieved: December 8, 2012.

[Sutb] Herb Sutter. Welcome to the Jungle. Online document, available at: `http://herbsutter.com/welcome-to-the-jungle/` – Last Retrieved: December 8, 2012.

[SW87] Bruce Shriver and Peter Wegner, editors. *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[Tel] Telecom Italia Labs. Official JADE Website. [Online; accessed 25-November-2012].

[Thea] The Dojo Foundation. Dojo Official Website. Online document, available at: `http://dojotoolkit.org/` – Last Retrieved: November 25, 2012.

[Theb] The jQuery Foundation. jQuery Official Website. Online document, available at: `http://jquery.com/` – Last Retrieved: November 25, 2012.

[Tho95] S. Rebecca Thomas. The placa agent programming language. In *ECAI Workshop on Agent Theories, Architectures, and Languages*, ECAI-94, pages 355–370, New York, NY, USA, 1995. Springer-Verlag New York, Inc.

[Tim] Tim Jansen. The actor guild framework official website. Online document, available at: `http://actorsguildframework.org/` – Last Retrieved: December 8, 2012.

[Typa] Typesafe. The akka framework official website. Online document, available at: `http://akka.io/` – Last Retrieved: December 8, 2012.

[Typb] Typesafe. The scala programming language official website. Online document, available at: `http://scala-lang.org/` – Last Retrieved: December 8, 2012.

[VA01] C.A. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Notices*, 36(12):20–34, 2001.

[vBCB03] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. of HOTOS'03*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[VCMDM09] T. Van Cutsem, S. Mostinckx, and W. De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems & Structures*, 35(1):80–98, 2009.

[VHR+07] Mirko Viroli, Tom Holvoet, Alessandro Ricci, Kurt Schelfthout, and Franco Zambonelli. Infrastructures for the environment of multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):49–60, jul 2007. Special Issue: Environment for Multi-Agent Systems.

[VRO06] Mirko Viroli, Alessandro Ricci, and Andrea Omicini. Operating instructions for intelligent agent coordination. *The Knowledge Engineering Review*, 21(1):49–69, March 2006.

[VSMS12] Daniel Villatoro, Jordi Sabater-Mir, and Jaime Simão Sichman, editors. *Multi-Agent-Based Simulation XII - International Workshop, MABS 2011, Taipei, Taiwan, May 2-6, 2011, Revised Selected Papers*, volume 7124 of *Lecture Notes in Computer Science*. Springer, 2012.

[256] World Wide Web Consortium (W3C). Official Website of the Simple Object Access Protocol (SOAP) Specification, 2007. Online document, available at: `http://www.w3.org/TR/soap/` – Last Retrieved: November 23, 2012.

[WHSA05] A.B. Wood, T.E. Horton, and R. St Amant. Effective tool use in a habile agent. In *Systems and Information Engineering Design Symposium 2005*, pages 75–81. IEEE, 2005.

[Wik] Wikimedia Foundation. Promise pipelining article on wikipedia. Online document, available at: `http://en.wikipedia.org/wiki/Promise_pipelining` – Last Retrieved: December 8, 2012.

[Woo97] M. Wooldridge. Agent-based software engineering. In *IEE Proceedings - Software*, volume 144, pages 26–37. IET, 1997.

[WOO07] D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Autonomous agents and multi-agent systems*, 14(1):5–30, 2007.

[Woo09] Michael Wooldridge. *An Introduction to Multiagent Systems (second edition)*. Wiley, 2. edition, 2009.

[Wora] World Wide Web Consortium (W3C). The ECMAScript Language Specification, Version 5.1. Online document, available at: `http://www.ecma-international.org/publications/standards/Ecma-262.htm` – Last Retrieved: November 25, 2012.

[Worb] World Wide Web Consortium (W3C). The HTML5 Specification. Online document, available at: `http://dev.w3.org/html5/spec/` – Last Retrieved: November 25, 2012.

[Worc] World Wide Web Consortium (W3C). The HTML5 Web Messaging Specification. Online document, available at: `http://dev.w3.org/html5/postmsg/` – Last Retrieved: November 25, 2012.

[Word] World Wide Web Consortium (W3C). The Web Workers Specification. Online document, available at: `http://www.w3.org/TR/workers/` – Last Retrieved: November 25, 2012.

[Wor04] World Wide Web Consortium (W3C). Web Services Architecture, 2004. Online document, available at: `http://www.w3.org/TR/ws-arch/` – Last Retrieved: November 18, 2012.

[Wor07] World Wide Web Consortium (W3C). Official Website of the Simple Object Access Protocol (SOAP) Specification, 2007. Online document, available at: `http://www.w3.org/TR/wsdl20/` – Last Retrieved: November 23, 2012.

[WPM05] Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors. *Environments for Multi-Agent Systems, First International Workshop, E4MAS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, volume 3374 of *Lecture Notes in Computer Science*. Springer, 2005.

[WPM06] Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors. *Environments for Multi-Agent Systems II, Second International Workshop, E4MAS 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers*, volume 3830 of *Lecture Notes in Computer Science*. Springer, 2006.

[WPM07] Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors. *Environments for Multi-Agent Systems III, Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers*, volume 4389 of *Lecture Notes in Computer Science*. Springer, 2007.

[Wri09] Alex Wright. Get smart. *Communication of ACM*, 52:15–16, January 2009.

[WSI] WSI Group (Web Service Interoperability Organization). Deliverables from the Basic Profile Working Group. Online document, available at: `http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile` – Last Retrieved: November 18, 2012.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP–European Conference on Object-Oriented Programming*, pages 55–77, London, UK, UK, 1988. Springer-Verlag.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming abcl/1. *SIGPLAN Not.*, 21(11):258–268, June 1986.

[Yon90] Akinori Yonezawa, editor. *ABCL: an object-oriented concurrent system*. MIT Press, Cambridge, MA, USA, 1990.

[YS02] Pinar Yolum and Munindar P. Singh. Commitment machines. In *Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, ATAL '01, pages 235–247, London, UK, UK, 2002. Springer-Verlag.

[YT87] A. Yonezawa and M. Tokoro. *Object-oriented concurrent programming*. MIT Press series in computer systems. MIT Press, 1987.