Università degli studi di Genova

Laurea magistrale in Informatica

Trace expressions for runtime verification and protocol-driven behaviour

Autore:
Angelo Ferrando

Relatori:
Prof. Davide Ancona
Prof. Viviana Mascardi

Correlatore:
Prof. Gianna Reggio

Tesi presentata nel rispetto dei requisiti
per il grado di Laurea Magistrale in Informatica

nel

Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

"Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so."

Douglas Adams

Contents

List of Figures

Dedicated to my family

Introduction

Today's software systems raise many challenges to their designers as they are required to be more and more autonomous, recoverable and reliable to guarantee the expected level of the offered services. Achieving all the three goals together requires to find the right balance between the ability of the system to operate with a high degree of freedom, including its ability to recover to an acceptable state in case of exceptional situations, and the guarantee of a behaviour compliant with the designers' requirements.

Self-adaptive systems, namely systems able to modify their behaviour and/or structure in response to their perception of the environment and the system itself, and their goals [83], are a widely accepted answer to the increasing need of autonomy and recoverability of modern complex systems.

Reliability can be achieved by enforcing all the system's components to respect given patterns of behaviour, known to be safe.

In this thesis we address the design and implementation of self-adaptive multiagent systems where compliance to a given interaction protocol is guaranteed by construction, allowing agents to operate in a reliable way. In the proposed framework, agents are driven by first class specifications of interaction protocols which can change at runtime, making self-adaptation a side result obtained almost for free.

Adaptation takes place according to the instructions of agents empowered to request protocol switches and acting as central controllers. In the most general setting, different agents could take the role of controller in different moments, or even in the same moment provided that they coordinate themselves for controlling the system. In our framework we assume that there is only one controller agent at

a time. Since switching to a new protocol may require to be in some safe state[1], the protocol's designer can specify when the agents are allowed to manage a switch request (and, as a consequence, when they are not).

We present the formalism that we have designed to define our protocols, in particular, we focus on the possible uses which can be made of it, starting from the use for runtime verification, where the goal is to create a monitor that checks the behaviour of a set of agents, to the more innovative use for "protocol-driven" behaviour, where all the agents are correct by construction. In the context of runtime verification, we compare our formalism to more traditional ones like LTL, showing the benefits resulting from the use of our formalisms. In particular, we show that any LTL formula can be translated into a term of our formalism which is equivalent from the point of view of runtime verification.

The code of "protocol-driven" agents is not generated from the protocol specification prior to the MAS deployment, as this would hard wire the protocol-compliant behaviour into the code preventing agents from adapting to protocol changes. Rather, the agent's interpreter takes the current state of the interaction protocol into account to devise which messages could be sent, allowing the agent to select and send one of them, or which messages could be received, allowing the agent to verify whether the received message – if any – was one among the expected ones and to react in a suitable way. Changing the executing protocol as the result of an acceptable protocol switch request causes the interpreter to call a cleanup procedure and to proceed in the normal way following the new protocol.

A characterizing feature of our approach is that protocol specifications take a global, rather than a local, perspective and each agent, before starting to follow a new protocol, projects the protocol onto itself by removing the protocol's components not involving itself. If all the agents in the MAS are driven by the same global protocol, the compliance of the MAS execution to the protocol comes for free.

In this work we present the formalism which can be used - and it has been used - to represent a large number of protocols. In fact, with our formalism, we can specify *regular*, *context-free* and *non context-free* languages[2]. This expressive power is a

[1] Consider for example an agent following the protocol "drive home" who is required by his kid in the backseat to move to the protocol "look at me, I feel bad...". Even if looking at the suffering kid is a high priority protocol to follow, the agent can move to it only when the car's speed is low enough, or, better, the car has been stopped.

[2] In Section 4.3 we report some examples of languages that we have implemented.

key advantage when we compare our formalism with LTL in the runtime verification context, considering that LTL is less expressive than ω-regular languages (this aspect will be clearer in Section 4.4).

The proposed self-adaptive MAS framework is not only presented, designed and theorized but it is also implemented in two famous tools used for the development of MASs: Jason and JADE[3]. Both the implementations follow the same approach, in fact, they keep the representation of the protocols (using the fixed formalism) in the same way distinguishing only the implementation of the protocol-driven agent's interpreter, that is the body of the agent which must query the protocol in order to know what the agent can do in the current state. Consequently the implementation in these two tools (and in any other tool) is extremely faster because the protocol representation, interrogation and so on are obtained once and for all using the same language[4], without the necessity to rewrite the same code for each different implementation (promoting the reuse of code).

In order to show that our framework works correctly, we have reported all the results obtained by the experiments made in Jason and JADE. In Chapter 5 we present some examples of protocol specification using our formalism, after that, in Sections 6.3.2 and 6.3.3 we report all outputs produced by the execution of the protocol-driven agents which follow the protocols defined in Chapter 5. In both cases, that are Jason and JADE implementations, we have obtained excellent results showing that in both implementations all agents work rightly and they are correct by construction following the protocol totally.

The thesis is structured as follows, in Chapter 1 we give all theoretical and practical basics, in Chapter 2 we present the state of the art, in Chapter 3 we introduce the building blocks of our framework, in Chapter 4 we define the formalism which we have used to represent our protocols, in Chapter 5 we report some protocol defined with the formalism presented, in Chapter 6 we analyze all technical details and the implementation of our framework, in Chapter 7 we focus on the conclusions and the presentation of some possible future developments.

[3]We present both tools in Sections 1.2 and 1.3 respectively.
[4]All technical details are given in Chapter 6.

Chapter 1

Background

1.1 Multiagent Systems and Distributed Artificial Intelligence

Since its inception in the mid to late 1970s distributed artificial intelligence (DAI) evolved and diversified rapidly. Today it is an established and promising research and application field which brings together and draws on results, concepts, and ideas from many disciplines, including artificial intelligence (AI), computer science, sociology, economics, organization and management science, and philosophy.

As defined in [90], an agent is a computational entity such as a software program or a robot that can be viewed as perceiving and acting upon its environment and that is autonomous in that its behaviour at least partially depends on its own experience. As an intelligent entity, an agent operates flexibly and rationally in a variety of environmental circumstances given its perceptual and effectual equipment. Behavioral flexibility and rationality are achieved by an agent on the basis of key processes such as problem solving, planning, decision making, and learning. As an interacting entity, an agent can be affected in its activities by other agents and perhaps by humans. A key pattern of event in multiagent systems is *goal-* and *task-oriented* coordination, both in cooperative and in competitive situations. In the case of cooperation several agents try to combine their efforts to accomplish as a group what the individuals cannot, and in the case of competition several agents try to get what only some of them can have. The long-term goal of DAI is to develop mechanisms and methods that enable agents to interact

like humans (or even better), and to understand events among intelligent entities whether they are computational, human, or both. This goal raises a number of challenging issues that all are centered around the elementary question of when and how to interact with whom.

To make the above considerations more concrete, a closer look has to be taken on multiagent systems and thus on "interacting, intelligent agents":

- "Agents" are autonomous, computational entities that can be viewed as perceiving their environment through sensors and acting upon their environment through effectors. To say that agents are computational entities means that they physically exist in the form of programs that run on computing devices. To say that they are autonomous means that to some extent they have control over their behaviour without the intervention of humans and other systems. Agents pursue goals or carry out tasks in order to meet their design objectives, and in general these goals and tasks can be supplementary as well as conflicting.
  For N. R. Jennings et al. [64], an agent is a computer system, situated in some environment, that is capable of flexible autonomous actions in order to meet its design objectives. There are thus three key concepts in their definition: situatedness, autonomy, and flexibility. In detail,

  - Situatedness, in this context, means that the agent receives sensory input from its environment and that it can perform actions which change the environment in some way;
  - Autonomy is a difficult concept to pin down precisely, but they mean it in the sense that the system should be able to act without the direct intervention of humans (or other agents), and that it should have control over its own actions and internal state;
  - By flexible, they mean that the system is: *responsive*, agents should perceive their environment and respond in a timely fashion to changes that occur in it, *pro-active*, agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behaviour and take the initiative where appropriate and *social*, agents should be able to interact, when appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities.

- "Intelligent" indicates that the agents pursue their goals and execute their tasks such that they optimize some given performance measures. To say that agents are intelligent does not mean that they are omniscient or omnipotent, nor does it mean that they never fail. Rather, it means that they operate flexibly and rationally in a variety of environmental circumstances, given the information they have and their perceptual and effectual capabilities.

- "Interacting" indicates that the agents may be affected by other agents or perhaps by humans in pursuing their goals and executing their tasks. Events can take place indirectly through the environment in which they are embedded (e.g., by observing one another or by carrying out an action that modifies the environmental state) or directly through a shared language (e.g., by providing information in which other agents are interested or which confuses other agents). DAI primarily focuses on coordination as a form of event that is particularly important with respect to goal attainment and task completion. The purpose of coordination is to achieve or avoid states of affairs that are considered as desirable or undesirable by one or several agents. To coordinate their goals and tasks, agents have to explicitly take dependencies among their activities into consideration. Two basic, contrasting patterns of coordination are cooperation and competition. In the case of cooperation, several agents work together and draw on the broad collection of their knowledge and capabilities to achieve a common goal. Against that, in the case of competition, several agents work against each other because their goals are conflicting. Cooperating agents try to accomplish as a team what the individuals cannot, and so fail or succeed together. Competitive agents try to maximize their own benefit at the expense of others, and so the success of one implies the failure of others.

1.1.1 Agent Communications Language performatives

Performativity is a term for the capacity of speech and communication not just to communicate but rather to act or consummate an action, or to construct and perform an identity. A common example is the act of saying "I pronounce you man and wife" by a licensed minister before two people who are prepared to wed (or "I do" by one of those people upon being asked whether they take their partner in

marriage). An umpire calling a strike, a judge pronouncing a verdict, or a union boss declaring a strike are all examples of performative speech.

Speech Act Theory as introduced by Oxford philosopher J.L. Austin [11] and further developed by American philosopher J.R. Searle, considers the types of acts that utterances can be said to perform:

- Locutionary Acts;
- Illocutionary Acts;
- Perlocutionary Acts.

Agent Communication Language (ACL) is based on the speech act theory: messages are actions, or communicative acts, as they are intended to perform some action by virtue of being sent. The specification consists of a set of message types and the description of their pragmatics, that is the effects on the mental attitudes of the sender and receiver agents. Every communicative act is described with both a narrative form and a formal semantics based on modal logic.

The most popular ACLs are:

- FIPA-ACL [1] (by the Foundation for Intelligent Physical Agents, a standardization consortium);
- KQML [50] (Knowledge Query and Manipulation Language).

Both rely on the speech act theory developed by Searle in the 1960s [86] and enhanced by Winograd and Flores in the 1970s. They define a set of performatives, also called Communicative Acts, and their meaning (e.g. tell). The content of the performative is not standardized, but varies from language to language.

Speech act theory uses the term performative to identify the illocutionary force of this special class of utterance. Example performative verbs include promise, report, convince, insist, tell, request, and demand. Illocutionary forces can be broadly classified as assertives (statements of fact), directives (commands in a master-slave structure), commissives (commitments), declaratives (statements of fact), and expressives (expressions of emotion). Performatives are usually represented in the stylized syntactic form "I hereby tell..." or "I hereby request..."

Because performatives have the special property that "saying it makes it so," not all verbs are performatives. For example, stating that "I hereby solve this problem" does not create the solution. Although the term speech is used in this discussion, speech acts have to do with communication in forms other than the spoken word. In summary, speech act theory helps define the type of message by using the concept of the illocutionary force, which constrains the semantics of the communication act itself. The sender's intended communication act is clearly defined, and the receiver has no doubt as to the type of message sent. This constraint simplifies the design of our software agents.

To make agents understand each other they have to speak the same language; they can also have a common ontology. An ontology is a part of the agent's knowledge base that describes what kind of things an agent can deal with and how they are related to each other.

1.2 Jason

Jason[1] [22] is an engine for an extended version of the AgentSpeak language [84]. It implements the operational semantics of that language, and provides a platform for the development of multiagent systems, with many user-customisable features. Jason is available Open Source, and is distributed under GNU LGPL.

One of the most interesting aspects of AgentSpeak is that it was inspired by and based on a model of human behaviour that was developed by philosophers. This model is called the belief-desire-intention (BDI) model. Belief-desire-intention architectures originated from the work of the Rational Agency project at Stanford Research Institute in the mid-1980s. The origins of the model lie in the theory of human practical reasoning developed by the philosopher Michael Bratman [23], which focuses particularly on the role of intentions in practical reasoning. The conceptual framework of the BDI model is described in [24], which also describes a specific BDI agent architecture called IRMA.

We describe all constructs of the extended version of the AgentSpeak language used by Jason that programmers can use, which can be separated into three main categories: beliefs, goals and plans. There are various different types of constructs used in each category.

[1] `http://jason.sourceforge.net/`

1.2.1 Beliefs

The very first thing to learn about AgentSpeak agents is how to represent the agents' beliefs. As we mentioned above, an agent has a belief base, which in its simplest form is a collection of literals, as in traditional logic programming. In programming languages inspired by logics, information is represented in symbolic form by predicates such as:

```
tall(john).
```

which expresses a particular property - that of being tall - of an object or individual, in this case 'John' (the individual to whom the term `john` refers). To represent the fact that a certain relationship holds between two or more objects, we can use a predicate such as:

```
likes(john, music).
```

which, pretty obviously, states that John likes music. A literal is such a predicate or its negation. However, unlike in classical logic, we are here referring to the concept of modalities of truth as in the modal logic literature, rather than things that are stated to be true in absolute terms. When a formula such as `likes(john, music)` appears in an agent's belief base, that is only meant to express the fact that the agent currently believes that to be true; it might well be the case that John does not like music at all.

1.2.1.1 Annotations

Perhaps one important difference in the syntactical representation of logical formula in Jason compared with traditional logic programming is the use of annotations. These are complex terms providing details that are strongly associated with one particular belief. Annotations are enclosed in square brackets immediately following a literal, for example:

```
busy(john)[expires(autumn)].
```

which, presumably, means that the agent believes that John is busy, but as soon as autumn starts, that should be no longer believed to hold. Note that the annotation expires(autumn) is giving us further information about the busy(john) belief in particular.

1.2.1.2 Rules

Rules allow us to conclude new things based on things we already know. Including such rules in an agent's belief base can simplify certain tasks, for example in making certain conditions used in plans more succinct. The notation is slightly different respect to the Prolog language, the reason being that the types of formula that can appear in the body of the rules also appear elsewhere in the AgentSpeak language (in a plan 'context'), so we use that same notation here for the sake of homogeneity.

Consider the following rules:

```
likely_color(C,B) :-
  color(C,B)[source(S)] & (S == self | S == percept).
likely_color(C,B) :-
  color(C,B)[degOfCert(D1)] &
  not (color(_,B)[degOfCert(D2)] & D2 > D1) &
  not ~color(C,B).
```

The first rule says that the most likely color of a box is either that which the agent deduced earlier, or the one it has perceived. If this fails, then the likely color should be the one with the highest degree of certainty associated with it, provided there is no strong evidence the color of the box is not that one. To the left of the ' `:-` ' operator, there can be only one literal, which is the conclusion to be made if the condition to the right is satisfied (according to the agent's current beliefs).

1.2.2 Goals

In agent programming, the notion of goal is fundamental. Indeed, many think this is the essential, defining characteristic of agents as a programming paradigm.

Whereas beliefs, in particular those of a perceptual source, express properties that are believed to be true of the world in which the agent is situated, goals express the properties of the states of the world that the agent wishes to bring about. Although this is not enforced by the AgentSpeak language, normally, when representing a goal `g` in an agent program, this means that the agent is committed to act so as to change the world to a state in which the agent will, by sensing its environment, believe that g is indeed true. This particular use of goals is referred to in the agent programming literature as a declarative goal.

In AgentSpeak, there are two types of goals: *achievement goals* and *test goals*. The type of goal described above is (a particular use of) an achievement goal, which is denoted by the ‘ `!` ’ operator. So, for example, we can say `!own(house)` to mean that the agent has the goal of achieving a certain state of affairs in which the agent will believe it owns a house, which probably implies that the agent does not currently believe that ‘ `own(house)` ’ is true.

The fact that an agent may adopt a new goal leads to the execution of plans, which are essentially courses of actions the agent expects will bring about the achievement of that goal.

Perhaps an important distinction to make is with the notion of ‘goal’ as used in Prolog, which is rather different from the notion of goal used above. A goal or ‘goal clause’ in Prolog is a conjunction of literals that we want the Prolog interpreter to check whether they can be concluded from the knowledge represented in the program; the Prolog interpreter is essentially proving that the clause is a logical consequence of the program. This is also something that we will need doing as part of our agents - that is, checking whether the agent believes a literal or conjunction of literals - and this is why there is another type of goal in AgentSpeak, called a test goal and denoted by the ‘ `?` ’ operator. Normally, test goals are used to retrieve information that is available in the agent’s belief base. Therefore, when we write `?bank_balance(BB)`, that is typically because we want the logical variable `BB` to be instantiated with the specific amount of money the agent currently believes its bank balance is.

1.2.3 Plans

An AgentSpeak plan has three distinct parts: the triggering event, the context, and the body. Together, the triggering event and the context are called the head of the plan. The three plan parts are syntactically separated by ‘ : ’ and ‘ <- ’ as follows:

```
triggering_event : context <- body.
```

We briefly describe what is the idea behind each part of a plan.

1.2.3.1 Triggering event

As we saw in the initial chapters, there are two important aspects of agent behaviour: reactiveness and pro-activeness. Agents have goals which they try to achieve in the long term, determining the agent's pro-active behaviour. However, while acting so as to achieve their goals, agents need to be attentive to changes in their environment, because those changes can determine whether the agents will be effective in achieving their goals and indeed how efficient they will be in doing so. More generally, changes in the environment can also mean that there are new opportunities for the agents to do things, and perhaps opportunities for considering adopting new goals that they previously did not have or indeed dropping existing goals. There are, accordingly, two types of changes is an agent's mental attitudes which are important in an agent program: changes in beliefs (which, recall, can refer to the information agents have about their environment or other agents) and changes in the agent's goals. Changes in both types of attitudes create, within the agent's architecture, the events upon which agents will act. Further, such changes can be of two types: addition and deletion. As we mentioned in the beginning of this chapter, plans are courses of actions that agents commit to execute as a consequence of such changes (i.e. events). The triggering event part of a plan exists precisely to tell the agent, for each of the plans in their plan library, which are the specific events for which the plan is to be used. If an event that took place matches the triggering event of a plan, that plan might start to execute, provided some conditions are satisfied. If the triggering event of a plan matches a particular event, we say that the plan is relevant for that particular event.

1.2.3.2 Context

As for triggering events, the context of a plan also relates to an important aspect of reactive planning systems. We have seen that agents have goals, and plans are used to achieve them, but also that agents have to be attentive to changes in the environment. Dynamic environments are complicated to deal with first because changes in the environment may mean we have to act further, but also because, as the environment changes, the plans that are more likely to succeed in achieving a particular goal also change. This is why reactive planning systems postpone committing to courses of action (i.e. a plan) so as to achieve a particular goal until as late as possible; that is, the choice of plan for one of the many goals an agent has is only made when the agent is about to start acting upon it. Typically, an agent will have various different plans to achieve the same goal (in addition, various plans for different goals can be competing for the agent's attention). The context of a plan is used precisely for checking the current situation so as to determine whether a particular plan, among the various alternative ones, is likely to succeed in handling the event (e.g. achieving a goal), given the latest information the agent has about its environment. Therefore, a plan is only chosen for execution if its context is a logical consequence - and we discuss later exactly what that means - of the agent's beliefs. A plan that has a context which evaluates as true given the agent's current beliefs is said to be applicable at that moment in time, and is a candidate for execution.

1.2.3.3 Body

The body of a plan, in general terms, is the easiest part of a plan to understand. This is a sequence of formulas determining a course of action - one that will, hopefully, succeed in handling the event that triggered the plan. However, each formula in the body is not necessarily a 'straight' action to be performed by the agent's effectors (so as to change the environment). Another important construct appearing in plan bodies is that of a goal, this allows us to say what are the (sub)goals that the agent should adopt and that need to be achieved in order for that plan to handle an event successfully. We can refer to the term subgoals, given that the plan where they appear can itself be a plan to achieve a particular goal - recall that the triggering events allow us to write plans to be executed when the

agent has a new goal to achieve. In fact, there are other things that can appear in the body of a plan.

1.2.4 Agent life cycle

We now describe how the Jason engine runs an agent program. An agent operates by means of a reasoning cycle, which can be divided into 10 main steps:

1. *Perceiving the environment* - The agent senses the environment so as to update its beliefs about the state of environment.

2. *Updating the belief Base* - Once the list of percepts has been obtained, the belief base needs to be updated to reflect perceived changes to the environment.

3. *Receiving communication from other agents* - The engine checks for messages that might have been delivered to the agent's "mailbox".

4. *Selecting "socially acceptable" messages* - Before messages are processed, they go through a selection process to determine whether they can be accepted by the agent or not.

5. *Selecting an event* - In each reasoning cycle, only one pending event will be dealt with. Therefore, the reasoning engine needs to select an event to be handled in a particular reasoning cycle. If the set of events turns out to be empty (i.e. there have been no changes in the agent's beliefs and goals since the last event was handled), then the reasoning cycle proceeds directly to step 9.

6. *Retrieving all relevant plans* - After an event has been selected, the engine of the language should find a plan that will allow the agent to act so as to handle that event. The first thing to do is to find, in the Plan Library component all plans which are relevant for the given event. This is done by retrieving all plans from the agent's plan library that have a triggering event that can be unified with the selected event.

7. *Determining the applicable plans* - The reasoning engine needs to select, from the relevant plans, all those which are currently applicable; in order to do this, it needs to check whether the context of each of the relevant

plans is believed to be true; in other words, whether the context is a logical consequence of the agent's belief base.

8. *Selecting one applicable plan* - The predefined selection function chooses an applicable plan based on the order in which they appear in the plan library. This in turn is determined by the order in which plans are written in an agent source code, or indeed the order in which plans are communicated to the agent.

9. *Selecting an intention for further execution* - Typically an agent has more than one intention in the set of intentions, each representing a different focus of attention. Unless the programmer customizes this selection function, the agent will be dividing its attention equally among all of its intentions.

10. *Executing one step of an intention* - The engine behaves according to each type of formula. Recall that there are six different types of formula that can appear in a plan body: environment action, achievement goals, test goals, mental notes, internal actions and expressions.

Final stage before restarting the cycle - Some intentions might be in the set of suspended intentions either waiting for feedback on action execution or waiting for message replies from other agents. Before another reasoning cycle starts, the engine checks whether any such feedback and replies are now available, and if so, the relevant intentions are updated (e.g. further instantiation might occur) and included back in the set of intentions, so that they have a chance of being selected for further execution in the next reasoning cycle (at Step 9). The agent is now ready to start another reasoning cycle (Step 1).

1.3 JADE

JADE[2] (Java Agent DEvelopment Framework) [18] is a software Framework fully implemented in the Java language. It simplifies the implementation of multiagent systems through a middleware that complies with the FIPA specifications and through a set of graphical tools that support the debugging and deployment phases. A JADE-based system can be distributed across machines (which not even need

[2] `http://jade.tilab.com/`

to share the same OS) and the configuration can be controlled via a remote GUI. The configuration can be even changed at run-time by moving agents from one machine to another, as and when required. JADE is completely implemented in Java language and the minimal system requirement is the version 5 of JAVA (the run time environment or the JDK).

Besides the agent abstraction, JADE provides a simple yet powerful task execution and composition model, peer to peer agent communication based on the asynchronous message passing paradigm, a yellow pages service supporting publish subscribe discovery mechanism and many other advanced features that facilitate the development of a distributed system.

Thanks to the contribution of the LEAP project, ad hoc versions of JADE exist designed to deploy JADE agents transparently on different Java-oriented environments such as Android devices and J2ME-CLDC MIDP 1.0 devices. Furthermore suitable configurations can be specified to run JADE agents in networks characterized by partial connectivity including NAT and firewalls as well as intermittent coverage and IP-address changes.

1.3.1 Architecture

A JADE platform is composed of agent containers that can be distributed over the network. Agents live in containers which are the Java process that provides the JADE runtime and all the services needed for hosting and executing agents. There is a special container, called the *main container*, which represents the bootstrap point of a platform: it is the first container to be launched and all other containers must join to a main container by registering with it.

When the main container is launched, two special agents are automatically instantiated and started by JADE, whose roles are defined by the FIPA Agent Management standard:

1. The Agent Management System (AMS) is the agent that supervises the entire platform. It is the contact point for all agents that need to interact in order to access the white pages of the platform as well as to manage their life cycle. Every agent is required to register with the AMS (automatically carried out by JADE at agent start-up) in order to obtain a valid `AID`.

2. The Directory Facilitator (DF) is the agent that implements the yellow pages service, used by any agent wishing to register its services or search for other available services. The JADE DF also accepts subscriptions from agents that wish to be notified whenever a service registration or modification is made that matches some specified criteria. Multiple DFs can be started concurrently in order to distribute the yellow pages service across several domains. These DFs can be federated, if required, by establishing cross-registrations with one another which allow the propagation of agent requests across the entire federation.

1.3.2 Creating agents

Creating a JADE agent is as simple as defining a class that extends the `jade.core.Agent` class and implementing the `setup()` method as exemplified in the code below.

```
import jade.core.Agent;
public class HelloWorldAgent extends Agent {
  protected void setup() {
    // Printout a welcome message
    System.out.println("Hello World. I'm an agent!");
  }
}
```

More appropriately a class, such as the `HelloWorldAgent` class shown above, represents a type of agent exactly as a normal Java class represents a type of object. Several instances of the `HelloWorldAgent` class can be launched at runtime. Unlike normal Java objects, which are handled by their references, an agent is always instantiated by the JADE run-time and its reference is never disclosed outside the agent itself (unless of course the agent does that explicitly). Agents never interact through method calls but rather by exchanging asynchronous messages.

The `setup()` method is intended to include agent initializations. The actual job an agent has to perform is typically carried out within "behaviours". Examples of typical operations that an agent performs in its setup() method are: showing a GUI, opening a connection to a database, registering the services it provides in the yellow pages catalogue and starting the initial behaviours. It is good practice not to define any constructor in an agent class and to perform all initializations inside the `setup()` method. This is because at construction time the agent is not yet

linked to the underlying JADE run-time and thus some of the methods inherited from the Agent class may not work properly.

1.3.2.1 Agent identifiers

Consistent with the FIPA specifications, each agent instance is identified by an "agent identifier". In JADE an agent identifier is represented as an instance of the `jade.core.AID` class. The `getAID()` method of the `Agent` class allows retrieval of the local agent identifier. An `AID` object includes a globally unique name (`GUID`) plus a number of addresses. The name in JADE has the form `<local-name>@<platform-name>` such that an agent called `Peter` living on a platform called `foo-platform` will have `Peter@foo-platform` as its globally unique name. The addresses included in the `AID` are the addresses of the platform the agent inhabits. These addresses are only used when an agent needs to communicate with another agent living on a different compliant FIPA platform.

The `AID` class provides methods to retrieve the local name (`getLocalName()`), the `GUID` (`getName()`) and the addresses (`getAllAddresses()`).

We can therefore enrich the welcome message of our `HelloWorldAgent` as follows:

```
protected void setup() {
  // Printout a welcome message
  System.out.println("Hello World. I'm an agent!");
  System.out.println("My local-name is "+getAID().getLocalName());
  System.out.println("My GUID is "+getAID().getName());
  System.out.println("My addresses are:");
  Iterator it = getAID().getAllAddresses();
  while (it.hasNext()) {
    System.out.println("- "+it.next());
  }
}
```

The local name of an agent is assigned at start-up time by the creator and must be unique within the platform. If an agent with the same local name already exists in the platform, the JADE runtime prevents the creation of the new agent.

```
String localname = "Peter";
AID id = new AID(localname, AID.ISLOCALNAME);
```

The platform name is automatically appended to the `GUID` of the newly created `AID` by the JADE runtime. Similarly, knowing the `GUID` of an agent, its `AID` can be obtained as follows:

```
String guid = "Peter@foo-platform";
AID id = new AID(guid, AID.ISGUID);
```

1.3.2.2 Agent initialization

The `HelloWorldAgent` class described previously can be compiled, as with normal Java classes, by typing:

```
javac -classpath <JADE-classes> HelloWorldAgent.java
```

Of course the JADE libraries must be in the Classpath for the compilation to succeed. At that point, in order to execute a Hello World agent, i.e. an instance of the `HelloWorldAgent` class, the JADE runtime must be started and a local name for the agent to execute must be chosen:

```
java -classpath <JADE-classes>;. jade.Boot Peter:HelloWorldAgent
```

This command starts the JADE runtime and tells it to launch an agent whose local name is `Peter` and whose class is `HelloWorldAgent`. Again both the JADE libraries and the `HelloWorldAgent` class must be in the Classpath. As a result of the typed command, the following printouts produced by the Hello World agent should appear.

```
Hello World. I'm an agent!
My local-name is Peter
My GUID is Peter@anduril:1099/JADE
My addresses are:
- http://anduril:7778/acc
```

1.3.2.3 Agent tasks

The actual job, or jobs, an agent has to do is carried out within "behaviours". A behaviour represents a task that an agent can carry out and is implemented as an

object of a class that extends `jade.core.behaviours.Behaviour`. To make an agent execute the task implemented by a behaviour object, the behaviour must be added to the agent by means of the `addBehaviour()` method of the `Agent` class.

Behaviours can be added at any time when an agent starts up (in the `setup()` method) or from within other behaviours. Each class extending `Behaviour` must implement two abstract methods. The `action()` method defines the operations to be performed when the behaviour is in execution. The `done()` method returns a boolean value to indicate whether or not a behaviour has completed and is to be removed from the pool of behaviours an agent is executing.

Behaviour scheduling and execution. An agent can execute several behaviours concurrently. However, it is important to note that the scheduling of behaviours in an agent is not pre-emptive (as for Java threads), but cooperative. *This means that when a behaviour is scheduled for execution its `action()` method is called and runs until it returns.* Therefore it is the programmer who defines when an agent switches from the execution of one behaviour to the execution of another.

This approach often creates difficulties for inexperienced JADE developers and must always be kept in mind when writing JADE agents. Though requiring an additional effort, this model does have several advantages:

- It allows a single Java thread per agent which is quite important especially in environments with limited resources such as cellphones.
- It provides improved performance since behaviour switching is far faster than Java thread switching.
- It eliminates all synchronization issues between concurrent behaviours accessing the same resources since all behaviours are executed by the same Java thread. This also results in a performance enhancement.
- When a behaviour switch occurs, the status of an agent does not include any stack information, implying that it is possible to take a 'snapshot' of it. This allows the implementation of some important advanced features, such as saving the status of an agent in a persistent storage for later resumption (agent persistency), or transferring the agent to another container for remote execution (agent mobility).

It is important to note that a behaviour such as that shown below will prevent any other behaviour from being executed because its `action()` method will never return.

```
public class OverbearingBehaviour extends Behaviour {
  public void action() {
    while (true) {
      // do something
    }
  }
  public boolean done() {
    return true;
  }
}
```

One-shot behaviour, cyclic behaviour and generic behaviours. The three primary behaviour types available with JADE are as follows:

1. *One-shot* behaviours are designed to complete in one execution phase; their `action()` method is thus executed only once.
   The `jade.core.behaviours.OneShotBehaviour` class already implements the done() method by returning true and can be conveniently extended to implement new one-shot behaviours.

   ```
   public class MyOneShotBehaviour extends OneShotBehaviour {
     public void action() {
       // perform operation X
     }
   }
   ```

   In this example, operation `X` is performed only once.

2. *Cyclic* behaviours are designed to never complete; their action() method executes the same operations each time it is called.
   The `jade.core.behaviours.CyclicBehaviour` class already implements the `done()` method by returning false and can be conveniently extended to implement new cyclic behaviours.

   ```
   public class MyCyclicBehaviour extends CyclicBehaviour {
     public void action() {
       // perform operation Y
     }
   }
   ```

In this example, operation `Y` is performed repetitively until the agent executing the behaviour terminates.

3. *Generic* behaviours embed a status trigger and execute different operations depending on the status value. They complete when a given condition is met.

```
public class ThreeStepBehaviour extends Behaviour {
  private int step = 0;
  public void action() {
    switch (step) {
      case 0:
        // perform operation X
        step++;
        break;
      case 1:
        // perform operation Y
        step++;
        break;
      case 2:
        // perform operation Z
        step++;
        break;
    }
  }
  public boolean done() {
    return step == 3;
  }
}
```

In this example, the step member variable implements the status of the behaviour. Operations `X`, `Y` and `Z` are performed sequentially after which the behaviour completes.

JADE also provides the possibility of composing behaviours together to create complex behaviours.

Scheduling operations. JADE provides two ready-made classes (in the `jade.core.behaviours` package) which can be implemented to produce behaviours that execute at selected points in time.

1. The `WakerBehaviour` has `action()` and `done()` methods pre-implemented to execute the `onWake()` abstract method after a given timeout (specified in the constructor) expires. After the execution of the `onWake()` method the behaviour completes.

```
public class MyAgent extends Agent {
  protected void setup() {
    System.out.println("Adding waker behaviour");
    addBehaviour(new WakerBehaviour(this, 10000) {
      protected void onWake() {
        // perform operation X
      }
    });
  }
}
```

   In this example, operation `X` is performed 10 seconds after the `'Adding waker behaviour'` text is printed.

2. The `TickerBehaviour` has `action()` and `done()` methods pre-implemented to execute the `onTick()` abstract method repetitively, waiting a given period (specified in the constructor) after each execution. A `TickerBehaviour` never completes unless it is explicitly removed or its `stop()` method is called.

```
public class MyAgent extends Agent {
  protected void setup() {
    addBehaviour(new TickerBehaviour(this, 10000) {
      protected void onTick() {
        // perform operation Y
      }
    });
  }
}
```

   In this example, operation `Y` is performed periodically every 10 seconds.

1.4 Techniques for checking the system's behaviour

There are two main approaches in order to check the system's behaviour. The first approach discussed is model checking, it is a static technique in bug detection that perform error checking statically, without running the program. Model checking

[37, 73] on the other hand computes the run-time states of the program without running the program. The second approach discussed is runtime verification, which unlike the static verification approaches as model checking it checks the correctness of a program, running the program.

1.4.1 Model checking

Model checking [37] is a technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is automatic and usually quite fast. Also, if the design contains an error, model checking will produce a counterexample that can be used to pinpoint the source of the error. The method, which was awarded the 1998 ACM Paris Kanellakis Award for Theory and Practice, has been used successfully in practice to verify real industrial designs, and companies are beginning to market commercial model checkers.

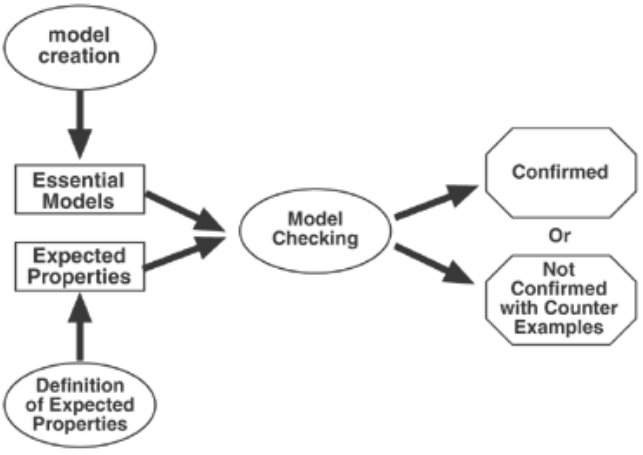


FIGURE 1.1: Model checking procedure

The main challenge in model checking is dealing with the state space explosion problem. This problem occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values. In such cases the number of global states can be enormous. Researchers have made considerable progress on this problem over the last ten years.

Below we report the main advantages and disadvantages in the use of model checking technique [36].

Main advantages of model checking:

- the user does not need to construct a correctness proof;
- it is fast compared to other rigorous methods such as the use of a proof checker;
- if the specification is not satisfied, the model checker will produce a counterexample execution trace that shows why the specification does not hold;
- it has no problem with partial specifications;
- it supports Temporal Logics which can easily express many of the properties that are needed for reasoning about concurrent systems, this is important because reasoning on the concurrency is often quite subtle and it is difficult to verify all possible cases manually.

Main disadvantages of model checking:

- Writing specifications is hard. This is also true for other verification techniques like automated theorem proving (this also occurs in the runtime verification context).
- State explosion is a major problem. The number of global system states of a concurrent system with many processes or complicated data structures can be enormous. All model checkers suffer from this problem. In fact, the state explosion problem has been the driving force behind much of the research in model checking and the development of new model checkers.

1.4.2 Runtime verification

Runtime verification is being pursued as a lightweight verification technique complementing verification techniques such as model checking and testing and establishes another trade-off point between these forces. One of the main distinguishing features of runtime verification is due to its nature of being performed at runtime, which opens up the possibility to act whenever incorrect behaviour of a software system is detected.

We follow [47, 66] and define a software failure as a deviation between the observed behaviour and the required behaviour of the software system. A fault is defined as the deviation between the current behaviour and the expected behaviour, which is typically identified by a deviation of the current and the expected state of the system. A fault might lead to a failure, but not necessarily. An error, on the other hand, is a mistake made by a human that results in a fault and possibly in a failure. According to IEEE [2], verification comprises all techniques suitable for showing that a system satisfies its specification. Traditional verification techniques comprise theorem proving [19], model checking [37], and testing [27, 78]. A relatively new direction of verification is runtime verification, which manifested itself within the previous years as a lightweight verification technique.

Definition 1.1. Runtime verification [66] is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property.

In runtime verification (RV) dynamic checking of the correct behaviour of a system can be performed by a monitor which is generated from a formal specification of the properties to be verified.

As happens for formal static verification, RV relies on a high level specification formalism to specify the expected properties of a system; similarly to testing, RV is a lightweight, effective but non exhaustive technique to verify complex properties of a system at runtime.

In contrast to formal static verification and testing, RV offers opportunities for error recovery which make this approach more attractive for the development of reliable software: not only a system can be constantly monitored for its whole lifetime to detect possible misbehavior, but also appropriate handlers can be executed for error recovery.

Main advantages of runtime verification:

- it ensures that the system may be stopped the moment issues are identified in a tractable manner;
- verification is not invasive, the system running is not affected by the presence of the monitor, this is because the monitor does not need to generate the

traces that have to checked (in this way the state explosion problem, which is typical of the static verification, does not happen);

- verification continues beyond system deployment.

Main disadvantages of runtime verification:

- it cannot prevent a wrong execution to take place, as it only verifies actual, already happened, traces of events.

1.4.3 Runtime verification versus Model checking

Runtime verification [66] has its origins in model checking, and, to a certain extend, the key problem of generating monitors is similar to the generation of automata in model checking. However, there are also important differences to model checking:

- While in model checking, all executions of a given system are examined to answer whether they satisfy a given correctness property φ, which corresponds to the language inclusion problem, runtime verification deals with the word problem.
- While model checking typically considers infinite traces, runtime verification deals with finite executions - as executions have necessarily to be finite.
- While in model checking a complete model is given allowing to consider arbitrary positions of a trace, runtime verification, especially when dealing with online monitoring, considers finite executions of increasing size. For this, a monitor should be designed to consider executions in an incremental fashion.

These differences make it necessary to adapt the concepts developed in model checking to be applicable in runtime verification. For example, while checking a property in model checking using a kind of backwards search in the model is sometimes a good choice, it should be avoided in online monitoring as this would require, in the worst case, the whole execution trace to be stored for evaluation.

Furthermore, model checking suffers from the state explosion problem, which terms the fact that analyzing all executions of a system is typically been carried out by

generating the whole state space of the underlying system, which is often huge. Considering a single run, on the other hand, does usually not yield any memory problems, provided that when monitoring online only a finite history of the execution has to be stored. Last but not least, in online monitoring, the complexity for generating the monitor is typically negligible, as the monitor is often only generated once. However, the complexity of the monitor, i. e. its memory and computation time requirements for checking an execution are of important interest, as the monitor is part of the running system and should influence the system as less as possible.

1.5 LTL

LTL is a modal logic which has been introduced for specifying temporal properties of systems; despite its original main application is static verification through model checking, more recently it has been adopted as a specification formalism for RV, and some RV tools support it [34, 67].

1.5.1 LTL syntax and semantics

Given a finite set of atomic propositions AP, the set of LTL formulas over AP is inductively defined as follows:

- *true* is an LTL formula;
- if $p \in AP$ then p is an LTL formula;
- if φ and ψ are LTL formulas then $\neg\psi$, $\varphi \vee \psi$, $X\psi$, and $\varphi\, U \psi$ are LTL formulas.

Additional operators can be derived in the standard way: $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$, $F\varphi$ (or $\Diamond\varphi$) $= true\, U \varphi$, and $G\varphi$ (or $\Box\varphi$) $= \neg(true\, U \neg\varphi)$.

Let $\Sigma = 2^{AP}$ be the set of all possible subsets of AP; if $p \in AP$ and $a \in \Sigma$, then p holds in a iff $p \in a$. An LTL model is an infinite trace $w \in \Sigma^\omega$; $w(i)$ denotes the element $a \in \Sigma$ at position i in trace w; more formally, if $w = aw'$, then $w(0) = a$, and $w(i) = w'(i-1)$ if $i > 0$.

The semantics of a formula φ depends by the satisfaction relation $w, i \vDash \varphi$ (w satisfies φ in i) defined as follows:

- $w, i \vDash p$ iff $p \in w(i)$;
- $w, i \vDash \neg\phi$ iff $w, i \nvDash \phi$;
- $w, i \vDash \varphi \vee \psi$ iff $w, i \vDash \varphi$ or $w, i \vDash \psi$;
- $w, i \vDash X\varphi$ iff $w, i+1 \vDash \varphi$ (next operator);
- $w, i \vDash \varphi U \psi$ iff $\exists j \geq 0 \;\; w, j \vDash \psi$ and $\forall 0 \leq k < j \;\; w, k \vDash \varphi$ (until operator).

Finally, $w \vDash \varphi$ (w satisfies φ) holds iff $w, 0 \vDash \varphi$ holds.

We recall that the set of all models of LTL formulas is the language of star-free ω-regular languages over Σ [38].

In order to encode an LTL formula into an equivalent trace expression we exploit the result stating that an LTL formula can be translated into an equivalent non deterministic Büchi automaton [17].

1.5.2 Non deterministic Büchi automata

A Büchi automaton is a type of ω-automaton which extends a finite automaton to infinite inputs. It accepts an infinite input sequence if there exists a run of the automaton that visits (at least) one of the final states infinitely often.

A (non deterministic) Büchi automaton (NBA) is a tuple $(\Sigma, Q, Q_0, \delta, F)$, where

- Σ is a finite alphabet;
- Q is a finite non-empty set of states;
- $Q_0 \subseteq Q$ is a set of initial states;
- δ:$Q \times \Sigma \to 2^Q$ is a transition function;
- $F \subseteq Q$ is a set of accepting states.

A run of an automaton $(\Sigma, Q, Q_0, \delta, F)$ on a word $w \in \Sigma^\omega$ is an infinite trace $\rho = q_0 w(0) q_1 w(1) q_2 \ldots$, s.t. $q_0 \in Q_0$, and for all $i \geq 0$ $q_{i+1} \in \delta(q_i, w(i))$. A run ρ is called accepting iff $Inf(\rho) \cap F \neq \emptyset$, where $Inf(\rho)$ denotes the states visited infinitely often.

1.5.3 LTL$_3$

LTL$_3$ is a a three-valued semantics [17] for LTL formulas, devised to adapt the standard semantics to RV, to correctly consider the limitation that at runtime only finite traces can be checked.

Given a finite trace $\sigma \in \Sigma^*$ of length $|\sigma| = n$, a continuation of σ is an infinite trace $w \in \Sigma^\omega$ s.t. for all $0 \leq i < n$ $w(i) = \sigma(i)$.

Given a finite trace $\sigma \in \Sigma^*$, and an LTL formula φ, the LTL$_3$ semantics of φ, denoted by $\sigma \vDash_3 \varphi$, is defined as follows:

$$\sigma \vDash_3 \varphi = \begin{cases} \top & \text{iff } w \vDash \varphi \text{ for all continuations } w \text{ of } \sigma \\ \bot & \text{iff } w \nvDash \varphi \text{ for all continuations } w \text{ of } \sigma \\ ? & \text{iff neither of the two conditions above holds} \end{cases}$$

As an example, let us consider the formula $\varphi = p\,U\,q$, where $p, q \in AP$; according to the definition above, $\{p\}\{p\}\{q\} \vDash_3 \varphi = \top$, that is, φ is satisfied by the finite trace $\{p\}\{p\}\{q\}$, and monitoring succeeds; $\{p\}\{p\}\emptyset \vDash_3 \varphi = \bot$, that is, φ is not satisfied by the finite trace $\{p\}\{p\}\emptyset$, and monitoring fails; finally, $\{p\}\{p\}\{p\} \vDash_3 \varphi =?$, that is, at this stage monitoring is inconclusive, and the monitor has to keep monitoring the property expressed by φ. Assuming that $AP = \{p, q\}$, the LTL$_3$ semantics of $p\,U\,q$ corresponds to the finite state machine (FSM) defined in Figure 1.2, which fully determines the expected behaviour of a monitor for the RV of $p\,U\,q$.

More in general, for all LTL formulas φ, it is possible to build an FSM which is a deterministic finite automaton (DFA) where the alphabet is Σ (that is, 2^{AP}), all states are final, each state returns either $\top$ (successful), or $\bot$ (failure), or ? (inconclusive), and the behaviour of the FSM respects the LTL$_3$ semantics of φ: for all finite traces $\sigma \in \Sigma^*$, the FSM accepts σ with final state that returns $v \in \{\top, \bot, ?\}$ iff $\sigma \vDash_3 \varphi = v$.

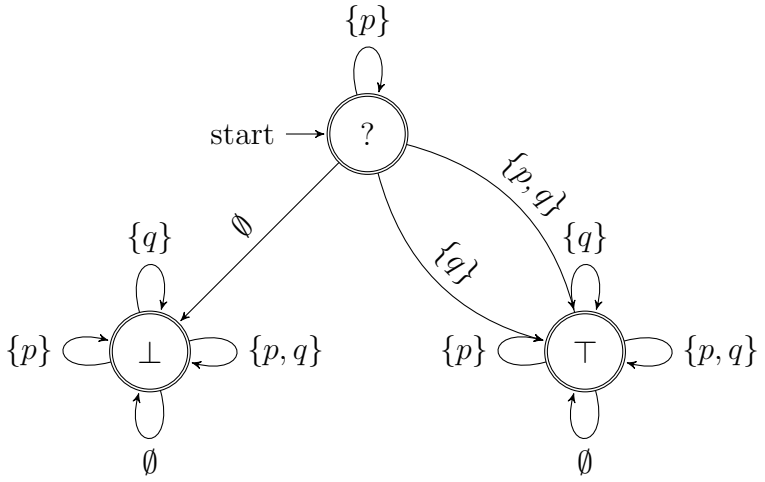


FIGURE 1.2: FSM of the monitor for pUq, with $AP = \{p, q\}$

The sequence of steps required to generate from an LTL formula φ an FSM that respects the LTL_3 semantics of φ [17] is summarized in Figure 1.3

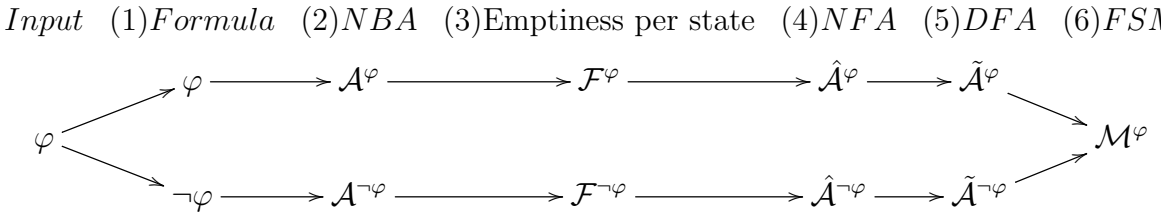


FIGURE 1.3: Steps required to generate an FSM from an LTL formula φ

For each LTL formula φ and $\neg\varphi$ (1), the equivalent NBAs $\mathcal{A}^{\varphi}$, and $\mathcal{A}^{\neg\varphi}$ are built (2), all states that generate a non empty language are identified (3) and made final and the NBAs are transformed into the corresponding NFAs $\hat{\mathcal{A}}^{\varphi}$, and $\hat{\mathcal{A}}^{\neg\varphi}$ (4), and, then, in the equivalent DFAs $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ (5). Finally, the product of $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ is computed, and from it the final FSM $\mathcal{M}^{\varphi}$ is derived by minimization, and by classifying the states in the following way: (q, q') returns $\top$ iff q' is not final in $\tilde{\mathcal{A}}^{\neg\varphi}$, $\bot$ iff q is not final in $\tilde{\mathcal{A}}^{\varphi}$, and ? if both q and q' are final in $\tilde{\mathcal{A}}^{\varphi}$, and $\tilde{\mathcal{A}}^{\neg\varphi}$, respectively.

1.6 Constrained Global Types

Constrained global types [3, 7, 71] are behavioral types for specifying and verifying multiparty protocols involving many distributed components, inspired by the

process algebra approach.

In [3] constrained global types have been used to generate monitor agents which were able to check the behaviour of other agents inside the system. Given a protocol expressed using constrained global types, monitors are able to verify the compliance of agents to the protocol[3].

Before explaining in greater detail the syntax and the semantics of constrained global types we must introduce two key concepts: *events* and *event types*.

Events. An event e is any observable event taking place in the MAS environment, including communicative actions, actions performed by agents, agent location and moves, and actions performed by artifacts. We do not face the transduction problem and assume that events are translated into symbols that agents can manipulate by some mediator between the agents and the environment.

Event types. From a logical point of view, an event type ϑ is a predicate on events. Its interpretation is the set of events that verify ϑ; we write $e \in \vartheta$ to mean that ϑ is true on e, and we also say that e has type ϑ.
We can better explain the event types with an example:

```
transport(policeman(marcus), prisoner(alice),
from(jail), to(room1), by(car))
∈ move(alice, jail, room1).
```

With respect to the actual event that took place in the environment and that was transduced into a symbolic form, the event type may an be identified by a different functor symbol with different arguments (like in example, where "move(...)" is the event type of a "transport(..)" event) and may abstract some details which are not relevant for the monitoring activities.

1.6.1 Syntax

The protocol specification using constrained global types represents a set of possibly infinite traces of events and is defined on top of the following operators:

[3] More details are given in Chapter 3.

• λ (*empty trace*), representing the singleton set $\{\varepsilon\}$ containing the empty trace ε of events.

• $\vartheta^n{:}\tau$ (*sequence with producer*), representing the set of all traces whose first event e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace in the set represented by τ. The integer n specifies the least required number of times $e \in \vartheta$ has to be "consumed" to allow a transition labeled by e. Each occurrence of a producer event type must correspond to the occurrence of a new event; in contrast, consumer event types correspond to the same event specified by a certain producer event type. The purpose of consumer event types is to impose constraints between branches of the fork operator, *without introducing new events*.

• $\vartheta{:}\tau$ (*sequence with consumer*), representing the set of all traces where $e \in \vartheta$, and the remaining part is a trace in the set represented by τ. ϑ must match with a producer ϑ^n event type available in another fork branch of the protocol.

• $\tau_1{+}\tau_2$ (*choice*), representing the union of the traces of τ_1 and τ_2.

• $\tau_1 | \tau_2$ (*fork*), representing the set obtained by shuffling the traces in τ_1 with the traces in τ_2.

• $\tau_1 \cdot \tau_2$ (*concat*), representing the set of traces obtained by concatenating the traces of τ_1 with those of τ_2.

Constrained global types are regular terms, that is, can be cyclic (recursive), and hence they can be represented by a finite set of syntactic equations. To make the treatment simpler, we limit our investigation to *contractive* (a.k.a. *guarded*) and *deterministic* constrained global types. A constrained global type τ is *contractive* if all infinite paths[4] in τ contain an occurrence of the ":" constructor. Determinism ensures that dynamic checking can be performed efficiently without backtracking. Intuitively, a constrained global type is deterministic if, in case more transition rules can be applied when event e takes place, they lead to equivalent constrained global types.

[4]By "path of a constrained global type" we mean "path in the possibly infinite tree corresponding to the term that represents the constrained global type".

1.6.2 Semantics

The state of a constrained global type τ can be represented by τ itself. In this section, when talking about constrained global types we will refer to their current state. Also, we will use "constrained global type" and "protocol" interchangeably.

The interpretation of a constrained global type is based on the notion of transition, a total function

$next : \mathbb{N} \times Pr \times Event \rightarrow \mathcal{P}_{fin}\ (Pr \times \mathbb{N})$,

where Pr and $Event$ denote the set of contractive and constrained global types and of events, respectively. The first argument of $next$ is the number used by the *sequence with producer* operator, it is necessary when the protocol requires synchronization among some events.

If τ_1 represents the current state of the protocol and the event e takes place, then the protocol can move to τ_2 iff $next(0, \tau_1, e) = (\tau_2, 0)$, that we write as $\tau_1 \overset{e}{\rightarrowtail} \tau_2$.

The auxiliary function $\epsilon(\_)$ specifies the constrained global types whose interpretation contains the empty sequence ϵ. Intuitively, a constrained global type τ s.t. $\epsilon(\tau)$ holds specifies a protocol that is allowed to successfully terminate.

Let τ_0 be a contractive and constrained global type. A *run* ρ for τ_0 is a sequence $\tau_0 \overset{e_0}{\rightarrowtail} \tau_1 \overset{e_1}{\rightarrowtail} ... \overset{e_{n-1}}{\rightarrowtail} \tau_n \overset{e_n}{\rightarrowtail} \tau_{n+1} \overset{e_{n+1}}{\rightarrowtail} ...$ such that (1) either the sequence is infinite, or it has finite length $k \geq 0$ and the last constrained global type τ_k verifies $\epsilon(\tau_k)$; and (2) for all τ_i , e_i , and τ_{i+1} in the sequence, $\tau_i \overset{e_i}{\rightarrowtail} \tau_{i+1}$ holds.

We denote by $\mathrm{A}(\rho)$ the possibly empty or infinite sequence of events $e_0 e_1 ... e_n ...$ contained in ρ. The interpretation $[\![\tau_0]\!]$ of τ_0 is the set $\{A(\rho) | \rho$ is a run for $\tau_0\}$. A contractive constrained global type τ is deterministic if for e any possible run ρ of τ and any possible τ' in ρ, if $\tau' \overset{e}{\rightarrow} \tau''$, and $\tau' \overset{e}{\rightarrow} \tau'''$, then $[\![\tau'']\!] = [\![\tau''']\!]$.

Chapter 2

Related work

Our work falls both in the research area on self-adaptive systems and on runtime verification (runtime monitoring) of MASs.

Self-adaptive systems:
The self-adaptive systems spun off from the wider area of distributed systems, be them based on web services, software agents, robots, or on other autonomous entities that need to react to unforeseen changes during their execution. Many surveys have been conducted to identify the main features of self-adaptive MASs [49, 57, 83, 91, 94] and interesting and original solutions have been proposed by the research community.

Proposals for standardizing the concepts involved in the self-adaptation process include [65], a meta-model to describe intelligent adaptive systems in open environments, and [29], a taxonomy of adaptive agent-based collaboration patterns, for their analysis and exploitation in the area of autonomic service ensembles. An analysis of linguistic approaches for self-adaptive software is presented in [85].

As far as the engineering of self-adaptive systems is concerned, authors either propose new methods and platforms or extend already existing methodologies. In the first category we mention the AMAS theory (Adaptive multiagent Systems [30]), which identifies design criteria to enable the emergence of an organization within the system and to guarantee the global function of the system even in critical situations, Unity [88], a decentralized architecture for autonomic computing based on multiple interacting agents, and DSOL [43], a declarative approach supporting dynamic service orchestration at run-time. As good representatives of the second

category we may mention [33], based on Gaia [95], and [44, 77], based on Tropos [26].

The approaches closer to ours focus on formalizing protocols that the agents may use during their life, including specific protocols to deal with unforeseen events: in these approaches agents are usually free to choose, from a bunch of usable protocols, which one they prefer, maintaining in this way the freedom to autonomously self adapt to the new situation but ensuring at the same time that a feasible interaction pattern is followed. Our work can be included in this research field, where we can speak of "protocol enforcement" or "protocol-driven agents". There are at least three ways to obtain a protocol-driven behaviour:

1. automatically generating the agent's code from the protocol specification in order to ensure the compliance by construction;

2. monitoring the agents interactions to identify a violation and enforcing a recovery;

3. making agents capable of directly executing or interpreting the formal protocol specification.

Proposals following the first approach received the attention from the community since the Agent-Oriented Software Engineering early days. The PASSI methodology [40] allows the designer to generate the agent's structure and its internal code starting from UML models and a similar functionality is offered by Prometheus [80]; West2East [31] offers libraries for both the translation of protocols represented in FIPA AUML[1] into different textual notations, and the automatic generation of an executable program compliant to the original interaction protocol; Dsml4Mas [89] can be used to design protocols following a model-driven approach for generating executable code from protocol specifications [60].

The second approach is discussed for example in [28], where a mechanisms for monitoring norms is proposed, in [59], where the global-level adaptation is based on the monitoring of the system's behaviour and on a dynamic reification of an organizational structure, and in [46], where a form of agent supervision, which constrains the actions of an agent so as to enforce certain desired behavioral specifications, is presented.

[1] `www.auml.org/auml/documents/ID-03-07-02.pdf`.

We followed the third approach, namely allowing agents to directly executing or interpreting the protocol. The related literature we are aware of is all centered around protocols conceived as first class entities and represented by means of commitments. For example in [12] commitment protocols can be directly accessed by the agents, as they are artifacts available in the platform, whereas in [93] agents use a "commitments plus axioms" protocol representation that enables them to flexibly accommodate the exceptions and opportunities that may arise at run time. A framework for modeling and handling exceptions is presented in [69] and in successive works by Singh et al. [56, 87], where a formal methodology is proposed to manage, even at run time, expected and unexpected exceptions in commitment-based MASs. Although these approaches are close to ours in spirit, commitment protocols are definitely different from constrained global types, which are inspired by global and multiparty session types [32] and include no notion of commitment. Also, extending existing agent environments or languages by implementing our framework on top of them should be always possible as long as these environments meet the few – and almost mandatory for any MAS. Implementing an approach based on commitment protocols, instead, requires a paradigm shift, where agents are designed and implemented adhering to the commitment approach.

In [74–76] interaction protocols are modeled as executable entities that can be referenced, inspected, composed, shared, and invoked between agents. The $\mathcal{RASA}$ formalism defined in those papers is close to ours, even if its expressiveness is lower as no fork operator is supported. Also, no exploitation of the framework on top of real agent systems is presented.

As far as self-adaptiveness of protocol-driven agents is concerned, the main sources of inspiration were [35, 39, 81, 82]. In [35] the authors propose a dynamic self-monitoring and self-regulating approach based on norms to express properties which allow agents to control their own behaviour. In [81] and [82] agents operating in open and heterogeneous MASs dynamically select protocols, represented in FIPA AUML, in order to carry collaborative tasks out. Since the selection is performed locally to the agent, some errors may occur in the process. The proposed mechanism provides the means for detecting and overcoming them.

Our work is similar to [39] both in the formalism used to represent protocols and in the idea of dynamic protocol switch, but our solution is actually implemented and thus must take many more practical aspects - for example what to do before switching to a new protocol, how to state when the agent is in a safe state and

can actually perform a protocol switch - into account.

Comparison. To the best of our understanding, none of the mentioned approaches covers the three stages of starting from a global description of the protocol, moving to local versions for the individual agents via projection, and interpreting these local versions on an actual agent framework to drive the agents' behaviour, as we do. One reason why our approach is different from others, is that the projection function takes protocol specifications and returns protocol specifications expressed in the same language. Usually, projection functions return either agent stubs/code (common in the MAS community) or protocol specifications in a language suitable for expressing the agent local viewpoint, different from the language for expressing the global one (common in the session types community). Having a unique formalism for protocol specification both at the global and at the local level is a simpler and more uniform approach.

In the MAS area, when the code of an agent is generated from a protocol specification, the language used for specifying the interaction protocol and the language used for implementing the agents are, again, different. The agent is "compiled" into some AOP language starting from the protocol's specification. On the contrary we do not generate any agent code into any AOP language. The protocol specification is interpreted and this gives the flexibility that meta-programming ensures, demonstrated for example by the easiness in implementing protocol switch.

Finally, w.r.t. our own previous work, in [3] and [8] (resp. [4]) we proposed to exploit the interaction protocol formalism (resp. the protocol projection mechanism) as a way to face (resp. to make more efficient) the runtime verification of protocol compliance via monitoring. Also, issues like self-adaptiveness, protocol driven agents, protocols as first class entities and protocol switch are not addressed by [3, 4, 8]. This thesis and the previous papers share the adoption of the same formalism (constrained global types) and tools (projection and `next` functions), but used for definitely different purposes.

Runtime monitoring of distributed systems:
Considering that our work is mainly focused on protocol-driven systems, the analysis in the runtime monitoring context is much less detailed.

As analyzed in [58], the most research in runtime monitoring has focused on monolithic software as opposed to distributed systems. There has been some research in monitoring distributed systems; Mansouri-Samani and Sloman overview this research area (up to 1992) [70].

Bauer et al. describe a distributed system monitoring approach for local properties that require only a trace of the execution at the local node [16]. Each node checks that specific safety properties hold and if violated, sends a report to a centralized diagnosis engine that attempts to ascertain the source of the problem and to steer the distributed system to a safe state. The diagnosis engine, being globally situated, collects the verdict from observations of local traces and forms a global view of the system to render a diagnosis of the source of the error.

Bhargavan et al. [20, 21] focus on monitoring distributed protocols such as TCP. They employ black-box monitors that have no knowledge of the internal state of the executing software, which is the same approach followed by our framework. Their monitors view traffic on the network, mimic the protocol actions in response to observed input, and compare the results to outputs observed on the network. A domain specific language, Network Event Recognition Language (NERL), is used to specify the events on the network that they wish to monitor and a specialized compiler generates a state machine to monitor for these events [21].

In Fornara et al. [54, 55, 79] the authors discuss their approach based on Normative MAS. An artificial institution catches the institutional events and verifies them with respect to a normative specification. As a result, protocol specifications are a special case with respect to a normative specification. So, even if the approach is different the aim is similar, i.e. to deal with open multiagent systems and monitor their correctness w.r.t. a specification.

Normative system approaches offer other advantages for multi agent systems because agents may integrate their practical reasoning with reasoning about the normative specification, although, also our protocol-driven agents could reason about trace expressions which are a First Class Entities.

Other closely related proposals are those by Criado et al. [42] and Baldoni et al. [13–15, 68]; in these papers, the authors suggest a way to implement a monitoring mechanism by exploiting the A&A metamodel and by reifying commitment-based protocols into artifacts. The proposal is implemented both on top of Cartago and Jade and on top of Jason/JaCaMo. However, our work is different from theirs,

because – at least from the runtime verification application – our approach is more generic, in fact, it works with each possible MASs architecture and not with only certain customized implementations.

Chapter 3

The framework

In the previous chapters, we have already discussed the importance of runtime verification of a MAS (that has always the peculiarity of being a distributed and complex system).

In this chapter, we illustrate our framework without going into technical details. It consists of three different building blocks that, if properly assembled, allow us to realize systems on which the verification process is more and more distributed and can be carried out earlier.

In the following sections, we present the high-level framework[1], starting from the less distributed setting, where one monitor checks all other agents and arriving to the most distributed setting, where each agent has its own monitor. The most distributed and safe setting, where verification takes place in such an early stage of the process that executing bad actions is even prevented, is the one where the monitor is fully integrated within the agent, and checks what the agent can do before it actually does it.

3.1 Building blocks

In order to describe our framework, we start with introducing its three building blocks:

[1]Without dwelling on all technical aspects which will be discussed later in the document.

- the *formalism*, that represents a protocol which describes a pattern of observable events;
- the *next* transition function that, given an observed event, defines how to move from a protocol state to the next one (or, in case it is used to generate, and not to verify, it defines the events which are allowed and where it leads inside the protocol states);
- the *project* function that, given a global protocol and a subset of agents, returns a new protocol corresponding to a local view which involves only the subset of selected agents.

3.1.1 The formalism

The formalism, as its name suggests, is the language which allows us to formulate our protocols. There are many formalisms that can be used; in particular we can summarize them with:

- constrained global types,
- trace expressions and
- template trace expressions.

The first one was presented in Section 1.6 and the last two will be presented in Chapter 4.

Intuitively, the formalism represents the state of a multiagent protocol from which several transition steps to other states (that is, to other states of the protocol represented with the same formalism) are possible, with a resulting event.

When we speak about a protocol, unless it is stated otherwise, we consider a *global view* where the behaviour of all agents must be specified.

3.1.2 The next transition function

As already highlighted in Section 1.6, given a state representing the current protocol situation, we can move to other states. Compared to the *next* function introduced in Section 1.6, we present a high-level version omitting all aspects which are

related to a particular formalism (the number used for synchronizations is present only in the constrained global types formalism).

The function which allows us to do this is the *next* transition function, which is defined as:

next: $Pr \times Event \rightarrow \mathcal{P}_{fin}(Pr)$

where:

- Pr is the set of protocols expressed using one of the formalisms introduced in the previous section;
- *Event* is the set of events which can be observed (or generated) inside our system.

The *next* transition function takes a protocol and an observed $event \in Event$[2] and returns a new set of protocols[3]

To make some examples[4]:

$next(\alpha{:}Pr_1, a){=}Pr_1$ if $a \in \alpha$[5]
$next(Pr_1{+}Pr_2, a){=}Pr_3$ if $next(Pr_1, a){=}Pr_3$ or $next(Pr_2, a){=}Pr_3$

It is important to note that Pr represents a set of global protocols and given a global protocol, the *next* transition function returns another global protocol. In order to make the process more distributed, we must introduce the last building block of our framework.

3.1.3 The project function

With a global protocol, we can represent the behaviour of each agent inside the system. In this way, if we only have one monitor which checks the entire system, it is correct that all the information collapses in the protocol representation (which

[2]As already said, an event can be an action, a message, etc. In our examples however, for practical reasons, all events correspond to a message exchange.

[3]In this way, given a well-defined protocol and an observed (or generated) event, the *next* transition function is nondeterministic; this will become clearer in Chapter 4.

[4]Here we omit all the formal details that will be widely analyzed in Chapter 4.

[5]In Section 1.6 we have already introduced *events* and *event types*; in this case we wonder if the event a belongs to the *event type* α.

encodes what all agents can or cannot do). However, in a MAS the distribution of the runtime verification is extremely important and it is necessary to split up the workload among multiple monitors.

Given a global protocol, the *project* function returns a local view of it. If we take, for instance, a global protocol on the set of agents $\{\text{agent}_1, \text{agent}_2, \text{agent}_3\}$, it allows us to represent all event traces involving agent_1, agent_2 and agent_3. We can use the *project* function passing the protocol and the subset $\{\text{agent}_1, \text{agent}_3\}$ in order to obtain a local view of the protocol representing only the event traces involving at least agent_1 or agent_3. It is important to note that the event traces represented by the projected protocol do not involve only the agents passed to the *project* function. For example, if we have an event involving agent_1 and agent_2, it would still be present in the projected protocol.

The *project* function is defined in the following way:

project: $Pr \times \mathcal{P}(Agents) \rightarrow Pr$

The *project* function allows us to distribute the protocol among more monitors instead of only one, as will become clearer in the following.

3.2 Playing LEGO with the building blocks

In the previous sections we presented our framework from a more abstract point of view. In this section, instead, we analyze in more detail some possible ways of composing the framework's building blocks.

We will study in more depth our settings from the simplest one, which is fully centralized and where faults are detected after they took place, to the most complex one, which is fully distributed and in which faults cannot take place. We explain the main differences between the *runtime verification* of a MAS and the creation of a MAS where each agent is *correct by construction*.

3.2.1 Centralized runtime verification

The initial goal of our work [3, 8] was testing whether the actual events in the MAS were compliant with the protocol specification. To define the operational

semantics of the protocol, and to implement the monitoring activity, we used the *next* transition function already defined.

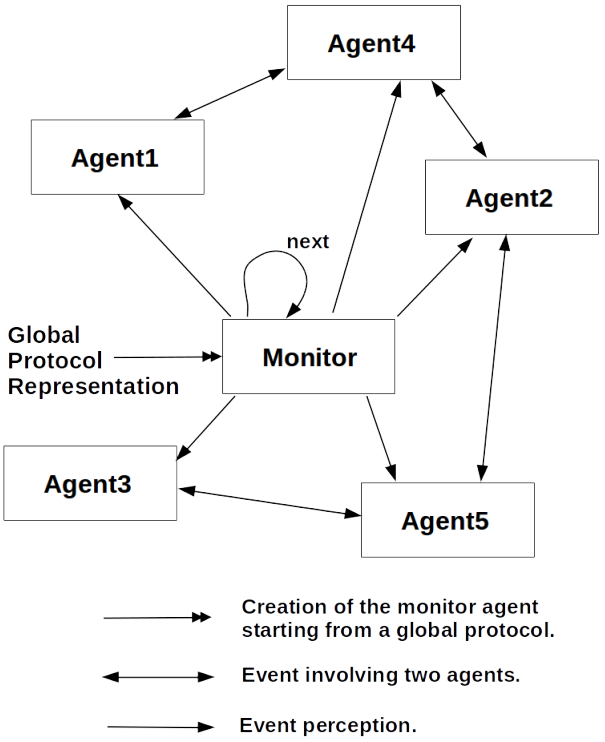


FIGURE 3.1: Runtime verification of a MAS using a monitor agent

All checks are made by a privileged agent called *monitor*; this is the only one which executes the *next* transition function observing all agents events (it is important to note that the *next* transition function, in this case, is called on the protocol representing a global view of the system status). From a more technical point of view, all agents can send and receive all messages[6] they want, thus the monitor can only observe events that have already occurred.

Unfortunately, this solution raises two problems: *bottlenecks* and *lack of distribution*.

If we have only one agent which verifies that the observable part of agent behaviour respects the pattern indicated by the protocol, we risk falling in a bottleneck problem; indeed, in a distributed system with N agents and 1 monitor, there is a risk of creation of a lot of traffic on the monitor caused by the high flow of events to verify. In particular, we can also note that this solution is not reliable, because

[6]We must remind that a message is a communicative event which is a particular case of event.

there is only one agent which checks all events; if the monitor dies, the entire system keeps on running with no agent verifying the event flow.

The lack of reliability and risk of deadlocks resulting from this approach can be addressed by adding more monitors to our system.

3.2.2 Distributed runtime verification

In order to solve the problems of lack of reliability and emergence of deadlocks, we can add more than one monitor to the system. In this way, each monitor must only check a subset of all agents.

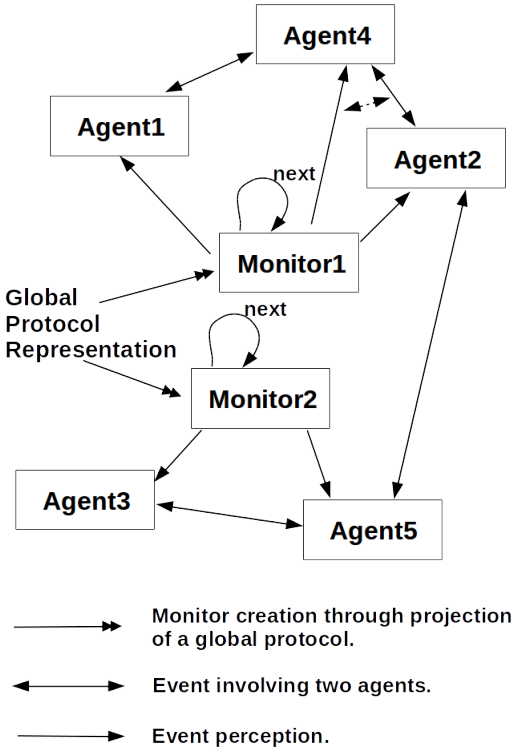


FIGURE 3.2: Runtime verification of a MAS using two monitors agent

The main difference from the single monitor approach lies in the addition of the projection phase. As already explained in Section 3.1.3, using the *project* function, we can obtain a local view starting from a global view of a protocol; this allows us to split the workload of 1 monitor among N monitors.

The main problem of this setting is the choice of how to split the set of agents into subsets, each of them checked by a different monitor[7].

What if we stress even more on distribution?

3.2.3 Ultra-distributed runtime verification

In order to make our implementation as distributed as possible we can simply add a monitor for each agent.

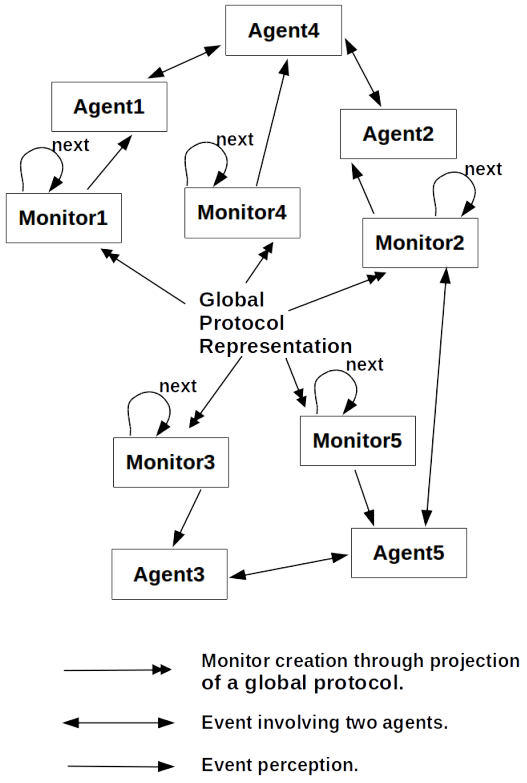


FIGURE 3.3: Runtime verification of a MAS using one monitor for each agent

Unfortunately, this solution is not efficient enough; when we have only one monitor, it has to verify the entire system (a great deal of work) but, if we have one monitor for each agent, it does not have many events to observe (a very small amount of

[7]Problem on which we do not linger because it is out of the scope of our work.

work). In particular, there is also the risk of checking the same event multiple times unnecessarily[8].

In spite of the inefficiency of this approach, this setting allows us to note a very important aspect; we use a monitor to check a single agent having a local view of a global protocol representation.

What if we collapse the monitor into the agent it checks?

3.2.4 Protocol-Driven agents

The penultimate setting of our framework consists in the absence of all monitors, bringing the protocol inside each agent by projection. In this way, each agent can follow the protocol by itself and it is correct by construction (it is guided by the protocol).

We call this particular kind of agents *protocol-driven agents*.

In protocol-driven agents, the purpose of the protocol is no longer to test the acceptability of an event which already took place (like in the monitor setting), but rather to generate all the events which are allowed in the current protocol state in order to drive the agent's behaviour. This goal can be easily achieved by generating all the events which are solutions allowed by *next*, given the current state of the protocol.

We define a *generate* function

generate: $Pr \rightarrow \mathcal{P}_{fin}(Event \times Pr)$

which takes one protocol Pr_0 and returns the set of couples of events and protocols, $\{ (Event_1,Pr_1), (Event_2,Pr_2), ..., (Event_n,Pr_n) \}$ such that $Pr_i \in next(Pr_0, Event_i)$ for all i such that $1 \leq i \leq n$.

In order to better explain the *generate* function, we have to decrease the level of abstraction, as it has already happened, considering the communicative events (messages) among the agents instead of generic events. Given an agent Ag which

[8] In the case with only one monitor, if an agent sends a message to another agent, the monitor checks if the message is compliant with the protocol once; but, in the case with a monitor for each agent, if an agent sends a message to another agent, this message is checked by both the sender's monitor and the receiver's monitor, even if only one check would be enough.

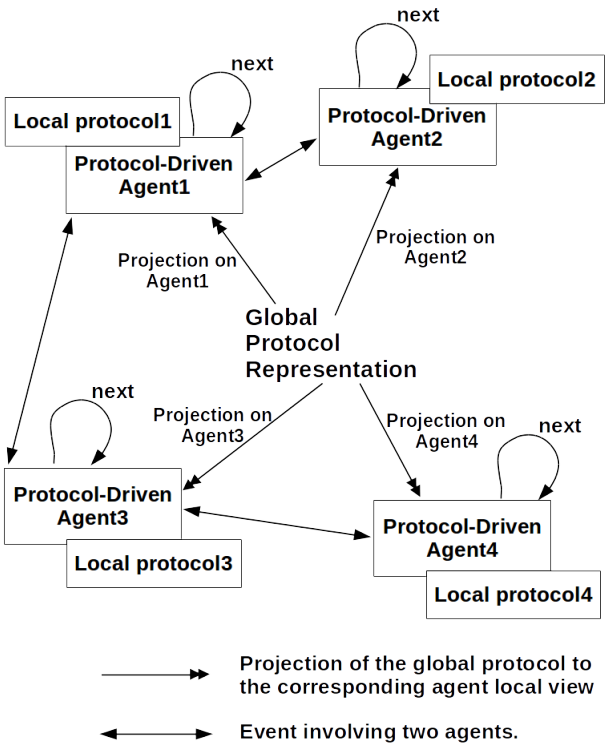


FIGURE 3.4: MAS where each agent is guided by a protocol and it is not necessary to check anything at runtime

calls *generate*, we name *InMsgs* the messages returned by *generate* where *Ag* is the receiver, and *OutMsgs* the messages returned by *generate* where *Ag* is the sender. Intuitively, *InMsgs* are the messages that *Ag* expects in that protocol's state (it cannot decide which one among them to receive, but it can verify that the received message is one of them), and *OutMsgs* are the messages that *Ag* can decide to send. All the messages returned by *generate* are associated with the state where the protocol would move if that event took place, in order to properly update the protocol's state.

Summarizing, all the settings can use the same protocol formalism. In the previous settings the monitor was the only agent which tracked the current state of the protocol and that checked events using the *next* transition function; in this solution, instead, a projection phase initially occurs for each agent inside the system and then all agents have a local version of the current state of protocol and they can call the *generate* function by themselves.

Advantages of the protocol-driven implementation. The protocol-driven approach brings the following advantages:

- safety by construction;
- good level of distribution of the verification process;
- absence of bottlenecks caused by the verification process;
- reliable setting.

Regarding to the first point; if a description of the global protocol, which all the agents in the MAS must respect, exists, the local protocol for each agent can be automatically obtained from the global one. This allows the protocol designer to concentrate on the global features of the MAS only, rather than implementing one local protocol for each agent, and the whole MAS to respect the global protocol by construction.

In the monitor setting, if a monitor dies, all the agents that it checks remain without an agent which controls them. Instead, in the protocol-driven setting, there are no monitors verifying the events which are compliant with the protocol; if an agent dies, all other agents continue to follow the protocol.

Regarding the last point, to better support the possibility of no response from an agent, we should extend the formalism introducing some concepts about timeouts. A work [6] which analyzes in detail the use of trace expressions with an exception handling extension exists[9] and it allows us to obtain a more reliable setting.

In the following section we can describe more in depth the last framework setting which uses self-adaptive protocol-driven agents.

3.2.5 Self-Adaptive Protocol-Driven Agents

The last setting of our framework consists in a further extension of the protocol-driven setting introducing the self-adaptiveness. A self-adaptive system "*is often centralized and operates with the guidance of a central controller or policy, assesses*

[9]The cited paper analyzes an extension of the formalism which implements the exception treatment in a runtime verification context similar to the monitor context which we have already met.

its own behaviour in the current surroundings, and adapts itself if the monitoring and analysis warrants it. Such a system often operates with an explicit internal representation of itself and its global goals" [92].

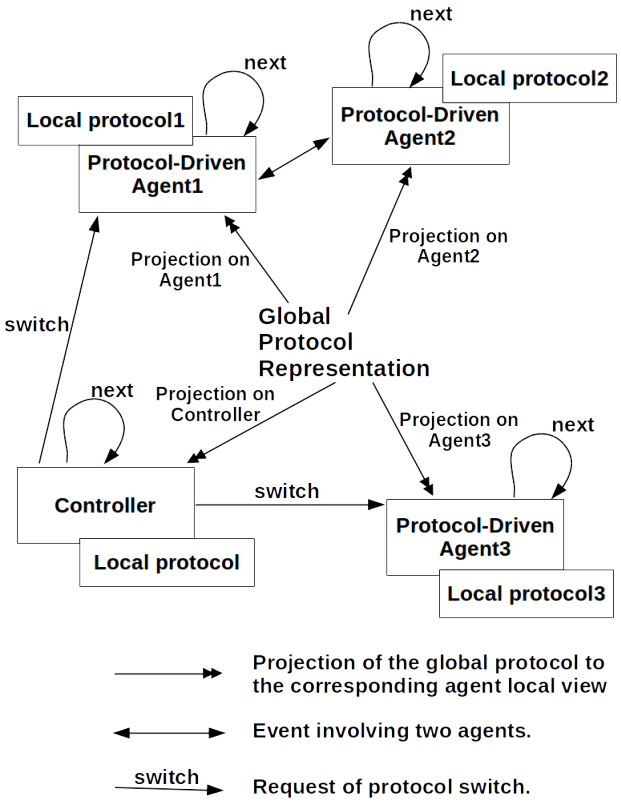


FIGURE 3.5: MAS where each agent is guided by a protocol and it is not necessary to check anything at runtime and where an agent can request a protocol switch to another agent.

The first observation to make on this setting is the presence of a special agent called *controller*; this particular agent has the power to require a protocol switch to other agents. When an agent receives a request for a protocol switch, it follows these steps:

1. it checks if the request of protocol switch was sent by a controller and not by an agent without privileges;
2. it gets the global version of the protocol it must switch to;
3. it projects the global protocol in order to obtain a local version involving only itself;
4. it follows the new projected protocol.

Unlike in the setting analyzed in Section 3.2.1 where only one agent (the monitor) had to verify all events observed inside the system, in this framework setting, there are no bottleneck problems. We can justify this assertion taking into account these two observations:

- The controller does not communicate with all the agents but only with a subset of them; consequently, an increase in the number of agents within the system does not imply an increase in the communications among the controller and the other agents.
- The controller workload is lower than the monitor workload. This is due to the low frequency of the protocol switch requests: indeed, during the agents' life the protocol switch requests do not occur frequently but only at certain points of the protocol.

In this setting of the framework, the agents able to adapt are "self-adaptive protocol-driven": they are characterized by one interaction protocol specified in one of the formalisms previously introduced (in Chapter 4 there are more details about the last version of our formalism) and by three mandatory components, the *knowledge base*, the *message queue* and the *environment's representation*, that should be directly implemented in the underlying agent tool (as we will analyze in greater detail in Section 6.1). As we make no other assumption on the agent's architecture, our framework is as general as possible and could be implemented in any underlying environment or programming language where these three components are available (namely almost all tools to program MAS, be they BDI-oriented like Jason [22], or not BDI-oriented like JADE [18]).

Besides self-adaptive protocol-driven agents, "normal" agents entirely implemented in the underlying tool can also be part of the self-adaptive MAS. Normal agents give the possibility to reuse existing code and to implement behaviours that cannot be conveniently modeled using an event protocol, for example because they must access legacy code or perform complex computations. However, no hypotheses can be made on their adaptability and reliability.

Being self-adaptive protocol-driven means that the agent behaves according to a given protocol. In each time instant, the self-adaptive protocol-driven agent can make only those internal choices which are allowed by the protocol in the current state. In case of events which depend on external choices, the agent can only verify

if the event that took place is compliant with the protocol and act consequently. Events could be in principle of any kind, but in order to demonstrate the feasibility of our approach in a neat way, in this document we limit ourselves to consider communicative events, where sending events are the result of an internal choice, whereas reception events are not under the agents' control and can be neither prevented nor forced, but only checked.

Adaptation takes place when the self-adaptive protocol-driven agent switches to a new protocol on request of the controller agent, which might follow a MAPE-like loop [63, 92]. In this document we make no assumptions on the controller's internal architecture and functioning. For our purposes, the controller is just an agent which has the power to request protocol switches to some self-adaptive protocol-driven agents and which may or may be not self-adaptive protocol-driven, depending on which type of behaviour it implements and which requirements it must meet.

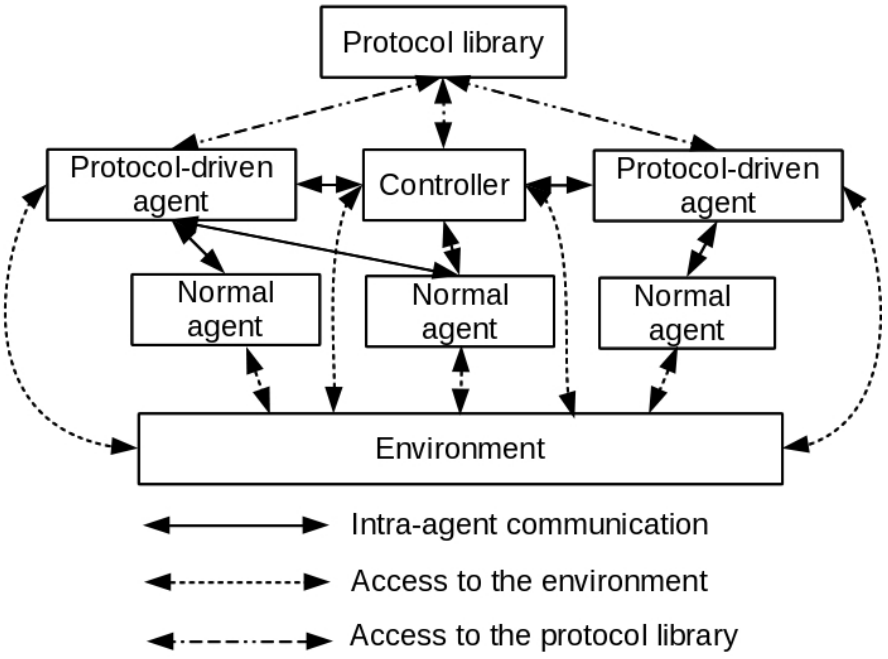


FIGURE 3.6: Architecture of a top-down, centralized self-adaptive MAS.

The components of this setting are shown in Figure 3.6. The protocol library may be either external, like in the figure, or hard-wired in the controller's knowledge base. From a logical viewpoint this makes little difference. In the first case, when the controller identifies which protocol the agents must follow to adapt to a new situation, it communicates the protocol's identifier to the agents which will

retrieve its representation from the library. In the second case, it will send the full protocol's representation inside the message's content.

Protocols in the library always take a global perspective (namely, a perspective where all the parties involved in the protocol are managed in a homogeneous way, without taking the point of view of one of them), but may involve a subset of the agents in the MAS: the controller may send different protocols to different subsets of agents, even if this requires a careful design of such protocols to avoid unwanted interferences.

Self-adaptive protocol-driven agents interact with all the agents, including the controller and normal ones, via message passing. This requires that the controller and normal agents are modeled in the protocol, making self adaptive protocol-driven agents aware of them.

Being self-adaptive protocol-driven does not contrast with the main agents' features identified by N. R. Jennings, K. P. Sycara and M. Wooldridge in [64] and already presented in Section 1.1:
– *Situatedness* is achieved by setting the agent's policies defining how to select a message to send among the possible ones and what to do when a message allowed by the protocol is received. These policies take information coming from the environment into account and can affect it as a side effect.

– *Autonomy* is preserved because protocols usually define different allowed patterns of events, without imposing the choice of which of them following: the choice is left to the agent, thus balancing its respect of the protocol and its autonomy.

– *Responsiveness* and *proactiveness* depend on the protocol itself. For example, a protocol describing the events between Alice and Bob, where Alice always sends a message a to Bob and Bob always receives it and does nothing, leaves room neither for responsiveness in Alice behaviour, nor for proactiveness in Bob's one. It is up to the protocol designer to cope with these issues in a proper way.

– *Sociality*, namely the ability to interact, when appropriate, with other agents and humans, is of course the main requirement for conceiving communication-intensive agents like self-adaptive protocol-driven ones and it is the assumption under which our proposal works.

As already introduced previously in the chapter, for supporting a self-adaptive protocol-driven approach to agent programming, a formalism for expressing protocols must exist together with a *generate* function for identifying the allowed actions (both sending and receiving) for moving from the current state of the protocol to the next one. What differentiates the behaviour of each agent are the *select* policy to select the message to send among the allowed ones, and the *react* policy to react to an incoming message. Two more policies must be defined to state how to manage *unexpected* messages and which *cleanup* actions to perform before switching from the currently executing protocol to the new one (the policies will be formally defined in Section 6.2.1).

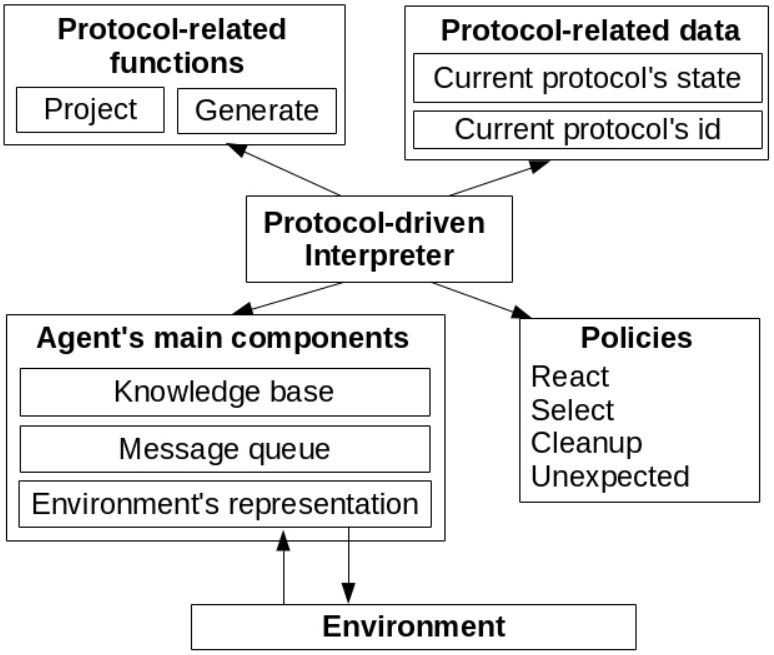


FIGURE 3.7: Architecture of a self-adaptive protocol-driven agent.

The architecture of a self-adaptive protocol-driven agent is depicted in Figure 3.7. The interpreter implements a cycle where it first checks if there is a protocol switch request and if it can be managed in the current state of the protocol. If yes, and if the sender has the power to make such a request - namely, if it is a system's controller -, a protocol switch is performed after some cleanup operations. If no protocol switch is foreseen by the protocol in that moment, or no protocol switch request has been received, the normal communication actions that can be performed are generated and, according to the precedence that the agent gives to receiving or sending (in case both options are available), one of them is performed. The environment representation and knowledge base are updated accordingly and the protocol moves to the next state. In case the received message was not foreseen

by the protocol, it is managed according to the unexpected policy. An alarm which is reset any time an action is performed allows to avoid deadlocks, for example in case the agent gives the precedence to receiving, but no messages are available in the message queue. When the alarm expires the agent cannot wait any longer and selects one of the possible messages to be sent (if any), forcing itself to adopt a sending precedence.

In order to move from a general description of the framework to its implementation, we need to fix some choices, in particular the protocol representation formalism (and, as a direct consequence, the *generate* and *project* functions) and the underlying agent tool. Our implementation instantiates the general one in the following way:

– All the self-adaptive protocol-driven agents use the same protocol formalism and are able to perform generation and projection by themselves.

– There is an external protocol library.

– The formalism for expressing protocols is that of "trace expressions" (it will be presented in Chapter 4).

– The projection algorithm is the one defined in [4]. Projection, already introduced as building block of our framework in Section 3.1.3, can be described as a function $project : Pr \times \mathcal{P}(Agents) \rightarrow Pr$ where Pr is the set of protocols, that are expressed with one of the existing formalism. Given a protocol Pr_1 and a set of agents $Agents$ as input, *project* returns a protocol $Pr_{1Agents}$ which contains only events involving agents in $Agents$: events that do not involve agents in $Agents$ are removed from $Pr_{1Agents}$.

We can go into details of protocol specifications in the case of self-adaptive protocol-driven agents, where it is important noting how the protocol switch is obtained.

Protocol specifications. As just mentioned in the previous section, all protocols are represented using a well defined formalism. In the background (see Section 1.6) we already presented the formalism of constrained global types.
In order to define our protocols we use a different formalism which originated as an extension of constrained global types: trace expressions[10].
Protocols thus are formalized using trace expressions. States of the protocol are

[10]In Chapter 4 we will analyze trace expressions in more detail.

represented by trace expressions as well, blurring the distinction between "protocol" and "protocol state". For requesting a protocol switch we introduce a special `switch` performative with a protocol as content. Like any other event, protocol switches may be allowed in some points of the protocol and not in others, making it possible for the agents to switch to a new protocol only at the right time.
A protocol is characterized by its identifier, its specification and the definition of event types identifying events allowed in a specific state of the protocol.

The description of events admitted inside our framework were introduced in Section 1.6. We remind that we are limiting our investigation to communicative events focusing on a low level customization.

Events. Events have the form `msg(Sender, Receiver, Perf, Content)`[11]. `Sender` and `Receiver` are agent identifiers, `Perf` is a performative in the agent communication language supported by the underlying tool (for example, KQML [72] in Jason and FIPA-ACL [53] in JADE), `Content` is the event content in some suitable representation language. This model supports point to point communication only, although a broadcast operator could be defined on top of that.
We use the term "events" rather than "message" to stress that a protocol always represents a global description of what is expected to go on: the notion of event summarizes that one agent sends a message and another is expected to receive it. From the viewpoint of one single agent Ag, an event corresponds to an incoming message if Ag is the receiver, and corresponds to a message that Ag is expected to send, if it is the sender.
For requesting a protocol switch, `msg(Sender, Receiver, switch, Protocol)` is used: `Sender`, which must have the power to request a protocol switch to `Receiver`, wants that `Receiver` starts behaving according to `Protocol` as soon as possible (we will explain what "as soon as possible" means in Section 6.2.2).
Event types, as already introduced in Section 1.6, add an abstraction level between the actual events taking place in the MAS and the protocol specification.
For example,
`ask_enter_treasure(hobbit1)` is the event type of `msg(hobbit1, bilbo, ask, enter_treasure)`. We state that `msg(hobbit1, bilbo, ask, enter_treasure)` $\in$ `ask_enter_treasure(hobbit1)` and we use the latter in the protocol specification.

[11] We remind that, without loss of generality, in this thesis we only consider communicative events.

If more than one hobbit, say `hobbit1`, `hobbit2` and `hobbit3`, could ask Bilbo to enter the treasure room and the protocol did not need to distinguish among them, we could introduce an event type `ask_enter_treasure` not depending on the hobbit, and state that the three events

- `msg(hobbit1, bilbo, ask, enter_treasure)`,
- `msg(hobbit2, bilbo, ask, enter_treasure)` and
- `msg(hobbit3, bilbo, ask, enter_treasure)`

have event type `ask_enter_treasure`. When the protocol is in a state where `ask_enter_treasure` is allowed, any actual event $\texttt{msg(hobbit}_\texttt{i}\texttt{, bilbo, ask, enter\_treasure)}$, with $\texttt{i} \in \{1, 2, 3\}$, can take place.

Chapter 4

Trace expressions

In this chapter, we present trace expressions as a constrained global types extension (see Chapter 1) and we formally compare the expressive power of trace expressions with the LTL, a formalism widely adopted in runtime verification. We show that any LTL formula can be translated into a trace expression which is equivalent from the point of view of runtime verification. Since trace expressions are able to express and verify sets of traces that are not context-free, we can derive that in the context of runtime verification our formalism is more expressive than LTL.

Trace expressions are a compact and expressive formalism, which can be employed to model complex interaction protocols, and to generate monitors for the Jason and Jade platforms, and can be generalized to support runtime verification of different kinds of properties and systems.

4.1 Runtime verification using trace expressions

As we mentioned in Section 1.4.2, there are several specification formalisms employed by RV; some of them are well-known formalisms that have been originally introduced for other aims, as regular expressions, context free grammars, and linear time temporal (LTL) logic, while others have been expressly devised for RV.

Trace expressions belong to this latter group; they are an evolution of constrained global types [3, 7, 71], which have been initially proposed for RV of agent interactions in multiagent systems. Trace expressions are an expressive formalism based

on a set of operators (including prefixing, concatenation, shuffle, union, and intersection) to denote finite and infinite traces of events. Their semantics is based on a labeled transition system defined by a simple set of rewriting rules which directly drive the behaviour of monitors generated from trace expressions.

In this chapter we introduce in detail the trace expressions formalism and we formally compare it with the LTL, a formalism already introduced in Section 1.5 which is widely used for RV, even though it was initially introduced for model checking.

When used for RV, the expressive power of LTL is reduced, because at runtime only finite traces can be checked, as already anticipated in Section 1.5.3. For instance, the formula Fp (finally p) which states that an event satisfying the predicate p will eventually occur after a finite trace of other occurred events, can only be partially verified at runtime, because no monitor is able to reject an infinite trace of events that do not satisfy p, which, of course, is not a model for Fp.

To provide a formal account for this limitation, a three-valued semantics for LTL, called LTL_3 has been proposed [17].

A third truth value "?" is introduced to specify that after a finite trace of events has been occurred, the outcome of a monitor can be inconclusive. For instance, if we consider the formula Fp, and the event e which does not satisfy p, then no monitor generated from Fp is able to decide whether Fp is satisfied or not after the trace eee.

In trace expressions this limitation of RV is naturally modeled by the standard semantics: if the semantics of a trace expression τ contains all finite traces e, ee, eee, ..., then it must also contain the infinite trace $e \dots e \dots$ because no monitor generated from τ will be able to reject it. This corresponds to the more formal claim stating that the semantics of any trace expression is a complete metric space of traces, when the standard distance between traces is considered.

As a consequence, when the standard semantics is considered, one can conclude that LTL and trace expressions are not comparable: neither is more expressive than the other. However, since the two formalisms are considered in the context of RV, if the more appropriate three-valued semantics is considered, then trace expressions are strictly more expressive than the LTL: every LTL formula can be encoded into a trace expression with an equivalent three-valued semantics, whereas

the opposite property does not hold, since trace expressions are also able to specify context-free and non context-free languages.

4.2 The trace expression formalism

Trace expressions introduce a novelty with respect to constrained global types: besides the union (a.k.a. choice), concatenation, and shuffle (a.k.a. fork) operators, trace expressions support intersection as well. Intersection replaces the constrained shuffle operator [3, 5], an extension of the shuffle operator introduced for making constrained global types more expressive. Constrained shuffle imposes synchronization constraints on the events inside a shuffle, thus making constrained global types and their semantics more complex; furthermore, constrained shuffle is not compositional: it cannot be expressed as an operation between sets of event traces (that is, the mathematical entities denoted by trace expressions). In contrast, the intersection operator has a simple, intuitive, and compositional semantics (as suggested by the name itself) and yet is very expressive; for instance, as shown in Section 4.3, it can be used for specifying non context-free sets of event traces.

4.2.1 Events

In the following we denote by $\mathcal{E}$ a fixed universe of events. An event trace over $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$. In the rest of the thesis the meta-variables e, w, σ and u will range over the sets $\mathcal{E}$, $\mathcal{E}^\omega$, $\mathcal{E}^*$, and $\mathcal{E}^\omega \cup \mathcal{E}^*$, respectively; juxtaposition $e\,u$ denotes the trace where e is the first event, and u is the rest of the trace. A trace expression over $\mathcal{E}$ denotes a set of event traces over $\mathcal{E}$.

As a possible example, we might have $\mathcal{E} = \{o.m \mid o \text{ object identity}, m \text{ method name}\}$ where the event $o.m$ corresponds to an invocation of method named[1] m on the target object o. This is a typical example of set of events arising when monitoring object-oriented systems (we will show an example later on).

It is important to remember also the special event *switch* that we have already seen in Section 3.2.5. Respect to all other events, it must be handled differently;

[1]Here, for simplicity, an event does not include the signature of the method as it should be the case for those languages supporting static overloading.

indeed, this event allows an agent to ask to another a *protocol switch* at runtime[2] (in this way all the agents can have different behaviour at different times of their life).

4.2.2 Event types

The *event types* have already been used previously in the thesis; they were introduced in Section 1.6, in the context of the constrained global types and they were used in Section 3.2.5 during the protocol specification. As already seen, the *event types* allow our formalisms to be more general, accordingly, also trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), rather than of single events; an event type denotes a subset of $\mathcal{E}$, and corresponds to a predicate of arity $k \geq 1$, where the first implicit argument corresponds to the event e under consideration; referring to the example where events are method invocations, we may introduce the type $safe(o)$ of all safe method invocations for a given object o, defined by the predicate *safe* of arity 2 s.t. $safe(e, o)$ holds iff $e = o.isEmpty$.

The first argument of the predicate is left implicit in the event type, and we write $e \in safe(o)$ to mean that $safe(e, o)$ holds. Similarly, the set of events specified by an event type ϑ is denoted by $[\![\vartheta]\!]$; for instance, $[\![safe(o)]\!] = \{e \mid e \in safe(o)\}$.

For generality, we leave unspecified the formalism used for defining event types; however, in practice we do not expect that much expressive power is required. For instance, for all examples presented in this work a formalism less powerful than regular expressions is sufficient.

4.2.3 Trace expressions

Similarly to a costrained global type, whose syntax was described in Section 1.6, a trace expression τ represents a set of possibly infinite event traces, and is defined on top of the following operators[3], in which, only the *intersection* operator is the new operator introduced with the trace expressions, all the others are inherited from the constrained global types:

[2]We postpone all technical details on how it is handled inside our interpreter in Chapter 6.

[3]Binary operators associate from left, and are listed in decreasing order of precedence, that is, the first operator has the highest precedence.

- λ (empty trace), denoting the singleton set $\{\varepsilon\}$ containing the empty event trace ε.
- $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace of τ.
- $\tau_1{\cdot}\tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2.
- $\tau_1{\wedge}\tau_2$ (*intersection*), denoting the intersection of the traces of τ_1 and τ_2.
- $\tau_1{\vee}\tau_2$ (*union*), denoting the union of the traces of τ_1 and τ_2.
- $\tau_1|\tau_2$ (*shuffle*), denoting the set obtained by shuffling the traces in τ_1 with the traces in τ_2.

To support recursion without introducing an explicit construct, trace expressions are regular (a.k.a. rational or cyclic) terms, as well as the constrained global types: they correspond to trees where nodes are either the leaf λ, or the node (corresponding to the prefix operator) ϑ with one child, or the nodes $\cdot$, $\wedge$, $\vee$, and $|$ all having two children. According to the standard definition of rational trees, their depth is allowed to be infinite, but the number of their subtrees must be finite. As originally proposed by Courcelle [41], such regular trees can be modeled as partial functions from $\{0,1\}^*$ to the set of nodes (in our case $\{\lambda, \cdot, \wedge, \vee, |\} \cup \mathcal{ET}$) satisfying certain conditions.

A regular term can be represented by a finite set of syntactic equations, as happens, for instance, in most modern Prolog implementations where unification supports cyclic terms.

As an example of non recursive trace expression, let $\mathcal{E}$ be the set $\{e_1, \ldots, e_7\}$, and ϑ_i, $i = 1, \ldots, 7$, be the event types such that $e \in \vartheta_i$ iff $e = e_i$ (that is, $[\![\vartheta_i]\!] = \{e_i\}$); then the trace expression

$$TE_1 = ((\vartheta_1{:}\lambda|\vartheta_2{:}\lambda){\vee}(\vartheta_3{:}\lambda|\vartheta_4{:}\lambda)){\cdot}(\vartheta_5{:}\vartheta_6{:}\lambda|\vartheta_7{:}\lambda)$$

denotes the following set of event traces:

$$\left\{ \begin{array}{l} e_1e_2e_5e_6e_7, e_1e_2e_5e_7e_6, e_1e_2e_7e_5e_6, e_2e_1e_5e_6e_7, e_2e_1e_5e_7e_6, e_2e_1e_7e_5e_6, \\ e_3e_4e_5e_6e_7, e_3e_4e_5e_7e_6, e_3e_4e_7e_5e_6, e_4e_3e_5e_6e_7, e_4e_3e_5e_7e_6, e_4e_3e_7e_5e_6 \end{array} \right\}$$

As an example of recursive trace expression, if ϑ_i denotes the same event type defined above for $i = 1, \ldots, 7$, and $[\![\vartheta]\!] = \{e_4, e_5, e_6, e_7\}$, $[\![\vartheta']\!] = \{e_1, e_2, e_6, e_7\}$, and $[\![\vartheta'']\!] = \{e_1, e_2, e_3, e_4\}$, then the trace expression

$$TE_2 = (E|\vartheta_1{:}\vartheta_2{:}\vartheta_3{:}\lambda) \wedge (E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\lambda) \wedge (E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\lambda)$$
$$E = \lambda \vee \vartheta{:}E \qquad E' = \lambda \vee \vartheta'{:}E' \qquad E'' = \lambda \vee \vartheta''{:}E''$$

denotes the set $\{e_1e_2e_3e_4e_5e_6e_7\}$.

Finally, the recursive trace expressions $T_1 = (\lambda \vee \vartheta_1{:}T_1) \cdot T_2$, $T_2 = (\lambda \vee \vartheta_2{:}T_2)$ represent the infinite but regular terms $(\lambda \vee \vartheta_1{:}(\lambda \vee \vartheta_1{:}\ldots)) \cdot (\lambda \vee \vartheta_2{:}(\lambda \vee \vartheta_2{:}\ldots))$ and $(\lambda \vee (\vartheta_2{:}(\lambda \vee (\vartheta_2{:}\ldots))))$, respectively.

In the rest of the work we will limit our investigation to *contractive* (a.k.a. *guarded*) trace expressions (as in Section 1.6 in the case of constrained global types). Even if already presented informally in Section 1.6, we can define here formally what is a *contractive* trace expression (or in the same way a *contractive* constrained global types).

Definition 4.1. A trace expression τ is *contractive* if all its infinite paths contain the prefix operator.

In contractive trace expressions all recursive subexpressions must be guarded by the prefix operator; for instance, the trace expression defined by $T_1 = (\lambda \vee (\vartheta{:}T_1))$ is contractive: its infinite path contains infinite occurrences of $\vee$, but also of the : operator; conversely, the trace expression $T_2 = \vartheta{:}T_2 \vee T_2$ is not contractive.

Trivially, every trace expression corresponding to a finite tree (that is, a non cyclic term) is contractive.

For all contractive trace expressions, any path from their root must always reach either a λ or a : node in a finite number of steps. Since in this work all definitions over trace expressions treat $\vartheta{:}\tau$ as a base case (that is, the definition is not propagated to the subexpression τ), restricting trace expressions to contractive ones has the advantage that most of the definitions and proofs requires induction, rather than coinduction, despite trace expressions can be cyclic. As a consequence, the implementation of trace expressions becomes considerably simpler. For this reason, in the rest of the thesis we will only consider contractive trace expressions.

$$\text{(prefix)}\frac{}{\vartheta{:}\tau \xrightarrow{e} \tau}\ e{\in}\vartheta \qquad \text{(or-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 \vee \tau_2 \xrightarrow{e} \tau_1'} \qquad \text{(or-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \vee \tau_2 \xrightarrow{e} \tau_2'}$$

$$\text{(and)}\frac{\tau_1 \xrightarrow{e} \tau_1' \quad \tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \wedge \tau_2 \xrightarrow{e} \tau_1' \wedge \tau_2'} \quad \text{(shuffle-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 | \tau_2 \xrightarrow{e} \tau_1' | \tau_2} \qquad \text{(shuffle-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 | \tau_2 \xrightarrow{e} \tau_1 | \tau_2'}$$

$$\text{(cat-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 \cdot \tau_2 \xrightarrow{e} \tau_1' \cdot \tau_2} \qquad \text{(cat-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \cdot \tau_2 \xrightarrow{e} \tau_2'}\ \varepsilon(\tau_1)$$

FIGURE 4.1: Operational semantics of trace expressions

$$\text{(ε-empty)}\frac{}{\varepsilon(\lambda)} \qquad \text{(ε-or-l)}\frac{\varepsilon(\tau_1)}{\varepsilon(\tau_1 \vee \tau_2)} \qquad \text{(ε-or-r)}\frac{\varepsilon(\tau_2)}{\varepsilon(\tau_1 \vee \tau_2)} \qquad \text{(ε-shuffle)}\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 | \tau_2)}$$

$$\text{(ε-cat)}\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 \cdot \tau_2)} \qquad \text{(ε-and)}\frac{\varepsilon(\tau_1) \quad \varepsilon(\tau_2)}{\varepsilon(\tau_1 \wedge \tau_2)}$$

FIGURE 4.2: Empty trace containment

As in constrained global types, which use the *next* transition function to move from a protocol state to another one, also in trace expressions we have a transition function (corresponding to the trace expressions semantics) which can be specified by the transition relation $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$, where $\mathcal{T}$ and $\mathcal{E}$ denote the set of trace expressions and of events, respectively. As it is customary, we write $\tau_1 \xrightarrow{e} \tau_2$ to mean $(\tau_1, e, \tau_2) \in \delta$.

$$next(\tau_0, e) = \{\tau_1, \tau_2, ..., \tau_n\} \iff \forall_{1 \leq i \leq n}.(\tau_0, e, \tau_i) \in \delta$$

The set generated from $next(\tau_0, e)$ can be infinite. If the trace expression τ_1 specifies the current valid state of the system, then an event e is considered valid iff there exists a transition $\tau_1 \xrightarrow{e} \tau_2$; in such a case, τ_2 will specify the next valid state of the system after event e. Otherwise, the event e is not considered to be valid in the current state represented by τ_1. Figure 4.1 defines the inductive rules for the transition function.

While the transition relation δ with its corresponding rules in Figure 4.1 defines the non empty traces of a trace expression, the predicate $\varepsilon(\_)$, inductively defined by the rules in Figure 4.2, defines the trace expressions that contain the empty trace ε. If $\varepsilon(\tau)$ holds, then the empty trace is a valid trace for τ.

Rule (prefix) states that valid traces of $\vartheta{:}\tau$ can only start with an event e of type ϑ (side condition $e \in \vartheta$), and continue with traces in τ.

Rules (or-l) and (or-r) state that the only valid traces of $\tau_1 \vee \tau_2$ have shape $e\ u$, where either $e\ u$ is valid for τ_1 (rule (or-l)), or $e\ u$ is valid for τ_2 (rule (or-r)).

Rule (and) states that the only valid traces of $\tau_1 \wedge \tau_2$ have shape $e\ u$, where $e\ u$ is valid for both τ_1 and τ_2.

Rules (shuffle-l) and (shuffle-r) state that the only valid traces of $\tau_1 | \tau_2$ have shape $e\ u$, where either $e\ u_1'$ and u_2 are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the shuffle of u_1' with u_2 (rule (shuffle-l)), or u_1 and $e\ u_2'$ are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the shuffle of u_1 with u_2' (rule (shuffle-r)).

Rules (cat-l) and (cat-r) state that the only valid traces of $\tau_1 \cdot \tau_2$ have shape $e\ u$, where either $e\ u_1'$ and u_2 are valid traces for τ_1 and τ_2, respectively, and u can be obtained as the concatenation of u_1' to u_2 (rule (cat-l)), or λ is a valid trace for τ_1 (side condition $\varepsilon(\tau_1)$) and $e\ u$ is a valid trace for τ_2 (rule (cat-r)).

For what concerns Figure 4.2, rules (ε-shuffle), (ε-cat) and (ε-and) require the empty trace to be contained in both subexpressions τ_1 and τ_2, whereas for the union operator it suffices that the empty trace is contained in either τ_1 (rule (ε-or-l)) or τ_2 (rule (ε-or-r)). The prefix operator can never build sets of traces containing the empty trace, whereas λ contains just the empty trace (rule (ε-empty)).

The set of traces $[\![\tau]\!]$ denoted by a trace expression τ is defined in terms of the transition relation δ, and the predicate $\varepsilon(\_)$. Since $[\![\tau]\!]$ may contain infinite traces, the definition of $[\![\tau]\!]$ is coinductive.

Definition 4.2. For all possibly infinite event traces u and trace expressions τ, $u \in [\![\tau]\!]$ is coinductively defined as follows:

- either $u = \varepsilon$ and $\varepsilon(\tau)$ holds,
- or $u = e\,u'$, and there exists τ' s.t. $\tau \xrightarrow{e} \tau'$ and $u' \in [\![\tau']\!]$ hold.

In the following we will need to consider the reflexive and transitive closure of the transition relation: if σ is a finite (possibly empty) event trace, then the relation $\tau \xrightarrow{\sigma} \tau'$ is inductively defined as follows: $\tau \xrightarrow{\sigma} \tau'$ holds iff

- $\sigma = \varepsilon$, and $\tau' = \tau$;
- or $\sigma = e\,\sigma'$, and there exists τ'' s.t. $\tau \xrightarrow{e} \tau''$, and $\tau'' \xrightarrow{\sigma'} \tau'$.

Let us consider again the previous examples of trace expressions:

$$
\begin{array}{l}
TE_1 = ((\vartheta_1{:}\lambda|\vartheta_2{:}\lambda)\vee(\vartheta_3{:}\lambda|\vartheta_4{:}\lambda))\cdot(\vartheta_5{:}\vartheta_6{:}\lambda|\vartheta_7{:}\lambda) \\
TE_2 = (E|\vartheta_1{:}\vartheta_2{:}\vartheta_3{:}\lambda)\wedge(E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\lambda)\wedge(E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\lambda) \\
E = \lambda\vee\vartheta{:}E \qquad E' = \lambda\vee\vartheta'{:}E' \qquad E'' = \lambda\vee\vartheta''{:}E'' \\
\forall\, i \in \{1..7\}\ [\![\vartheta_i]\!] = \{e_i\} \qquad [\![\vartheta]\!] = \{e_4, e_5, e_6, e_7\} \\
[\![\vartheta']\!] = \{e_1, e_2, e_6, e_7\} \quad [\![\vartheta'']\!] = \{e_1, e_2, e_3, e_4\}
\end{array}
$$

We show that there exist τ_1, τ_2 s.t. $TE_1 \xrightarrow{\sigma_1} \tau_1$, with $\sigma_1 = e_1e_2e_5e_6e_7$, $\varepsilon(\tau_1)$, $TE_2 \xrightarrow{\sigma_2} \tau_2$, with $\sigma_2 = e_1e_2e_3e_4e_5e_6e_7$, and $\varepsilon(\tau_2)$.

For $TE_1 \xrightarrow{\sigma_1} \tau_1$ we have $\vartheta_1{:}\lambda|\vartheta_2{:}\lambda \xrightarrow{e_1} \lambda|\vartheta_2{:}\lambda \xrightarrow{e_2} \lambda|\lambda$, hence $(\vartheta_1{:}\lambda|\vartheta_2{:}\lambda)\vee(\vartheta_3{:}\lambda|\vartheta_4{:}\lambda) \xrightarrow{e_1e_2} \lambda|\lambda$, and $TE_1 \xrightarrow{e_1e_2} (\lambda|\lambda)\cdot(\vartheta_5{:}\vartheta_6{:}\lambda|\vartheta_7{:}\lambda)$. Furthermore, $\vartheta_5{:}\vartheta_6{:}\lambda|\vartheta_7{:}\lambda \xrightarrow{e_5} \vartheta_6{:}\lambda|\vartheta_7{:}\lambda \xrightarrow{e_6} \lambda|\vartheta_7{:}\lambda \xrightarrow{e_7} \lambda|\lambda$, hence $\vartheta_5{:}\vartheta_6{:}\lambda|\vartheta_7{:}\lambda \xrightarrow{e_5e_6e_7} \lambda|\lambda$, and, because $\varepsilon(\lambda|\lambda)$, we can conclude $(\lambda|\lambda)\cdot(\vartheta_5{:}\vartheta_6{:}\lambda|\vartheta_7{:}\lambda) \xrightarrow{e_5e_6e_7} \lambda|\lambda$, hence, $TE_1 \xrightarrow{e_1e_2e_5e_6e_7} \lambda|\lambda$.

For $TE_2 \xrightarrow{\sigma_2} \tau_2$ we have $E|\vartheta_1{:}\vartheta_2{:}\vartheta_3{:}\lambda \xrightarrow{e_1e_2e_3} E|\lambda \xrightarrow{e_4e_5e_6e_7} E|\lambda$, and $E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\lambda \xrightarrow{e_1e_2} E'|\vartheta_3{:}\vartheta_4{:}\vartheta_5{:}\lambda \xrightarrow{e_3e_4e_5} E'|\lambda \xrightarrow{e_6e_7} E'|\lambda$, and, finally, $E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\lambda \xrightarrow{e_1e_2e_3e_4} E''|\vartheta_5{:}\vartheta_6{:}\vartheta_7{:}\lambda \xrightarrow{e_5e_6e_7} E''|\lambda$. Therefore $TE_2 \xrightarrow{e_1e_2e_3e_4e_5e_6e_7} (E|\lambda)\wedge(E'|\lambda)\wedge(E''|\lambda)$ and $\varepsilon(E|\lambda)$, $\varepsilon(E'|\lambda)$, and $\varepsilon(E''|\lambda)$, hence $\varepsilon((E|\lambda)\wedge(E'|\lambda)\wedge(E''|\lambda))$.

Since the semantics of trace expressions is coinductive, they can specify non terminating behaviour; for instance, the trace expression defined by $T = \vartheta_1{:}T$ denotes the set with just the infinite trace $e_1\, e_1 \ldots e_1 \ldots$ containing infinite occurrences of e_1; had we considered an inductive semantics, T would have denoted the empty set. For the very same reason, the trace expression defined by $T' = \lambda\vee\vartheta_1{:}T'$ denotes the set containing all finite traces of the event e_1, but also the infinite trace $e_1\, e_1 \ldots e_1 \ldots$. From the point of view of RV, the only difference between the two types is that for T' the monitored system is allowed to halt at any time, whereas for T the system can never stop.

Since at runtime it is not possible to check that a given monitored system will always eventually stop, trace expressions cannot denote sets of traces which are not complete metric spaces, with the standard distance between traces: $d(u_1, u_2) =$

2^{-n}, where n denotes the smallest index (starting from 0) at which the two traces are different; by convention, if the two traces are equal, than $n = \infty$, and $2^{-n} = 0$. For instance, if the semantics of a trace expression τ contains traces of arbitrarily large length of the event e_1, then it also contains the infinite trace $e_1\, e_1 \ldots e_1 \ldots$; indeed, the monitor associated with τ will not be able to reject it.

Such a limitation is independent of the used formalism, but it is intimately related to RV; as pointed out in Section 4.4, similar issues arise when the LTL is used for RV: its semantics has to be revisited to take into account the fact that at runtime only finite traces can be monitored and checked.

4.2.3.1 The *finite_composition* operator

In order to create more complex protocols, we must further modify the syntax of trace expressions. In particular, we need a new operator which allows us to replicate a trace expression a chosen number of times.

This new construct is called *finite_composition*; its syntax is:

$$finite\_composition(operator, \tau, variables)$$

where

- *operator* may be any binary operator ($|$, $\vee$, $\wedge$, $\cdot$);
- τ is the trace expression which we want to replicate;
- *variables* is the set of variables on which we want to iterate, it has this form: $\{(var(1), \{\}), (var(2), \{\}), ..., (var(n), \{\})\}$.
  Each tuple has two arguments, the first represents the variable on which we want to iterate, the second defines the set of modifiers. A modifier can be $add(var(i))$ or $remove(var(i))$, where $var(i)$ must be a variable which has been already defined in a *finite_composition* operator; for instance, if we have $(var(1), \{add(var(2)), remove(var(3))\})$, we are saying that $var(1)$ must iterate on the set which is fixed during the instantiation phase, adding also the current value of $var(2)$ and removing the current value of $var(3)$.

The *finite_composition* operator is like all the other trace expression operators. Indeed, it is an expression which represent a trace expression and it will be applied and instantiated only during the instantiation phase (see Section 6.3.1.6).

Considering the event type used in the example in Section 4.2.2: $[\![safe(o)]\!] = \{e \mid e \in safe(o)\}$, we want to write a trace expression which consists in a *shuffle* operator where each branch contains this *prefix* operator but, with a different argument inside the event type for each branch. For instance, like: $safe(o_1) : \lambda \mid safe(o_2) : \lambda \mid ... \mid safe(o_n) : \lambda$.

The *finite_composition* operator is different from all the other operators, indeed, it is a special operator which, during the instantiation phase (that is a preprocessing phase), transforms the trace expression where it appears in a trace expression like those described before where the *finite_composition* operator is removed making explicit the composition. A trace expression containing at least one *finite_composition* operator is called **template** trace expression, as we will see in more detail in Section 4.5, and it requires to be instantiated before it can be used; indeed, for a trace expression to be used at runtime, both for runtime verification and for generating self-adaptive protocol-driven agents, it must not have variable inside it and it must be fully instantiated (in this way we can consider the *finite_composition* operator as a sort of "meta-operator").

Returning to the example, without the *finite_composition* operator we would have written the *prefix* operator n times; using instead this new construct we can write:

$$finite\_composition(|, safe(var(1)) : \lambda, \{(var(1), \{\})\})$$

which, if applied passing $\{o_1, o_2, ..., o_n\}$ as the set on which var_1 iterates, returns the trace expression: $safe(o_1) : \lambda \mid safe(o_2) : \lambda \mid ... \mid safe(o_n) : \lambda$ (exactly the trace expression we want).

4.2.4 Deterministic trace expressions

As already anticipated informally in Section 1.6, the constrained global types can be *deterministic* and *nondeterministic*. This also applies to trace expressions, indeed, there are trace expressions τ for which the problem of word recognition is less efficient because of non determinism.

In the previous section, we have presented the syntax of trace expressions as a constrained global types syntax evolution highlighting the main differences among the operators; in particular, trace expressions have the same constrained global types operators with in addition the *intersection* operator.

Non determinism originates from the union, shuffle, and concatenation operators, because for each of them two possibly overlapping transition rules are defined; consequently, the new *intersection* operator does not influence the trace expressions *determinism*.

Let us consider the trace expression $\tau = (\vartheta_1{:}\vartheta_2{:}\lambda) \vee (\vartheta_1{:}\vartheta_3{:}\lambda)$, where $[\![\vartheta_i]\!] = \{e_i\}$ for $i \in \{1..3\}$. Both transitions $\tau \xrightarrow{e_1} \vartheta_2{:}\lambda$ and $\tau \xrightarrow{e_1} \vartheta_3{:}\lambda$ are valid, but $[\![\vartheta_2{:}\lambda]\!] \neq [\![\vartheta_3{:}\lambda]\!]$; therefore, to correctly accept the trace e_1e_3, both rules have to be applied simultaneously, and the set of trace expressions $\{\vartheta_2{:}\lambda, \vartheta_3{:}\lambda\}$ has to be considered, as it is done for non deterministic automaton.

Similarly, for the trace expression $\tau' = (\vartheta_1{:}\vartheta_2{:}\lambda)|(\vartheta_1{:}\vartheta_3{:}\lambda)$, both transitions $\tau' \xrightarrow{e_1} (\vartheta_2{:}\lambda)|(\vartheta_1{:}\vartheta_3{:}\lambda)$ and $\tau' \xrightarrow{e_1} (\vartheta_1{:}\vartheta_2{:}\lambda)|(\vartheta_3{:}\lambda)$ are valid, but $[\![(\vartheta_2{:}\lambda)|(\vartheta_1{:}\vartheta_3{:}\lambda)]\!] \neq [\![(\vartheta_1{:}\vartheta_2{:}\lambda)|(\vartheta_3{:}\lambda)]\!]$.

Finally, for the trace expression $\tau'' = (\lambda \vee \vartheta_1{:}\vartheta_2{:}\lambda){\cdot}(\vartheta_1{:}\lambda)$ both transitions $\tau'' \xrightarrow{e_1} (\vartheta_2{:}\lambda){\cdot}(\vartheta_1{:}\lambda)$ and $\tau'' \xrightarrow{e_1} \lambda$ are valid, but $[\![(\vartheta_2{:}\lambda){\cdot}(\vartheta_1{:}\lambda)]\!] \neq [\![\lambda]\!]$.

In the rest of this work we will focus on deterministic trace expressions: indeed, the problem of word recognition is simpler and more efficient in the deterministic case.

Deterministic trace expressions are defined as follows.

Definition 4.3. Let τ be a trace expression; τ is *deterministic* if for all finite event traces σ, if $\tau \xrightarrow{\sigma} \tau'$ and $\tau \xrightarrow{\sigma} \tau''$ are valid, then $[\![\tau']\!] = [\![\tau'']\!]$.

The trace expressions τ, τ', and τ'', as defined above, are not deterministic, while the respectively equivalent trace expressions $\vartheta_1{:}(\vartheta_2{:}\lambda \vee \vartheta_3{:}\lambda)$, $\vartheta_1{:}(((\vartheta_2{:}\lambda)|(\vartheta_1{:}\vartheta_3{:}\lambda)) \vee ((\vartheta_1{:}\vartheta_2{:}\lambda)|(\vartheta_3{:}\lambda)))$, and $\vartheta_1{:}(\lambda \vee \vartheta_2{:}\vartheta_1{:}\lambda)$ are deterministic.

4.3 Examples of specifications with trace expressions

In this section we provide some examples to show the expressive power of trace expressions. Unless specified otherwise, for simplicity in the rest of the thesis we will consider singleton event types, that is, event types representing a single event; with abuse of notation, we will abbreviate events with their corresponding singleton event types.

4.3.1 Derived operators

We first introduce some useful operators that will be used in the rest of the thesis.

Constants. The constants 1 and 0 denote the set of all possible traces over $\mathcal{E}$ and the empty set, respectively. Constant 1 is equivalent to the expression $T = \lambda \vee any{:}T$, where *any* is the event type s.t. $[\![any]\!] = \mathcal{E}$; constant 0 is equivalent to the expression $none{:}\lambda$, where *none* is the event type s.t. $[\![none]\!] = \emptyset$.

Filter operator. The filter operator is useful for making trace expressions more compact and readable. The expression $\vartheta \gg \tau$ denotes the set of all traces contained in τ, when deprived of all events that do not match ϑ. Assuming that event types are closed by complementation, the expression above is a convenient syntactic shortcut for $T|\tau$, where $T = \lambda \vee \overline{\vartheta}{:}T$, and $\overline{\vartheta}$ is the complement event type of ϑ, that is, $[\![\overline{\vartheta}]\!] = \mathcal{E} \setminus [\![\vartheta]\!]$.

The corresponding rules for the transition relation and the auxiliary function $\varepsilon(\_)$ can be easily derived:

$$\text{(cond-t)}\frac{\tau \xrightarrow{e} \tau'}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau'}\ e \in \vartheta \qquad \text{(cond-f)}\frac{}{\vartheta \gg \tau \xrightarrow{e} \vartheta \gg \tau}\ e \notin \vartheta \qquad (\varepsilon\text{-cond})\frac{\varepsilon(\tau)}{\varepsilon(\vartheta \gg \tau)}$$

4.3.1.1 Stack objects

We expand the example where events correspond to method invocations on objects; besides the already introduced event type $safe(o)$ s.t. $e \in safe(o)$ iff $e = o.isEmpty$, we define the following other event types:

$$[\![pop(o)]\!] = \{o.pop\},\ [\![top(o)]\!] = \{o.top\},\ [\![push(o)]\!] = \{o.push\};$$

$$[\![stack(o)]\!] = \{o.pop, o.top, o.push, o.isEmpty\};$$

$$[\![unsafe(o)]\!] = \{o.pop, o.top, o.push\}.$$

Our purpose is to specify through a trace expression *Stack* all safe traces of method invocations on a stack object o which we assume to be initially empty. Safety requires that methods *top* and *pop* can never be invoked on o when o represents the empty stack.

More in details, a trace of method invocations on a given object having identity o is correct iff any finite prefix does not contain more $pop(o)$ event types than $push(o)$, and the event type $top(o)$ can appear only if the number of $pop(o)$ event types is strictly less than the number of $push(o)$ event types occurring before $top(o)$.

The trace expression *Stack* is defined as follows:

$$\begin{aligned} Stack &= Any \wedge unsafe(o) \gg Unsafe \\ Unsafe &= \lambda \vee (push(o) : (Unsafe | (Tops \cdot (pop(o) : \lambda \vee \lambda)))) \\ Any &= \lambda \vee stack(o) : Any \\ Tops &= \lambda \vee top(o) : Tops \end{aligned}$$

A correct stack trace is specified by *Stack* which is the intersection of
Any and $unsafe(o) \gg Unsafe$; *Any* specifies any possible trace of method invocations on stack objects, whereas if an event has type $unsafe(o)$, then it has to verify the trace expression *Unsafe*, which requires that a *push* event must precede a possible empty trace of *top* events, which, in turn, must precede an optional event *pop*; the expression is recursively shuffled with itself, since any *push* event can be safely shuffled with a *top* or a *pop* event.

The specification is deterministic.
To make an example, we can consider $Stack \xrightarrow{\sigma} \tau$ with $\sigma = push(o)\, push(o)$

$\tau = Any \wedge unsafe(o) \gg (Unsafe | Tops \cdot ((pop(o):\lambda) \vee \lambda) | Tops \cdot ((pop(o):\lambda) \vee \lambda))$. We may observe that $\tau \xrightarrow{e} \tau_1$ and $\tau \xrightarrow{e} \tau_2$, with[4]

$$\begin{aligned} e &= pop(o) \\ \tau_1 &= Any \wedge unsafe(o) \gg (Unsafe | \lambda | Tops \cdot ((pop(o):\lambda) \vee \lambda)) \\ \tau_2 &= Any \wedge unsafe(o) \gg (Unsafe | Tops \cdot ((pop(o):\lambda) \vee \lambda) | \lambda) \end{aligned}$$

but $[\![\tau_1]\!] = [\![\tau_2]\!]$.

4.3.2 Alternating Bit Protocol

A more complex example concerning interactions is the alternating bit protocol (ABP), as defined by Deniélou and Yoshida [48], where two parties, Alice and Bob, are involved, and four different types of events can occur: Alice sends a first kind of message to Bob (event type msg_1), Alice sends a second kind of message to Bob (event type msg_2), Bob replies to Alice with an acknowledge to the first kind of message (event type ack_1), Bob replies to Alice with an acknowledge to the second kind of message (event type ack_2). The protocol has to satisfy the following constraints for all event occurrences:

- The n-th occurrence of the event of type msg_1 must precede the n-th occurrence of the event of type msg_2, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_1.
- The n-th occurrence of the event of type msg_1 must precede the n-th occurrence of the event of type ack_1, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_1.
- The n-th occurrence of the event of type msg_2 must precede the n-th occurrence of the event of type ack_2, which, in turn, must precede the $(n+1)$-th occurrence of the event of type msg_2.

[4]For efficiency reasons, our implementation exploits simplification opportunities after each transition step, therefore in practice for this example the two transitions would lead to the same expression.

The protocol can be specified by the following trace expression (starting from variable $AltBit_1$):

$$\begin{array}{rclcrcl} AltBit_1 & = & msg_1{:}M_2 & & AltBit_2 & = & msg_2 : M_1 \\ M_1 & = & msg_1{:}A_2 \vee ack_2{:}AltBit_1 & & M_2 & = & msg_2{:}A_1 \vee ack_1{:}AltBit_2 \\ A_1 & = & ack_1{:}M_1 \vee ack_2{:}ack_1{:}AltBit_1 & & A_2 & = & ack_2{:}M_2 \vee ack_1{:}ack_2{:}AltBit_2 \end{array}$$

In this case the prefix and union operators are sufficient for specifying the correct behaviour of the system, however, the corresponding trace expression is not very readable. More importantly, if only the prefix and union operators are employed, the size of the expressions grows exponentially with the number of different involved event types.

This problem can be avoided by the use of the intersection and filter operators.

Let $msg\_ack(i)$, $i \in \{1..2\}$, and msg denote the event types s.t. $[\![msg\_ack(i)]\!] = [\![msg_i]\!] \cup [\![ack_i]\!]$, $i \in \{1..2\}$, and $[\![msg]\!] = [\![msg_1]\!] \cup [\![msg_2]\!]$. Then the ABP can be specified by the following deterministic trace expression:

$$\begin{array}{rcl} AltBit & = & (msg \gg MM) \wedge (msg\_ack(1) \gg MA_1) \wedge (msg\_ack(2) \gg MA_2) \\ MM & = & msg_1{:}msg_2{:}MM \\ MA_1 & = & msg_1{:}ack_1{:}MA_1 \\ MA_2 & = & msg_2{:}ack_2{:}MA_2 \end{array}$$

The three trace expressions defined by MM, MA_1, and MA_2 correspond to the three constraints informally stated above. The main trace expression $AltBit$ can be easily read as follows: if an event has type msg_1 or msg_2, then it must verify MM, and if an event has type msg_1 or ack_1, then it must verify MA_1, and if an event has type msg_2 or ack_2, then it must verify MA_2.

The trace expression can be easily generalized to k different kinds of messages (with $k \geq 2$), with the size of the expression growing linearly with the number of different involved event types. For instance, for $k = 3$ we have the following trace

expression:

$$\begin{array}{rcl} AltBit & = & (msg \gg MM) \wedge (msg\_ack(1) \gg MA_1) \wedge \\ & & (msg\_ack(2) \gg MA_2) \wedge (msg\_ack(3) \gg MA_3) \end{array}$$

$$\begin{array}{rclrcl} MM & = & msg_1 : msg_2 : msg_3 : MM & MA_1 & = & msg_1 : ack_1 : MA_1 \\ MA_2 & = & msg_2 : ack_2 : MA_2 & MA_3 & = & msg_3 : ack_3 : MA_2 \end{array}$$

4.3.3 Non context free languages

Trace expressions allow the specification of non context free languages; let us consider for instance the typical example of non context free language $\{a^n b^n c^n \mid n \geq 0\}$. This language can be specified by the following trace expression (defined by T)

$$T = (a\_or\_b \gg AB) \wedge (b\_or\_c \gg BC)$$

$$AB = \lambda \vee (a{:}(AB \cdot (b{:}\lambda)))$$

$$BC = \lambda \vee (b{:}(BC \cdot (c{:}\lambda)))$$

where $[\![a]\!] = \{a\}$, $[\![b]\!] = \{b\}$, $[\![c]\!] = \{c\}$, $[\![a\_or\_b]\!] = \{a, b\}$, and $[\![b\_or\_c]\!] = \{b, c\}$.

Assuming the universe of events $\mathcal{E} = \{a, b, c\}$, the expression $a\_or\_b \gg AB$ denotes all traces of events over $\mathcal{E}$ that, when restricted to finite length[5] and to events a or b, correspond to the sequence $a^n b^n$ for some $n \in \mathbb{N}$; similarly, the expression $b\_or\_c \gg BC$ denotes all traces of events over $\mathcal{E}$ that, when restricted to finite length and to events b or c, correspond to the sequence $b^n c^n$ for some $n \in \mathbb{N}$. Therefore the finite traces of expression T, which is the intersection of $a\_or\_b \gg AB$ and $b\_or\_c \gg BC$, are the non-context free language $\{a^n b^n c^n \mid n \geq 0\}$.

Although T is deterministic, it has the drawback that non correct traces can be detected with a certain latency. For instance the transition $T \xrightarrow{aabc} T'$ holds, with $T' = (a\_or\_b \gg (b{:}\lambda)) \wedge (b\_or\_c \gg \lambda)$, and clearly $aabc$ is not a valid prefix for the language; however, $[\![T']\!] = \emptyset$, and T' is not able to accept any further event, that is, recognition fails, independently from the next event.

To avoid this problem, the following equivalent (assuming that $\mathcal{E} = \{a, b, c\}$) deterministic trace expression can be employed:

[5]Recall that for a comparison with context-free languages we need to disregard infinite traces; for instance, $a\_or\_b \gg AB$ and $b\_or\_c \gg BC$ contain also the infinite traces a^ω and b^ω, respectively.

$$\begin{array}{rclcrcl} T_2 & = & (AB{\cdot}C){\wedge}(b\_or\_c{\gg}BC) & & AB & = & \lambda{\vee}(a{:}(AB{\cdot}(b{:}\lambda))) \\ BC & = & \lambda{\vee}(b{:}(BC{\cdot}(c{:}\lambda))) & & C & = & \lambda{\vee}c{:}C \end{array}$$

In this case, $AB{\cdot}C$ forces events of type c to occur only after all required events of type b have been already occurred. In this case there is no T_2'' s.t. $T_2 \stackrel{aabc}{\rightarrow} T_2''$ holds; indeed, $T_2 \stackrel{aab}{\rightarrow} T_2'$ with

$$T_2' = ((b{:}\lambda){\cdot}(\lambda{\vee}(c{:}C))){\wedge}(b\_or\_c{\gg}(BC{\cdot}(c{:}\lambda))),$$

and there exists no T_2'' s.t. $T_2' \stackrel{c}{\rightarrow} T_2''$, since the only possible transition from T_2' is $T_2' \stackrel{b}{\rightarrow} T_2''$, with

$$T_2'' = (\lambda{\vee}(c{:}C)){\wedge}(b\_or\_c{\gg}((\lambda{\vee}(b{:}BC{\cdot}(c{:}\lambda))){\cdot}((c{:}\lambda){\cdot}(c{:}\lambda)))),$$

and $[\![T_2'']\!] = \{cc\}$.

4.4 Comparison with the LTL

In this section we formally prove that trace expressions are more expressive than the LTL, when both formalisms are used for RV. To this purpose we consider the LTL_3 semantics [17], an adaptation of the standard semantics of LTL formulas expressly introduced to take into account the limitations of RV due to its inability to check infinite traces. Despite there are LTL formulas which do not have an equivalent trace expression according to the standard LTL semantics, when LTL_3 is considered such a difference is no longer exhibited: for any LTL formula φ it is possible to build a contractive and deterministic trace expression τ such that the monitors generated by φ and τ, respectively, are behaviorally equivalent.

4.4.1 Comparing trace expressions and LTL

We have shown in Section 1.5 that LTL formulas as $p\,U\,q$ cannot be fully verified at runtime, therefore a three-valued semantics LTL_3 has been introduced. To be able to compare LTL formulas with trace expressions, the same three-valued semantics is considered for trace expressions as well.

Given a finite trace $\sigma \in \Sigma^*$ of length $|\sigma| = n$, a continuation of σ is an finite or infinite trace $u \in \Sigma^* \cup \Sigma^\omega$ s.t. for all $0 \leq i < n$ $u(i) = \sigma(i)$.

The three-valued semantics of a trace expression τ is defined as follows:

$$\sigma \in [\![\tau]\!]_3 = \begin{cases} \top & \text{iff } u \in [\![\tau]\!] \text{ for all continuations } u \text{ of } \sigma \\ \bot & \text{iff } u \notin [\![\tau]\!] \text{ for all continuations } u \text{ of } \sigma \\ ? & \text{iff neither of the two conditions above holds} \end{cases}$$

Let us consider again the formula $\varphi = pUq$; if we assume that each atomic predicate in AP has a corresponding event type denoted in the same way, then the closest trace expression τ into which φ can be translated is defined by $T = p{:}T \vee q{:}1$, where 1 is the derivable constant introduced in Section 4.3 denoting all possible traces. If we consider the standard semantics we have that, since $\{p\}$ is an event that satisfies p, $\{p\}^\omega \in [\![\tau]\!]$, but $\{p\}^\omega \not\models \varphi$. However, when considering the three-valued semantics we have that for all $v \in \{\top, \bot, ?\}$ and $\sigma \in \Sigma^*$, $\sigma \vDash \varphi = v$ iff $\sigma \in [\![\tau]\!]_3 = v$. In particular, for all $n \geq 0$, $\{p\}^n \vDash \varphi =?$ and $\{p\}^n \in [\![\tau]\!]_3 =?$.

To translate an LTL formula φ into a trace expression τ s.t. the three-valued semantics is preserved, we exploit the result presented in Section 1. First, φ is translated into an equivalent FSM $\mathcal{M}^\varphi$, then $\mathcal{M}^\varphi$ is translated into an equivalent contractive and deterministic trace expression τ^φ. The latter translation is defined as follows:

- if the initial state returns $\top$, then φ is a tautology, and the corresponding trace expression is the constant 1;
- if the initial state returns $\bot$, then φ is a unsatisfiable, and the corresponding trace expression is the constant 0;
- if the initial state returns ?, then the corresponding trace expression is defined by a finite set of equations $X_1 = \tau_1, \ldots, X_n = \tau_n$, where n is the number of states in $\mathcal{M}^\varphi$ that return ?, each of such states is associated with a distinct variable X_i, X_1 is the variable associated with the initial state which corresponds to the whole trace expression τ^φ.

  The expressions τ_i are defined as follows: let k be the number of states $q_1, \ldots, q_k$ that do not return $\bot$ for which there exists an incoming edge, labeled with the element $a_i \in 2^{AP}$, from the node associated with X_i; we

know that $k > 0$, because the node associated with X_i returns ?. Then $\tau_i = a_1{:}f(q_1) \vee \ldots \vee a_k{:}f(q_k)$, where $f(q)$ is defined as follows: if q returns $\top$, then $f(q) = 1$, otherwise (that is, q returns ?), $f(q) = X_q$ (that is, the variable uniquely associated with q is returned).

Since all variables in the expressions $\tau_1, \ldots, \tau_n$ are guarded by the prefix operator, τ^φ is contractive; furthermore, it is deterministic because $\mathcal{M}^\varphi$ is deterministic.

Theorem 4.4. *Let $\mathcal{M}^\varphi$ be the FSM equivalent to φ generated by the procedure described in Section 1. Then, the trace expression τ^φ generated from $\mathcal{M}^\varphi$ as specified in Section 4.4.1 preserves the semantics of $\mathcal{M}^\varphi$: for all $\sigma \in \Sigma^*$ $\mathcal{M}^\varphi$ accepts σ with output $v \in \{\top, \bot, ?\}$ iff $\sigma \in [\![\tau^\varphi]\!]_3 = v$.*

Proof: The proof proceeds by induction on the length of σ.
Base case: $\sigma = \varepsilon$.
The cases where the initial state of the FSM returns $\top$ or $\bot$ are immediate to be proved (in the case of $\bot$ the monitoring ends). The proof when the initial state returns ? is based on the fact that by construction $[\![\tau^\varphi]\!] \neq \emptyset$ and there always exists a trace u s.t. $u \notin [\![\tau^\varphi]\!]$, therefore $\varepsilon \in [\![\tau^\varphi]\!]_3 =?$. □
Inductive step: $\sigma = \sigma' e$.
Inductive hypothesis: $\mathcal{M}^\varphi$ accepts σ' with output $v \in \{\top, \bot, ?\}$ and $\sigma' \in [\![\tau^\varphi]\!]_3 = v$.
Without loss of generality, we consider that the monitoring ends immediately when an unexpected event occurs; that is when the FSM visits a $\bot$ node and the trace expression can not move to another state consuming the occurred event. Consequently, $\mathcal{M}^\varphi$ accepts σ' with output $v \in \{\top, ?\}$ and $\sigma' \in [\![\tau^\varphi]\!]_3 = v$.
Analyzing all the possible cases:

- if $v = \top$, for the *inductive hypothesis*, the current state in FSM returns $\top$ and the corresponding trace expression is the constant 1 (by construction); consuming the event e both remain in a state which accepts all incoming events.
- if $v =?$, for the *inductive hypothesis*, the current state in the FSM returns ? and the corresponding trace expression is an equation of the form $X = \tau$ (by construction), which is built as described above (in the previous page). Consequently,

- if the event e brings the FSM in the state $\top$, the trace expression moves in the constant 1 by construction. Both return $\top$;
- if the event e brings the FSM in the state $\bot$, the trace expression can not move to another state by construction and the monitoring ends returning $\bot$. Both return $\bot$;
- if the event e brings the FSM in a state ?, the trace expression moves in a new state represented by an equation of the form $X = \tau$ which is built as described above. Both return ?.

In Section 4.3.3 we have shown a trace expression τ that specifies a non context free language of traces (when only finite traces are considered). More formally, $\sigma \in [\![\tau \cdot 1]\!]_3 = \top$ iff $\sigma \in \{a^n b^n c^n \mid n \geq 0\}$.

This means that for RV (that is, when the three-values semantics is considered) trace expressions are strictly more expressive than LTL, since the LTL is less expressive than ω-regular languages.

4.5 Template trace expressions

As already anticipated in Section 4.2.3.1, to be able to write more complex protocols, we can extend our formalism introducing template trace expressions. The main novelty of template trace expressions w.r.t. trace expressions is the presence of *parameters* inside the protocol definition.

In Section 4.2.3.1 we have presented the *finite_composition* operator as a "meta-operator", consequently, also the template trace expressions are a sort of "meta-formalism", in the sense they cannot be directly used as they are. Indeed, they are templates which must be applied to some arguments in order to obtain plain trace expressions. Parameters are present only in the template definition: when template trace expressions are actually used either for runtime verification or for protocol driven behaviour generation, all terms must be ground, i.e. variables must have already been instantiated.

In order to better explain this new trace expressions formalism extension, we introduce it with an example.

Here is how we would have represented a client-server protocol with trace expressions.

```
SERVER =
  receive_request(client1):
    (serve_request(client1):
      SERVER).
```

An example of a correct trace would be:

```
receive_request(client1):
  (serve_request(client1):
    (receive_request(client1):...))
```

where ... indicates that the trace is infinite.

The `SERVER` protocol is a client-server protocol made up by a loop in which the server receives a `serve_request` from the client1 replying with a `receive_request`.

If we would like to change the client we should modify the protocol. This is not very convenient because our protocols support switch interactions and we could take advantage of this feature in order to change, for instance, the clients that communicate with the server.

Using instead template trace expressions we can avoid this problem, defining the protocol as follows:

```
SERVER =
  receive_request(var(1)):
    (serve_request(var(1)):
      SERVER).
```

In this way we have written a generic protocol where we can change the involved agents simply changing the domain of parameter `var(1)`.

The domain of `var(1)` is set during the application stage. In fact, a template trace expression must be "applied" in order to turn into a "normal" trace expression, which can be used as described in Chapter 3. In particular, after the application

stage, the obtained global protocol must be projected. The projection is still a trace expression, where events not involving the agents in the given set are removed.

Let us consider a more complex example:

```
SERVER1 =
  receive_request(client1):
    (serve_request(client1):
      SERVER1),
SERVER2 =
  receive_request(client2):
    (serve_request(client2):
      SERVER2),
SERVER3 =
  receive_request(client3):
    (serve_request(client3):
      SERVER3),
SERVER = SERVER1|(SERVER2|SERVER3)
```

In this example, we have three protocol branches, each of which is similar to the simple protocol described before, combined using the shuffle operator. The first branch involves `client1`, the second `client2`, the third `client3`. As we can see we have to write the same piece of code many times and above all it is impossible to change agents at runtime, for example during a protocol switch.

Instead, using template trace expressions, we may write:

```
SERVERT =
  receive_request(var(1)):
    (serve_request(var(1)):
      SERVERT),
SERVER = finite_composition(|, SERVERT, [m(var(1), [])])
```

The construct `finite_composition`, as already defined in Section 4.2.3.1, is used to compose many times a trace expression with a chosen operator, in this case the *shuffle* operator. If we iterate the `var(1)` parameter on the set containing

{client1, client2, client3} we get the same results as before, without var(1). The great advantage of this approach, is that the set over which var(1) ranges can be decided at runtime, hence allowing the agents to implement a (limited) form of **dynamic protocol generation**.

An important aspect, which has been already analyzed in Section 4.2.3.1, is the presence of a list of modifiers in the finite_composition operator. A modifier allows changing a parameter iteration range.

Since the wide use of template trace expressions following in the work, it is important to make an example showing the mechanism of modifiers.

Supposing that we have some trace expressions which iterate on var(1), var(2), var(3), var(4) and var(5). An example of a list of modifiers, which might appears in a nested finite_composition operator, could be:

```
[
  m(var(3), [remove(var(1))]),
  m(var(4), [add(var(1))]),
  m(var(5), [add(var(1)), remove(var(2))])
]
```

where

- var(3) iterates on all its previously set values **except** the current value of var(1);
- var(4) iterates on all its previously set values **plus** the current value of var(1);
- var(5) iterates on all its previously set values **except** the current value of var(1) and **plus** the current value of var(2).

To understand how the modifiers are used we can see this simple example:

```
Alice =
  hello_world(alice, var(2)):lambda,
SayHello =
  finite_composition(|, Alice, [m(var(2), [remove(var(1))])]),
```

```
HelloWorld =
  finite_composition(\/, SayHello, [m(var(1), [])]).
```

In this case, we create a protocol where the main trace expression consists of a `union` operator; each its branch is a `shuffle` where each branch contains `hello_world(alice, var(2))` with `var(2)` which can take all values previously set for it except (we have used the `remove` keyword) for the current value of `var(1)`. In this way, we can have parameters which iterate on a set of values except (`remove` keyword) or plus (`add` keyword) a fixed parameter.

In order to better explain the application and projecting phases, we can see (with a short piece of *code*[6]) some sample calls:

```
SERVERT =
  (receive_request(var(1))):
    (serve_request(var(1)):
      SERVERT),
SERVER = finite_composition(|, SERVERT, [m(var(1), [])]),
apply(SERVER,
      [t(var(1), [client1, client2, client3])],
      INSTANTIATEDSERVER),
project(INSTANTIATEDSERVER, [agent1], PROJECTEDSERVER).
```

The *apply* predicate instantiates the `SERVER` protocol returning its instantiation in `INSTANTIATEDSERVER` variable. Afterwards, the obtained "normal" trace expression without parameters can be projected; in our example, the projection is on an agent called `agent1`.

After the application and projection phases we obtain a "customized" protocol driven agent. This agent will have to only choose what to do during its execution on the basis of what is expected by the protocol; the respect of the global protocol is guaranteed because each agent directly derives from it via projection, and each agent is guided by the same interpreter.

[6]The `project` function was previously described in Section 3.1.3 and its implementation will be discussed in Chapter 6. The `apply` function, instead, is the function necessary for the protocol instantiation and we will analyze it better in Chapter 6.

Chapter 5

Examples

In the following sections we present some examples that have been developed in order to show the potential of our approach.

5.1 Iterated Contract Net Protocol

The protocol in this example has been developed in order to guide protocol-driven agents and also to make runtime verification. In particular, it does not take advantages from the protocol switches, indeed, it is defined using only one trace expression. This trace expression, as just said, could be used both to generate agents which directly follows the protocol and also to create monitors which can check at runtime the protocol.

The FIPA Iterated Contract Net Interaction Protocol (ICNP) [52] is an extension of the basic FIPA Contract Net IP, but it differs by allowing multi-round iterative bidding.

As with the FIPA Contract Net IP, the initiator issues m initial call for proposals with the *cfp* act. Of the n participants that respond, k are propose messages from participants that are willing and able to do the task under the proposed conditions and the remaining j are from participants that refuse.

Of the k proposals, the initiator may decide if this is the final iteration and it can accept p of the bids ($0 \leq p \leq k$), and consequently it rejects all the others.

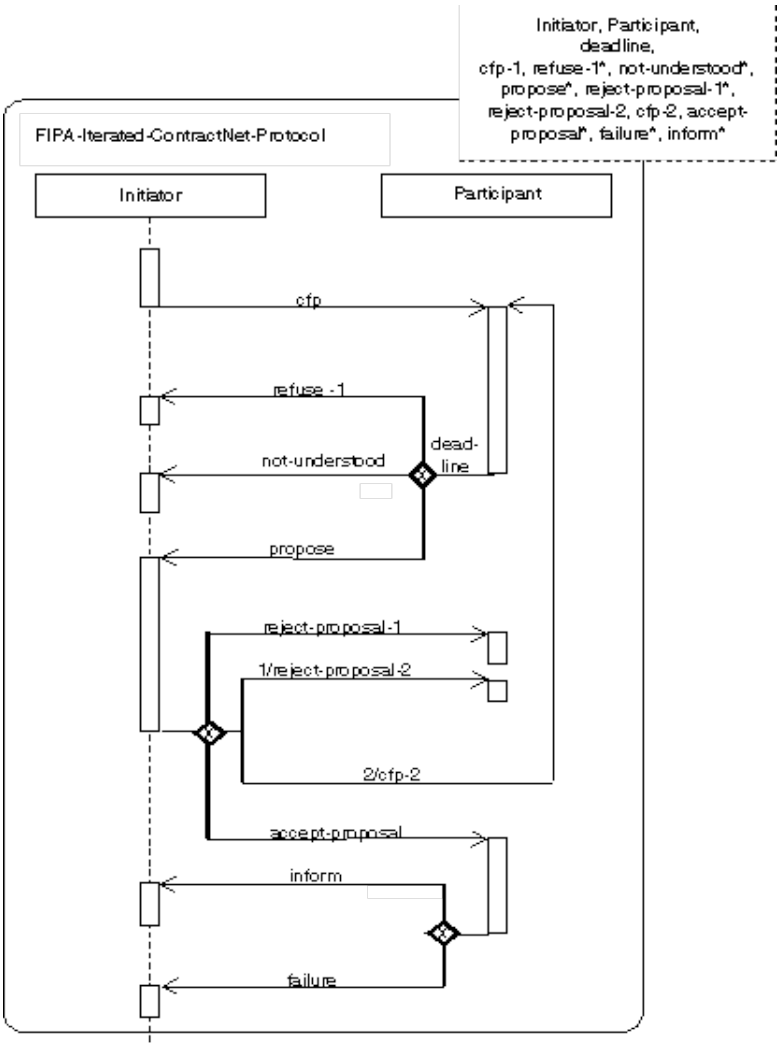


FIGURE 5.1: FIPA Iterated Contract Net Protocol from the FIPA Iterated Contract Net Interaction Protocol Specification (`http://www.fipa.org/specs/fipa00030/`).

Alternatively the initiator may decide to iterate the process by issuing a revised *cfp* to l of the participants and rejecting the remaining k-l participants.

The intent is that the initiator seeks to get better bids from the participants by modifying the call and requesting new (equivalently, revised) bids. The process terminates when the initiator refuses all proposals and does not issue a new *cfp*, the initiator accepts one or more bids or all the participants refuse to bid.

Below we represent the ICNP protocol using trace expressions formalism.

This is the first example which uses the template extension formalism.

Thanks to the *finite_composition* operator, we can write the trace expression *Agent* once for all agents[1].

```
Agent =
  (cfp(var(2), var(1))):
    (((refuse(var(1), var(2))):lambda) \/
    ((propose(var(1), var(2))):
      (((counter_propose(var(2), var(1))):Agent) \/
      (((reject_proposal(var(2), var(1))):lambda) \/
      ((accept_proposal(var(2), var(1))):
        (((inform(var(1), var(2))):lambda) \/
        ((failure(var(1), var(2))):lambda)))))),
```

The notations:

- `lambda`, instead of λ;
- $\backslash/$, instead of $\vee$;
- $/\backslash$, instead of $\wedge$;
- `finite_composition(|, T, [m(var(1), []), ...,m(var(n), [])])`, instead of $finite\_composition(|, T, \{(var(1), \{\}), ..., (var(n), \{\})\})$.

are a low level customization, this is the way we represent operators in Prolog (we will see better in detail in Section 6).

It is a short and compact protocol to represent using trace expressions formalism, indeed, the only operators we have to use are: *prefix*, *shuffle* and *union*. In this particular example, *switch* messages are not required and the protocol is characterized by only one trace expression (which consists in a `finite_composition` operator).

```
ICNP = finite_composition(|, Agent, [m(var(1), [])]).
```

In the next example, instead, we analyze a protocol where the protocol switch is necessary, as in the hobbit example (where, however, the template trace extension is used).

[1] In this case we have to initialize the `var(1)` parameter with a list containing all the agents inside the system.

5.2 Auction protocol

The protocol in this example has been developed in order to guide protocol-driven agents and also to make runtime verification. In particular, it does not take advantages from the protocol switches, indeed, it is defined using only one trace expression. This trace expression, as just said, could be used both to generate agents which directly follows the protocol and also to create monitors which can check at runtime the protocol.

In the FIPA English Auction Interaction Protocol [51], the auctioneer seeks to find the market price of a good by initially proposing a price below that of the supposed market value and then gradually raising the price. Each time the price is announced, the auctioneer waits to see if any buyers will signal their willingness to pay the proposed price. As soon as one buyer indicates that it will accept the price, the auctioneer issues a new call for bids with an incremented price. The auction continues until no buyers are prepared to pay the proposed price, at which point the auction ends. If the last price that was accepted by a buyer exceeds the auctioneer's (privately known) reservation price, the good is sold to that buyer for the agreed price. If the last accepted price is less than the reservation price, the good is not sold.

The auctioneer's calls, expressed as the general *cfp* act, are multi-cast to all participants in the auction. For simplicity, only one instance of the message is portrayed. Note also that in a physical auction, the presence of the auction participants in one room effectively means that each acceptance of a bid is simultaneously broadcast to all participants and not just the auctioneer. This may not be true in an agent marketplace, in which case it is possible for more than one agent to attempt to bid for the suggested price. Even though the auction will continue for as long as there is at least one bidder, the agents will need to know whether their bid (represented by the propose act) has been accepted. Hence the appearance in the IP of the accept-proposal and reject-proposal acts, despite this being implicit in the English Auction process that is being modeled.

Below we give two different protocol representations of FIPA English auction protoco in order to better explain the advantages in the use of the *intersection* operator. These two examples have been proposed by Luca Franceschini, a master student at University of Genoa, which, under my supervision, has implemented

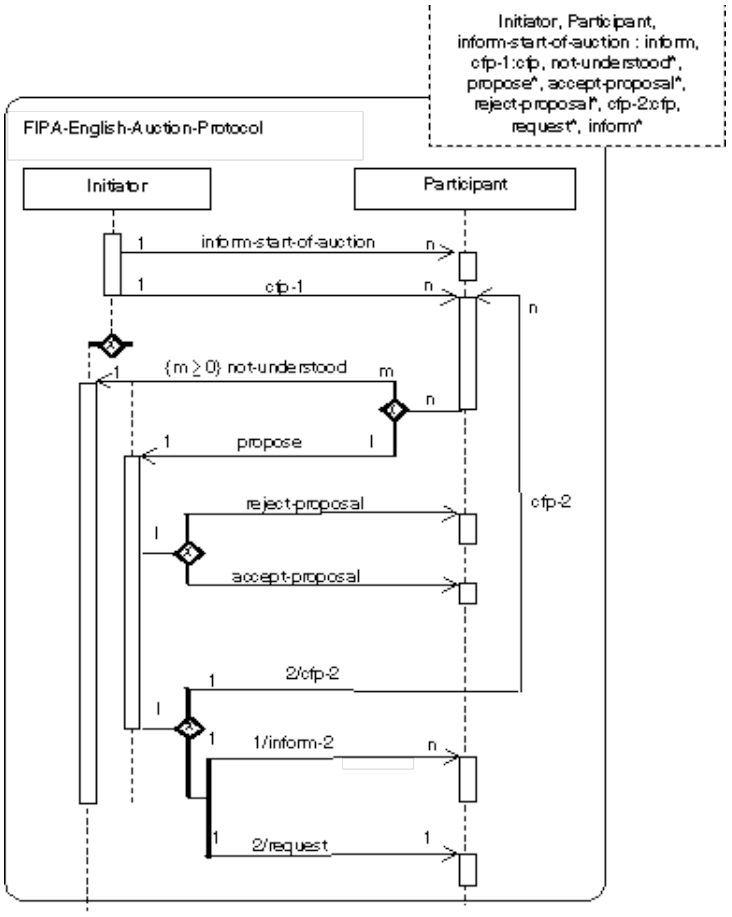


FIGURE 5.2: FIPA English Auction Protocol from FIPA English Auction Interaction Protocol Specification (`http://www.fipa.org/specs/fipa00031/`).

the FIPA English auction protocol as individual exercise of the Intelligent Systems and Machine Learning (ISML) course.

5.2.1 Solution without the *intersection* operator

In this case, the main trace expression is `Auction`.

The *initiator* sends the `inform_start` message to each *participant*. After that, it sends the first call for proposal `cfp_1`.

```
Auction = InformStart * Cfp1AndReply,
InformStart = finite_composition(|,
                inform_start(var(1)):lambda, [m(var(1), [])]),
Cfp1AndReply = finite_composition(|,
                 cfp_1(var(1)):CfpReply, [m(var(1), [])]) *
```

```
                AfterAllRepliesFirstTime,
```

A participant can answer with a `not_understood` message, or it can answer with a `propose` message waiting for a response from the *initiator*.

```
CfpReply = (propose(var(1)):lambda) \/
           (not_understood(var(1)):lambda),
```

The *initiator* may choose to accept a *participant* and to reject all the others, or it can reject all *participants*, thus ending the protocol without a winner.

```
AfterAllRepliesFirstTime = PriceRaise \/
                           (AllRejected * EndWithoutWinner),
PriceRaise = finite_composition(\/,
               PriceRaiseA, [m(var(1), [])]),
AllRejected = finite_composition(|,
                reject(var(1)):lambda, [m(var(1), [])]),
EndWithoutWinner = Inform2,
Inform2 = finite_composition(|,
            inform_2(var(1)):lambda, [m(var(1), [])]),
```

If the *initiator* accepts a *participant*, it must reject all *participants* except the winner (this is obtained using the `remove` modifier). After that, the *initiator* can restart the protocol sending a second call for proposal `cfp_2` to all *participants*, or it stops the protocol sending an `inform_2` message to all *participants* and after that it sends a `request` message to only the winner.

```
PriceRaiseA = (accept(var(1)):
                (finite_composition(|,
                  OneRejectedA, [m(var(2),[remove(var(1))])]))
                  * (Cfp2AndReply \/ EndWithWinner),
Cfp2AndReply = finite_composition(|,
                 (cfp_2(var(1)):CfpReply), [m(var(1), [])]) *
               AfterAllRepliesFirstTime,
OneRejectedA = reject(var(2)):lambda,
EndWithWinner = Inform2 * (Request \/ lambda),
Request = request(var(1)):lambda,
```

In this case, it is important to note the abuse of the `finite_composition` operator. This is due to the absence of synchronization which is not explicitly represented inside the protocol (this synchronization between all agents inside the system is necessary in the auction protocol).

In particular, if we consider the protocol focusing only on its trace expression representation, there should not be any problem, but we must remember that, at a certain point of the execution[2] (implementation side), we will have to instantiate the protocol and all parameters will disappear[3].

In our case, the trace expression used to formalize the FIPA English auction protocol could become very huge, this is due to the presence of a lot of `finite_composition` operators.

In order to solve this memory problem, we can take advantages from the use of the *intersection* operator (as already said in Section 4.2 the only new operator introduced in trace expressions w.r.t. constrained global types). The following trace expression corresponds to a different implementation of the auction protocol, where, instead of using a lot of `finite_composition` operators, we can collapse the trace expressions and synchronized them thanks to the *intersection* operator.

5.2.2 Solution with the *intersection* operator

In this case, the main trace expression is still `Auction`; it consists in a *intersection* operator where the left operand represents an abstract view of the auction protocol, while the right operand represents a low level view customized for each single *participant*. Also here, the `finite_composition` operator will be necessary to write the trace expression of the right side once for all *participants*.

```
Auction = (AuctionSynchronization /\
           finite_composition(|,
             AuctionSingleParticipantA, [m(var(1), [])])).
AuctionSynchronization = InformStart *
                            (Cfp1AndReply *
```

[2]We have already given some hints in Section 4.2.3.1 and in Section 4.5 but we will clarify the mechanism in Section 6.

[3]It is important to understand that a protocol must be ground and without parameters in order to be used by a protocol-driven agent.

```
                                (AfterPropose \/ lambda)),
```

Also here the protocol starts with the `inform_start` and the `cfp_1` messages. The dotted line is used to highlight the division between the two sides of the *intersection* operator; in this way, we can observe, step by step, the protocol from both points of view.

```
InformStart = ((inform_start:InformStart) \/ lambda),
Cfp1AndReply = (cfp_1:(CfpReply | (Cfp1AndReply \/ lambda))),
-----------------------------------------------------------------------
AuctionSingleParticipantA = (inform_start(var(1))):
                              (cfp_1(var(1)): CfpReplyLosingA),
```

The trace expression `CfpReply` is in *shuffle* with the trace expression `Cfp1AndReply`, in this way, in the left side, the protocol allows an arbitrary number of `cfp_1` messages, but this number is limited to the number of *participants* due to the right side which has a *shuffle* operator with as many branches as the *participants*.

The *participant* can answer with a `not_understood` message thus abandoning the protocol or it can answer with a proposal.

```
CfpReply = ((propose:lambda) \/ (not_understood:lambda)),
-------------------------------------------------------------------
CfpReplyLosingA = ProposeLosingA \/
                  (not_understood(var(1)):lambda),
ProposeLosingA = (propose(var(1)):(AcceptA \/ RejectLosingA)),
```

The *initiator* can

- accept a *participant* rejecting all the others; after that, it restarts the protocol knowing that there is a *participant* accepted (`CfpReplyWinningA`) or it stops the protocol with the chosen winner (`EndWinningA`);
- reject all *participants*; after that, it ends either the protocol without a winner (`EndLosingA`) or it restarts the protocol knowing that there was not yet a *participant* accepted (`CfpReplyLosingA`).

```
AfterPropose = (AcceptReject *
                 ((cfp_2:(Cfp2AndReply *
                   (AfterPropose \/ lambda))) \/ End)),
Cfp2AndReply = (CfpReply | ((cfp_2:Cfp2AndReply) \/ lambda)),
AcceptReject = ((accept:Reject) \/
               ((reject:AcceptReject) \/ lambda)),
Reject = ((reject:Reject) \/ lambda),
End = (Inform2 | ((request:lambda) \/ lambda)),
Inform2 = ((inform_2:Inform2) \/ lambda),
-----------------------------------------------------------------
AcceptA =
  (accept(var(1)):
    ((cfp_2(var(1)):CfpReplyWinningA) \/ EndWinningA)),
EndWinningA = (inform_2(var(1)):
                ((request(var(1)):lambda) \/ lambda)),
RejectLosingA = (reject(var(1)):
                  (EndLosingA \/ (cfp_2(var(1)):CfpReplyLosingA))),
EndLosingA = (inform_2(var(1)):lambda),
CfpReplyWinningA = (ProposeWinningA \/
                   (not_understood(var(1)):lambda)),
ProposeWinningA = (propose(var(1)):(AcceptA \/ RejectWinningA)),
RejectWinningA = (reject(var(1)):
                   (EndWinningA \/ cfp_2(var(1)):CfpReplyLosingA)),
```

The left side consists in the abstract representation of the protocol, while the right side represents a more low level view of the protocol; it is easy to note the different levels of abstraction because, in `AuctionSynchronization` all *event types* have no arguments, instead, in
`AuctionSingleParticipantA` all *event types* have arguments.

In order to better explain the latter concept, we can consider an example which is extracted from the protocol just described:

- `inform_start` is a general *event type* where all events (or in this particular case, all messages), that belong to it, are of the form *msg(initiator, Participant, inform_start)*, where *Participant* can be any possible *participant* agent.

- `inform_start(participant1)` is a low level *event type*, indeed, the only event belonging to it is *msg(initiator, participant1, inform_start)*.

In this way, we synchronize the left side with the right side. The left side is necessary and it is used to synchronize the communication between all agents, instead, the right side is used to bound the communications to only the events under the protocol.

This last example is the first which exploits the potential of trace expressions. The two versions of the protocol allow us to understand that, even if the trace expressions formalism is quite simple, without paying attention we risk to cause a state explosion (memory problem).

5.3 Hobbit protocol

The protocol in this example has been developed in order to guide self-adaptive protocol-driven agents. In particular, it takes advantages from the protocol switches, indeed, it is defined using more than one trace expression. These trace expressions, as just said, could be used only to generate agents which directly follows the protocol (it would not make sense as a protocol to monitor).

We implemented an example where agents are freely inspired by characters from the Lord of the Rings. The purpose of this toy example is to explain how our approach works in practice, in a simplified and fictitious scenario.

The notation used for specifying protocols is the one introduced in Chapter 4. We describe the normal protocol **bh** involving Bilbo and the hobbit, whereas, being very similar, we do not specify Frodo's, Folco's and Sam's normal behaviour that we assume to be governed by **fr**, **fo**, **sa** protocols. The exceptional protocol **ef** involves Bilbo, Frodo, Folco and Sam. Gandalf acts as the controller of the MAS and may require a protocol switch from the normal protocol to the exceptional one.

Protocol involving Bilbo and the hobbit. To enter the treasure room, any hobbit must ask Bilbo and must respect his decision to either let him in or not. In the first case the hobbit thanks Bilbo and visits the room; in the second case he expresses his disappointment and may either start the protocol again, or give up.

Bilbo allows in only one hobbit per day so the only knowledge he needs to check and update is related to whether one hobbit already entered the room today or not. Of course Bilbo can adopt other policies for deciding when and why allowing the hobbit in. We made experiments with different ones.

The branch of the protocol specifying the interactions between Bilbo and Hobbit1 is modeled by the following trace expression.

```
Hobbit1Branch =
ask_enter_treasure(hobbit1):
 ((ok_enter(hobbit1):thanks(hobbit1):lambda) \/
  (no_enter(hobbit1):grunt(hobbit1):
     (lambda \/ hobbit1Branch)))
```

The branches for all the other hobbit are the same as `Hobbit1Branch` apart from the presence of `hobbit2`, `hobbit3`, ..., instead of `hobbit1`.

The `ask_enter_treasure(H)` event type holds for communicative events `msg(H, bilbo, ask, enter_treasure)` where `H` is the sender, who must be one among the hobbit, `bilbo` is the receiver, `ask` is the performative, `enter_treasure` is the content.

This request may be followed (sequence operator `:`) either by an event of type `ok_enter(H)` stating that `H` can enter the room, followed by a thanks by `H`, and then conclude (`lambda`), or (union operator $\backslash/$) by Bilbo's refusal to let `H` in (`no_enter(H)`) followed by an expression of disappointment by `H` (`grunt(H)`), followed by either the hobbit branch protocol again, or `lambda`.

The `BilboHobbit` trace expression has as many different branches as the hobbit with whom Bilbo is expected to interact. The shuffle operator `|` is used to specify interleaving among events in these branches, which are also put in interleaving with `(switch(bilbo,ef):lambda)` where `msg(gandalf,R,switch,Pr)` $\in$ `switch(R, Pr)`. This means that `msg(gandalf,bilbo,switch,ef)` is allowed to be received in any moment:

```
BilboHobbit =
(((Hobbit1Branch|Hobbit2Branch)|
   Hobbit3Branch)|...)|(switch(bilbo,ef):lambda).
```

Once projected onto Bilbo, `BilboHobbit` returns `BilboHobbits` itself as Bilbo is involved in any event and none can be discarded. When projected onto `hobbitj`, `BilboHobbits` returns `HobbitjBranch` which drives the behaviour of the `j`-th hobbit.

Exceptional Protocol. If an emergency takes place, Bilbo should ask Frodo to help him moving the treasure to a safer place. If Frodo can help Bilbo, he gives a positive answer, otherwise he asks Sam and Folco (no matter in which order). Only after these events take place (concatenation operator `*`), all the agents are ready to manage a new switch request from Gandalf, asking them to recover to their normal protocol. The `ExceptionalFlow` protocol, identified by `ef`, is specified by the following trace expressions:

```
ExceptionalFlow =
  (ask_help(bilbo,frodo):
    ((ok_help(frodo,bilbo):lambda) \/
    (cannot_help(frodo,bilbo):
      (ask_help(frodo,sam):lambda |
       ask_help(frodo,folco):lambda)))) * Recover,
Recover =
  ((switch(frodo,fr):lambda) |
   (switch(bilbo,bh):lambda) |
   (switch(sam,sa):lambda) |
   (switch(folco,fo):lambda)).
```

Gandalf acts as the controller. The protocol that drives his behaviour is

```
Gandalf =
  ((switch(frodo,ef):lambda) |
   (switch(bilbo,ef):lambda) |
   (switch(sam,ef):lambda) |
   (switch(folco,ef):lambda)) * Recover.
```

where `Recover` is defined as in the `ExceptionalFlow` protocol. The decision of sending messages to Frodo, Bilbo, Sam and Folco is fired by the (paranormal) perception of a crowd of ogres coming near to the treasure room. After that (concatenation operator `*`) Gandalf sends a message to all the agents to switch

back to their normal behaviour. The constraint that when one protocol switch message is associated with the SwitchMsg variable, no other switch request can enter the message queue, is respected. In fact Frodo, Bilbo, Sam and Folco are able to manage switch requests at any time: as soon as they receive the first request by Gandalf they manage it and set SwitchMsg to null. When Gandalf sends the second request, they keep it in SwitchMsg until they complete the management of the exceptional situation and become ready to switch back to their normal life.

5.4 Secret protocol

The protocol in this example has been developed in order to guide self-adaptive protocol-driven agents. In particular, it takes advantages from the protocol switches, indeed, it is defined using more than one trace expression. These trace expressions, as just said, could be used only to generate agents which directly follows the protocol.

In this section we analyze a protocol which is interesting from point of view of the safety. We can summarize the main phases of the protocol in:

1. When the protocol starts, there are only agents which have a *normal* behaviour; each agent can send (receive) messages to (from) each other agent. The only special agent is the *boss* agent that is a privileged agent ables to request a protocol switch to two agents promoting them to be *secret* agents.
2. The *boss* agent promotes the two chosen agents.
3. Afterwards, there are both *normal* and *secret* agents. A *secret* agent can communicate only with another *secret* agent; if a *normal* agent try to speak to a *secret* agent, the second must refuse the communication (with a fixed rejection message).
4. At the end, the *boss* agent requests a protocol switch to the two *secret* agents downgrading them to be *normal* agents.
5. The protocol restarts at the point 1.

Unlike the ICNP protocol example[4], this is a cyclic protocol[5], indeed, it never ends. Now we can introduce the corresponding trace expression.

We can represent the agents normal behaviour with the `NormalAgent` trace expression.

```
Communication =
  (tell_me(var(1), var(2))):
    (((say_to(var(2), var(1))):Communication) \/
    ((shut_up(var(2), var(1))):Communication)),
Communications =
  finite_composition(|,
    finite_composition(\/,
      Communication,
      [m(var(2), [remove(var(1))])]), [m(var(1), [])]),
Switch1 = ((switch(var(3), var(4))):lambda),
Switch2 = ((switch(var(4), var(3))):lambda),
NormalAgent = ((Communication|Switch1)|Switch2).
```

Where:

- `var(1)` is a list containing all *normal* agents;
- `var(2)` is a list containing all *normal* agents;
- `var(3)` is a chosen agent which will be promoted to be *secret*;
- `var(4)` is a chosen agent which will be promoted to be *secret*.

In the first line we can describe how an agent communicates with another agent; after, a composition is done using the specific operator. The iteration is done on the two parameters `var(1)` and `var(2)`; in particular, the two `finite_composition` operators are nested, the outermost iterates on `var(1)`, while the innermost iterates on `var(2)` except (thanks to the `remove` keyword) the current value of `var(1)`[6].

When an agent becomes a *secret* agent, it switches to a different protocol.

[4]Even the ICNP could be represented as a cyclic protocol.

[5]Usually, all protocols defined with our formalism are cyclic.

[6]It is necessary, in this case, because a communication like `(tell_me(agent1, agent1)):...` is useless.

```
SecretSpeak1 = (tell_me(var(1), var(2))):
                 (((say_to(var(2), var(1))):SecretSpeak1) \/
                 ((shut_up(var(2), var(1))):SecretSpeak1)),
SecretSpeak2 = (tell_me(var(2), var(1))):
                 (((say_to(var(1), var(2))):SecretSpeak2) \/
                 ((shut_up(var(1), var(2))):SecretSpeak2)),
```

These two trace expressions represent the communication between the two *secret* agents. We can note that, as already said before, a *secret* agent can speak only to another *secret* agent.

The parameters `var(1)` and `var(2)` represent the two *secret* agents.

```
NoSpeakAgent1 = (tell_me(var(3), var(1))):
                  ((shut_up(var(1), var(3))):NoSpeakAgent1),
NoSpeakAgents1 = finite_composition(|,
                   NoSpeakAgent1, [m(var(3), [])]),
NoSpeakAgent2 = (tell_me(var(3), var(2))):
                  ((shut_up(var(2), var(3))):NoSpeakAgent2),
NoSpeakAgents2 = finite_composition(|,
                   NoSpeakAgent2, [m(var(3), [])]),
```

If a *secret* agent receives a `tell_me` message from a *normal* agent, it answers with a `shut_up` message. Also here we take advantage of the `finite_composition` operator in order to avoid replications of the same trace expression for each agent.

```
SwitchBack = ((switch_back(var(1), var(2))):lambda),
SecretAgent = (SecretSpeak1 |
               (SecretSpeak2 |
               (NoSpeakAgents1 |
               (NoSpeakAgents2 |
                SwitchBack)))).
```

As already said at the beginning of this section, our protocol is cyclic; consequently, each *secret* agent can receive a protocol switch request, in this way a *secret* agent returns to be a *normal* agent which follows the `NormalAgent` behaviour.

The last trace expression represents the *boss* agent behaviour.

```
Boss = ((switch(var(2), var(1))):
         ((switch(var(1), var(2))):
           ((switch_back(var(2),var(1))):
             ((switch_back(var(1), var(2))):Boss)))).
```

This agent is very simple because it must only require protocol switches to the two *secret* agents.

Chapter 6

Implementation

In this chapter we present all the technical and design aspects of our framework. The chapter is divided into three parts: the first introduces all requirements which must be satisfied by an agent in order to become a self-adaptive protocol-driven agent, the second presents a high level version of the interpreter and all functions that must be implemented by a self-adaptive protocol-driven agents to be able to reason about the protocol (which way to go inside the possible choices, how to react to a particular state of the protocol like a message reception, etc...), the last part, instead, is the most technical one, with all the details of the Prolog code implementing the next, project and generate functions, and the implementations of our framework in Jason and JADE tools.

6.1 Requirements

A self-adaptive protocol-driven agent Ag is characterized by the following components:

• A unique agent identifier.

• A knowledge base $KBase$ supporting the operations $?K$ (is knowledge formalized as K available in $KBase$, and which actual value is associated with it?), $+K$ (add knowledge formalized as K to $KBase$) and $-K$ (remove knowledge formalized as K from $KBase$). We assume that the knowledge base includes information on

1. the precedence policy stating which kind of communicative action (sending, `prec(send)`, or receiving, `prec(rec)`) the agent should give the precedence to, in case the protocol's state allows interactions of both kinds. Like any other piece of knowledge in the knowledge base, this information may change during time. However, some protocols[1] work only when the agents involved in them follow consistent precedence policies, and changing the policy at runtime may lead to deadlocks or unwanted behaviours;

2. the timeout `timeout(T)` stating for how long the agent can wait for being able to either sending or receiving a message; the agent's alarm is set to `T` any time a communicative action takes place and expires after `T` time units;

3. the list of agents which, in *Ag*'s opinion, have the power to impose a protocol switch (`empowered(AgList)`). This list can change at runtime, for example because the trust of *Ag* towards some agents changes.

• A representation *Env* of the environment supporting external actions that the agent performs on the environment via effectors, and sensing actions performed by the agent via sensors. As customary, we make no assumptions on how sensors and effectors are implemented and we leave out from our investigation how the environment representation is kept consistent with the actual environment's state.

• A symbolic representation for all events which occur inside the framework. As already anticipated in the *events* paragraph in Section 3.2.5, in the case of communicative events the translation from a message to its symbolic representation can be very easy but, if we consider a generic event, the translation could be more difficult.

• The currently executing protocol *Pr*, whose syntax has been introduced in Section 4.2.

• A message queue *MsgQueue* storing interactions *Intr* corresponding to incoming messages that still have to be processed. The syntax of *Intr* is given in Section 3.2.5. Switch protocol requests have highest priority and always reach the top of the queue, when pushed into it. These messages are the only ones which are

[1]For example the protocol where Alice sends an arbitrary number of "ping" to Bob who answers with the same number of "pong", and then the protocol starts again, requires a synchronization between Alice and Bob in such a way that Bob avoids answering as soon as it receives a "ping" message, because Alice might want to send more "ping"s: Alice should give the precedence to sending and Bob to receiving.

not tagged as "unexpected" when received in a protocol's state where they were not expected. In this case, they are locally stored[2] and managed as soon as the protocol reaches a state where a switch protocol request can be accepted.

6.2 Design

6.2.1 Policies

As already anticipated in Section 3.2.5 and at the beginning of this chapter, even if a self-adaptive protocol-driven agent is totally guided by the protocol, it must be able to:

- react to a message reception;
- reason on the possible choices when the protocol provides more than one path;
- clean the knowledge base after a switch request;
- manage an unexpected message.

The agent must be equipped with: • A received message reaction policy, stating how to update the knowledge base and the environment's representation as a consequence of the reception of one incoming message among the allowed ones (as returned by the *generate* function). The policy implementation may require to perform internal or sensing actions and, as a side effect, may affect the agent's knowledge base and the environment. No communicative actions can be performed as part of the policy.

react: $Intr \times KBase \times Env \rightarrow KBase \times Env$

• An outgoing message selection policy, stating which outgoing interaction (if any) among the allowed ones the agent will perform. The selection policy depends on the knowledge base and on the environment's representation and affects both of them. Like for the reaction policy, its implementation may rely on internal and

[2]We assume that agents cannot have more than one pending protocol switch request at a time. Although the assumption is strong, it allows us to keep the interpreter implementation simple. It can be relaxed using a queue of pending requests instead of a variable.

sensing actions but not on communicative ones. The selection policy returns `null` in case either there are messages allowed by the protocol but none of them can be sent for reasons wired into the agent's selection function, or there are no messages at all.

select: $\mathcal{P}(Intr \times Pr) \times KBase \times Env \rightarrow ((Intr \times Pr) \bigcup \{\text{null}\}) \times KBase \times Env$

• A cleanup policy stating what actions the agent should perform before switching to another protocol.

cleanup: $KBase \times Env \rightarrow KBase \times Env$

• An unexpected message management policy stating what to do when a normal message not foreseen by the protocol is received, or when a switch message is sent by an agent who has not the power to act as a controller. The policy may vary according to the protocol currently under execution.

unexpected: $Intr \times KBase \times Env \rightarrow KBase \times Env$

The operational semantics of protocol-driven agents is given by the following interpreter which can call the *generate* and *project* functions.

For clarity of the presentation, we leave the knowledge base and environment components out of the arguments and return values of *react*, *select*, *cleanup*, *unexpected*. All these functions take the current knowledge base and environment and can update them.

6.2.2 The interpreter

In this section we present a high level version of our interpreter leaving all implementation aspects in Section 6.3.

Let KBase_0, Env_0, Pr_0, MsgQueue_0 be the initial knowledge base, environment, global protocol and message queue of Ag, respectively. A SwitchMsg variable is used to store the pending protocol switch request. An alarm (the Al variable) is initially associated with the Timeout value set by the agent's designer. A mechanism for checking whether the Al expired should be available.
Following Prolog's syntax, we use the "_" symbol to identify variables whose value does not matter.

Initialization

KB = KBase_0;

En = Env_0;

Pr = project(Pr_0, {Ag});

MQ = MsgQueue_0;

SwitchMsg = null;

?timeout(T);

Al = set(T);

Interpreter

while true {

/ C1: If a switch protocol request has been received, it is assigned to SwitchMsg and removed from the message queue */*

if (top(MQ) == msg(S, Ag, switch, PrSwitch))

SwitchMsg = pop(MQ);

/ The couples (interaction, next prot. state) allowed in the current state of the protocol are generated. For sake of presentation, we divide them into those involving incoming messages (InMsgs) and those involving messages that can be sent (OutMsgs) */*

InMsgs $\bigcup$ OutMsgs = generate(Pr);

/ C2: If the reception of a switch protocol request is allowed in the current state of the protocol, and there is such a request, it is managed: SwitchMsg is reset */*

if (SwitchMsg == msg(S, Ag, switch, PrSwitch) $\wedge$

(msg(S, Ag, switch, PrSwitch), _) $\in$ InMsgs)

{ SwitchMsg = null;

/ C2.1 If the sender has the power to request a protocol switch, then the cleanup actions are performed according to the "cleanup" policy, the protocol is changed to the projection of PrSwitch onto* Ag*, and the current interpreter cycle is exited, otherwise the protocol switch request is managed by the "unexpected" policy */*

if (?empowered(AgList) $\wedge$ S $\in$ AgList)

/* Protocol switch */

{ cleanup(); Pr = project(PrSwitch, {Ag}); }

else unexpected(msg(S, Ag, switch, PrSwitch));

continue;

}

/ Note that, if condition C2 is not met, namely either there is no switch request or the*

*protocol does not allow to manage it, nothing is done. If there is a switch request, it remains associated with variable SwitchMsg for being managed later, when the protocol will allow it */*

/ C3: If the message queue is empty, the alarm has expired, and there are no messages to send, the agent is stuck: the interpreter exits its main loop */*

if (empty(MQ) $\wedge$ expired(Al) $\wedge$ select(OutMsgs) == null;)
    break;

/ C4: If the message queue is empty, the alarm has not expired, and the agent gives the precedence to reception, the current interpreter cycle is exited in the hope that, at the next one, some message will be available in the message queue; there is no risk to always remain in this condition as the alarm sooner or later will expire */*

if (empty(MQ) $\wedge$ $\neg$expired(Al) $\wedge$?prec(rec))
    continue;

/ C5: If the message queue is not empty and the top message is not among those expected by the protocol (condition (top(MQ),_) $\notin$ InMsgs which includes the case InMsgs == $\emptyset$), the message is unexpected. It is popped and managed according to the agent's "unexpected" policy */*

if ($\neg$empty(MQ) $\wedge$ (top(MQ),_) $\notin$ InMsgs)
    { Msg = pop(MQ); unexpected(Msg); continue; }

/ C6: If the message queue is not empty, the top message is among the expected ones (this condition is superfluous but allows us to name the Pr_n component for successive use) and either the agent gives the precedence to reception and the current state of the protocol allows to receive messages, or the agent gives the precedence to sending but there are no messages to send, the first message in the queue is popped, the knowledge base and environment are updated according to the "react" policy, the protocol moves to the new state Pr_n and the alarm is reset */*

if ($\neg$empty(MQ) $\wedge$ (top(MQ),$\mathrm{Pr_n}$) $\in$ InMsgs $\wedge$
  (?prec(rec) $\vee$ (?prec(send) $\wedge$ select(OutMsgs) == null))
    { Msg = pop(MQ); react(Msg); Pr = $\mathrm{Pr_n}$;
      ?timeout(T); Al = set(T); continue; }

/ C7: The interpreter reaches this point if either the agent gives the precedence to sending, or it cannot wait for receiving messages because the message queue is empty*

and the timeout expired: the outgoing message selection is performed according to the "select" policy; the result of the selection cannot be `null` *otherwise C3 would have been verified; the selected message is sent; the protocol moves to the next state; the alarm is reset */*

(Msg,Pr_n) = select(OutMsgs); send(Msg);

Pr = Pr_n; ?timeout(T); Al = set(T); }

6.2.2.1 Extending the interpreter to cope with template trace expressions

In the protocol switch cases (*C1* and *C2* statements of the interpreter), if we want to manage template trace expressions instead of trace expressions, some changes are necessary some changes due to the fact that all protocols can contain parameters which have to be instantiated (we highlight these changes in **bold**).

/ C1': If a switch protocol request has been received, it is assigned to SwitchMsg and removed from the message queue */*

if (top(MQ) == msg(S, Ag, switch, (PrSwitch, **Parameters**)))

SwitchMsg = pop(MQ);

/ The couples (interaction, next prot. state) allowed in the current state of the protocol are generated. For sake of presentation, we divide them into those involving incoming messages (InMsgs) and those involving messages that can be sent (OutMsgs) */*

InMsgs $\bigcup$ OutMsgs = generate(Pr);

/ C2': If the reception of a switch protocol request is allowed in the current state of the protocol, and there is such a request, it is managed: SwitchMsg is reset */*

if (SwitchMsg == msg(S, Ag, switch, (PrSwitch, **Parameters**)) $\wedge$

(msg(S, Ag, switch, PrSwitch), _) $\in$ InMsgs)

{ SwitchMsg = null;

/ C2.1' If the sender has the power to request a protocol switch, then the cleanup actions are performed according to the "cleanup" policy, the protocol is instantiated obtaining PrSwitchInstantiated, the instantiated protocol is changed to the projection of PrSwitchInstantiated onto* Ag*, and the current interpreter cycle is exited, otherwise the protocol switch request is managed by the "unexpected" policy */*

if (?empowered(AgList) $\wedge$ S $\in$ AgList)

/* Protocol switch */

{

```
        cleanup();
        instantiate_template(PrSwitch, Parameters, PrSwitchInstantiated);
        Pr = project(PrSwitchInstantiated, {Ag});
       }
      else unexpected(msg(S, Ag, switch, (PrSwitch, Parameters)));
      continue;
    }
```

All the other cases of the interpreter remain unchanged.

Even if this change may seem to have limited effects, it allows us to give a huge support for dynamic protocol generation. Indeed, the *Parameters* variable can take on different values at runtime, on the base of the agent's knowledge base.

In order to better explain the dynamic protocol generation importance, we can consider a typical case, where the agents which participate to the protocol can change at runtime. Without template trace expressions, we could not formalize the protocol in a way that it supports adding and removing agents at runtime.

For example, the protocol

```
Agent1 =
  say_hello(agent1, agent2):lambda |
  say_hello(agent1, agent3):lambda,
Agent2 =
  say_hello(agent2, agent1):lambda |
  say_hello(agent2, agent3):lambda,
Agent3 =
  say_hello(agent3, agent1):lambda |
  say_hello(agent3, agent2):lambda,
Pr = Agent1|(Agent2|Agent3).
```

is extremely simple but, if a new agent `agent4` joins the protocol at runtime, there is no way to dynamically adapt it in such a way that `agent1`, `agent2`, and `agent3` can communicate with it. Instead, using template trace expressions

```
AgentI =
  finite_composition(
    |,
    say_hello(var(1), var(2)):lambda,
```

```
    [m(var(2), [remove(var(1))])]
  ),
Pr = finite_composition(|, (AgentI|(reset(var(1)):lambda)), [m(var(1), [])]).
```

we can represent the protocol in a more compact way and, above all, we can instantiate the protocol at runtime during the protocol switch phase: it is enough that `agent4` requires a protocol switch (in this case, the protocol switch has as interaction type the `reset` type[3]) to all other agents passing as *Parameters* the set {`agent1, agent2, agent3, agent4`}.

6.3 Implementation

6.3.1 Prolog

In this section we introduce all Prolog predicates which implement the building blocks of our framework. All the features offered by the various components of the framework are written using the Prolog language:

- the *preprocessing function*, which instantiates the template trace expressions, is implemented by the `apply` predicate;
- the *projection function*, which given a global protocol returns a local version for each protocol-driven agent, is implemented by the `project` predicate;
- the *next* transition function, which allows moving from a state of the protocol to another, is implemented by the `next` predicate;
- the *generate* function, which returns all possible messages that are allowed in the current state of the protocol, is implemented by the `inMsgs` and `outMsgs` predicates (thus separating the messages which can be received from those which can be sent).

6.3.1.1 The *event types*

From an implementation point of view, an event type is defined by means of the *has_type/2* predicate whose first argument is the event, and the second argument is the type. For example,

[3]However, for the purpose of this example, it is not necessary to go into details.

$collect\_parcel(agent2, parcel3,$
$parcelweight(4, kg))$
$\in collect\_parcel.$

is implemented by means of the Prolog fact

```
has_type(
   collect_parcel(agent2, parcel3,
   parcelweight(4, kg)),
   collect_parcel).
```

More in general, if we want to state that any *collect_parcel/3* event has *collect_parcel* type, no matter which are its three arguments, we will use anonymous logical variables instead of the three ground arguments:

```
has_type(collect_parcel(_, _, _), collect_parcel).
```

In the rest of the chapter, we will use frequently the event types (*EventType*) and interaction types (*IntType*) terminologies interchangeably. This follows from the fact that we are considering only communicative event types, as already highlighted many times in this work.

6.3.1.2 The *empty* function

The auxiliary function $\epsilon(\_)$, which was presented in Chapter 4 and it was defined by the rules in Figure 4.2, implemented by the *empty/1* Prolog predicate, specifies the trace expressions whose interpretation contains the empty sequence.

```
empty(lambda) :- !.
empty(T1\/T2) :- (empty(T1), !; empty(T2)).
empty(T1|T2)  :- !, empty(T1), empty(T2).
empty(T1*T2)  :- !, empty(T1), empty(T2).
empty(T1/\T2) :- !, empty(T1), empty(T2).
empty(_>>T)   :- !, empty(T).
```

The `!` is a Prolog goal which always succeeds, but cannot be backtracked past; it is best used to prevent unwanted backtracking. Intuitively, a trace expression τ s.t. $\epsilon(\tau)$ holds specifies a protocol that is allowed to successfully terminate.

6.3.1.3 The *next* transition function

The *next* predicate implements the homonym function defining the semantics of trace expressions.

Basic cases.

Prefix. If the trace expression is a sequence (: operator) consisting of the event type ET followed by the trace expression T (first argument), and the actual event perceived in the environment is E (second argument), and E has type ET, then $ET{:}T$ can move to T (third argument).

```
next(ET:T,E,T) :- has_type(E,ET).
```

Concatenation. If *T1* can move to *T3* upon observing the event E, then the concatenation *T1*T2* can move to the concatenation *T3*T2* upon observing the event E; if *T1* contains ϵ, that is, *empty(T1)* holds (second rule), then *T1*T2* can move to *T3* if *T2* can move to *T3*.

```
next(T1*T2,E,T3*T2) :-
   next(T1,E,T3).
next(T1*T2,E,T3) :-
   !,empty(T1), next(T2,E,T3).
```

Intersection. If *T1* can move to *T3* upon observing the event E and *T2* can move to *T4* upon observing **the same** event E, then the intersection ($\wedge$ operator) between *T1* and *T2* can move to the intersection between *T3* and *T4*, $T3 \wedge T4$.

```
next(T1/\T2,E,T3/\T4) :-
   next(T1,E,T3), next(T2,E,T4).
```

Union. If *T1* can move to *T2* upon observing the event E, then the union ($\vee$ operator) between *T1* and any other trace expression, $T1 \vee \_$, can move to *T2* upon observing the event E (and the converse for the second rule).

```
next(T1\/_,E,T2) :- next(T1,E,T2).
next(_\/T1,E,T2) :- !, next(T1,E,T2).
```

Shuffle. If *T1* can move to *T3* upon observing the event *E*, then the shuffle (| operator) between *T1* and *T2*, *T1*|*T2*, can move to *T3*|*T2* upon observing the event *E* (and the converse for the second rule).

```
next(T1|T2,E,T3|T2) :- next(T1,E,T3).
next(T1|T2,E,T1|T3) :- next(T2,E,T3).
```

Filter. If the observed event *E* has type *ET* and *T1* can move to *T2* upon observing the event *E*, then the filter (≫ operator) can move to *T2* filtered with the same event type *ET*; otherwise, the filter can move to itself.

```
next(ET>>T1,E,ET>>T2) :-
   event(E),
   (has_type(E,ET) *-> next(T1,E,T2); T2=T1).
```

Where `Cond -> T ; F` represents implication, `T` is executed only if the `Cond` is true, otherwise (`;`) `F` is executed. The `*->` operator has the same behaviour with in addition the possibility of backtracking in the left side.

6.3.1.4 The *generate* function

As will be known in Sections 6.3.2.2 and 6.3.3.2, during the interpreter execution, after that an agent calls the `next` predicate (upon receiving or sending a message), it saves the new state obtained by the `next` predicate as the new `current_state` (using the agent's name as the key). This is possible thanks to the `recorda` predicate, which saves, given a key, the predicate passed as argument; this predicate can be recovered using the `recorded` predicate, using the same key (in this case the agent's name).

The generate function, as we have already seen in Section 6.2.2, is implemented by two different predicates:

the `inMsgs` predicate,

```
inMsgs(MyName, ListToReceive) :-
  recorded(MyName, current_state(LastState), _),
  findall(msg(Sender, MyName, Performative, Content, NewState),
  next(LastState, msg(Sender, MyName, Performative, Content),
  NewState), ListToReceive).
```

which, given the name of a protocol-driven agent, returns a list containing all messages allowed in the current state of the protocol, where the agent is the receiver;

the `outMsgs` predicate,

```
outMsgs(MyName, ListToSend) :-
  recorded(MyName, current_state(LastState), _),
  findall(msg(MyName, Receiver, Performative, Content),
  next(LastState, msg(MyName, Receiver, Performative, Content),
  NewState), ListToSend).
```

which, given the name of a protocol-driven agent, returns a list containing all messages allowed in the current state of the protocol, where the agent is the sender.

In both predicates the search of all possible messages, which are consistent with the current state of the protocol, is done using the `findall` predicate.

The `findall(Object,Goal,List)` produces a list `List` of all the objects `Object` that satisfy the goal `Goal`. Often `Object` is a variable, in which case the query can be read as: Give me a list containing all the instantiations of `Object` which satisfy `Goal`.

6.3.1.5 The *project* function

The projection algorithm has been designed and implemented by Ancona et al. in [4] for the constrained global types formalism; this algorithm has been adapted in this thesis for the trace expressions formalism.

The projection algorithm implementation w.r.t the *empty* and *next* algorithm implementation must deal with the problem of cyclic terms, indeed, one of the reasons for the choice of SWI Prolog is that it manages infinite terms in an efficient way. Since we need to record the association between any type and its projection in order to correctly detect and manage cycles, we exploited the SWI Prolog library `assoc` for association lists, `http://www.swi-prolog.org/pldoc/man?section=assoc`. The three predicates of the library `assoc` that we use for our implementation are

- `empty_assoc`(*-Assoc*): *Assoc* is unified with an empty association list.
- `get_assoc`(*+Key, +Assoc, ?Value)*: *Value* is the value associated with *Key* in the association list *Assoc*.

- `put_assoc`(*+Key, +Assoc, +Value, ?NewAssoc*): *NewAssoc* is an association list identical to *Assoc* except that *Key* is associated with *Value*. This can be used to insert and change associations.

The projection is implemented by a predicate `project(T, ProjAgs, ProjT)` where `T` is the trace expression to be projected, `ProjT` is the result, and `ProjAgs` is the set of agents onto which the projection is performed. The algorithm exploits the predicate `involves(IntType, ProjAgs)` succeeding if `IntType` may involve one agent, as a sender or a receiver, in `ProjAgs`.

Currently `involves` looks for actual interactions `ActInt` whose type is `IntType` and assumes that senders and receivers in `ActInt` are ground terms, but it could be extended to take agents' roles into account or in other more complex ways. It uses the "or" Prolog operator `;` and the `member` predicate offered by the library `lists`. It exploits the predicate `has_type(ActInt, IntType)` implementing the definition of the type `IntType` of an actual interaction `ActInt`.

```
involves(IntType, List) :-
    has_type(msg(Sender, Receiver, _, _), IntType),
    (member(Sender, List);member(Receiver, List)).
```

For the implementation of `project/3` we use an auxiliary predicate `project/6` with the following three additional arguments:

- an initially empty association `A` to keep track of cycles;
- the current depth of the trace expression under projection, initially set to `0`;
- the depth of the deepest sequence operator belonging to the projected type, initially set to `-1`.

```
project(T, ProjAgs, ProjT) :-
    empty_assoc(A), project(A, 0, -1, T, ProjAgs, ProjT).
```

The predicate is defined by cases.

1. `lambda` is projected into `lambda`.

```
project(_Assoc, _Depth, _DeepestSeq, lambda, _ProjAgs, lambda):- !.
```

2. If `Type` has been already met while projecting the trace expression (`get_assoc(Type, Assoc, (AssocProjType,LoopDepth))` succeeds), then its projection `ProjT` is `AssocProjType` if `LoopDepth =< DeepestSeq` and is `lambda` otherwise. The "if-then-else" construct is implemented in Prolog as `Condition -> ThenBranch ; ElseBranch`.

```
project(Assoc, _Depth, DeepestSeq, Type, _ProjAgs, ProjT) :-
    get_assoc(Type,Assoc,(AssocProjType,LoopDepth)),!,
    (LoopDepth =< DeepestSeq -> ProjT=AssocProjType; ProjT=lambda).
```

3. `T = (IntType : T1)`. `ProjT` is recorded in the association `A` along with the current depth `Depth` (`put_assoc((IntType:T1),Assoc,(ProjT,Depth),NewAssoc)`). If `IntType` involves `ProjAgs`, `ProjT=(IntType:ProjT1)` where `ProjT1` is obtained by projecting `T1` onto `ProjAgs`, with association `NewAssoc`, depth of the type under projection increased by one, and depth of the deepest sequence operator equal to `Depth`. If `IntType` does not involve `ProjAgs`, then the projection on `T` is the same as `T1` with association `NewAssoc`, depth of the type under projection equal to `Depth`, and depth of the deepest sequence operator equal to `DeepestSeq`.

```
project(Assoc, Depth, DeepestSeq, (IntType:T1), ProjAgs, ProjT) :- !,
    put_assoc((IntType:T1),Assoc,(ProjT,Depth),NewAssoc),
    (involves(IntType, ProjAgs) ->
    IncDepth is Depth+1,
    project(NewAssoc,IncDepth,Depth,T1,ProjAgs,ProjT1),
    ProjT=(IntType:ProjT1);
    project(NewAssoc,Depth,DeepestSeq,T1,ProjAgs,ProjT)).
```

4. `T = (IntType >> T1)`. `ProjT` is recorded in the association `A` along with the current depth `Depth` (`put_assoc((IntType>>T1),Assoc,(ProjT,Depth),NewAssoc)`). Considering that the filter operator concerns all events which belong to `IntType` and also those that do not belong to `IntType`, `ProjT=(IntType>>ProjT1)` where `ProjT1` is obtained by projecting `T1` onto `ProjAgs`, with association `NewAssoc`, depth of the type under projection increased by one, and depth of the deepest sequence operator equal to `Depth`.

```
project(Assoc, Depth, DeepestSeq, (IntType>>T1), ProjAgs, ProjT) :- !,
```

```
put_assoc((IntType>>T1),Assoc,(ProjT,Depth),NewAssoc),
IncDepth is Depth+1,
project(NewAssoc,IncDepth,Depth,T1,ProjAgs,ProjT1),
ProjT=(IntType:ProjT1).
```

5. `T = T1 op T2`, where `op` $\in \{\backslash/, |, *, /\backslash\}$: the association between `T1 op T2` and the projected type `ProjT` is recorded in the association `Assoc` along with the current depth `Depth`, `T1` and `T2` are projected into `ProjT1` and `ProjT2` respectively, with association equal to `NewAssoc`, depth of the type under projection increased by one and depth of the deepest sequence operator equal to `DeepestSeq`. The result of the projection is `ProjT=(ProjT1 op ProjT2)`. For example, if `op` is `*`, the Prolog clause is:

```
project(Assoc, Depth, DeepestSeq, (T1*T2), ProjAgs, ProjT) :- !,
    put_assoc((T1*T2),Assoc,(ProjT,Depth),NewAssoc),
    IncDepth is Depth+1,
    project(NewAssoc, IncDepth, DeepestSeq, T1, ProjAgs, ProjT1),
    project(NewAssoc, IncDepth, DeepestSeq, T2, ProjAgs, ProjT2),
    ProjT=(ProjT1*ProjT2).
```

6.3.1.6 The *apply* function

As already anticipated in Section 4.5, we need a function (to be used before of the projection phase) which instantiates the protocol removing all parameters and returning a "ground"[4] trace expression.

The *apply* function is similar to the *project* function, it must manage the problem of cyclic terms. The *apply* function is implemented by a predicate `apply(T, Params, InsT)` where `T` is the template trace expression to be instantiated, `Params` are the values to be assigned to the parameters inside the template trace expression, and `InsT` is the result, that is the instantiated trace expression. This trace expression is exactly the one that will be passed as argument to the *project* function.

For the implementation of `apply/3` we use an auxiliary predicate `apply/7` with the following three additional arguments (the fourth arguments is a copy of parameters

[4]A trace expression is ground when it has not parameters inside it, that is when it is fully instantiated.

used as backup of global parameters in order to manage nested `finite_composition` operators):

- an initially empty association `A` to keep track of cycles;
- the current depth of the trace expression under instantiation, initially set to `0`;
- the depth of the deepest sequence operator belonging to the instantiated type, initially set to `-1`.

```
apply(T, Params, InsT) :-
    empty_assoc(A), apply(A, 0, -1, T, InsT, Params, Params).
```

The predicate is defined by cases.

1. `lambda` is instantiated in `lambda`.

   ```
   apply(_, _, _, lambda, lambda, _CurPar, _GlobalPar):- !.
   ```

2. If `Type` has been already met while instantiating the trace expression (`get_assoc(Type, Assoc, (AssocInsType,LoopDepth))` succeeds), then its instantiation `InsT` is `AssocInsType` if `LoopDepth =< DeepestSeq` and is `lambda` otherwise.

   ```
   apply(Assoc, _Depth, DeepestSeq, Type, InsT, CurPar, GlobalPar) :-
      get_assoc(Type,Assoc,(AssocInsType,LoopDepth)),!,
      (LoopDepth =< DeepestSeq -> InsT=AssocInsType; InsT=lambda).
   ```

3. `T = (IntType op T1)`, where `op` $\in \{$`:`, `>>`$\}$. `InsT` is recorded in the association `A` along with the current depth `Depth` (`put_assoc((IntType:T1),Assoc,(InsT, Depth),NewAssoc)`). If `IntType` contains parameters (`var(i)`), these are substituted by the `apply_inside` predicate with the current correct values obtaining `IntType1`. `InsT = (IntType1:InsT1)` where `InsT1` is obtained by instantiating `T1`, with association `NewAssoc`, depth of the type under instantiation increased by one, and depth of the deepest sequence operator equal to `Depth`.

   ```
   apply(Assoc, Depth, DeepestSeq, (IntType:T1), InsT, CurPar, GlobalPar) :-
      !, put_assoc((IntType:T1),Assoc,(InsT,Depth),NewAssoc),
      IncDepth is Depth+1,
   ```

```
    apply_inside(IntType, IntType1, CurPar),
    apply(NewAssoc,IncDepth,Depth,T1,InsT1,CurPar,GlobalPar),
    InsT=(IntType1:InsT1).

/* if the term is a parameter */
apply_inside(T, InsT, CurPar) :-
    functor(T, var, _), !, /* the term's functor is var */
    arg(1, T, N), /* get the identification number of the parameter */
    /* get the current value associated with this parameter */
    member(t(var(N), Value), CurPar),
    InsT = Value.  /* replace the parameter with the value recovered */
/* if the term is not a parameter but it could contain some parameters */
apply_inside(T, InsT, CurPar) :-
    functor(T, F, A), A > 0, !, /* it is a compound term */
/* propagate the search of parameters within subterms */
    findall(
        InsTerm,
        (arg(_, T, Term), apply_inside(Term, InsTerm, CurPar)),
        InsTerms),
    /* rebuild the term with the same functor and arity (each argument is a
free variable) */
    functor(InsT, F, A),
    /* unify each free variable inside the new term with the corresponding
subterm instantiated */
    create_term(InsT, InsTerms, 1).
/* the term is an atom (a term with arity 0, consequently there is nothing
to replace) */
apply_inside(T, T, _CurPar).

/* given a compound term with arity N and a list of length N, unify each argument
of the term with the corresponding element of the list */
create_term(_, [], _).
create_term(Term, [H|T], N) :-
    arg(N, Term, H),
    N1 is N+1,
    create_term(Term, T, N1).
```

The `functor` predicate allows us to extract the functor from the term passed as first argument; whereas the `arg` predicate allows us to select/unify a single argument in a term. These two predicates are used to manipulate terms, in this case, the `apply_inside` predicate executes them in order to extract information from the `IntType` interaction event type. It is necessary to find and replace all `var(N)` terms (thus identifying the functor `var`, using the `functor` predicate, and the argument `N`, using the `arg` predicate) with the current value selected inside the current lap of the `finite_composition` operator.

4. `T = T1 op T2`, where `op` $\in$ {`\/`, `|`, `*`, `/\`}: the association between `T1 op T2` and the instantiated type `InsT` is recorded in the association `Assoc` along with the current depth `Depth`, `T1` and `T2` are instantiated in `InsT1` and `InsT2` respectively, with association equal to `NewAssoc`, depth of the type under instantiation increased by one and depth of the deepest sequence operator equal to `DeepestSeq`. The result of the instantiation is `InsT=(InsT1 op InsT2)`. For example, if `op` is `*`, the Prolog clause is:

```
apply(Assoc, Depth, DeepestSeq, (T1*T2), InsT, CurPar, GlobalPar) :-
   !, put_assoc((T1*T2),Assoc,(InsT,Depth),NewAssoc),
   IncDepth is Depth+1,
   apply(NewAssoc, IncDepth, DeepestSeq, T1, InsT1, CurPar, GlobalPar),
   apply(NewAssoc, IncDepth, DeepestSeq, T2, InsT2, CurPar, GlobalPar),
   InsT=(InsT1*InsT2).
```

5. `T = finite_composition(Op, T1, Vars)`, where `Op` $\in$ {`\/`, `|`, `*`, `/\`}, `T1` is the trace expression to be replicated and `Vars` is the set of parameters on which iterates the operator. `InsT` is obtained by calling an auxiliary predicate `finite_composition/9` which instantiates T1 as many times as the number of values of the first parameter multiplied by the number of values of the second and so on[5].

```
apply(Assoc, Depth, DeepestSeq, finite_composition(Op, T1, Vars), InsT,
CurPar, GlobalPar) :- !,
   finite_composition(Op, T1, CurParameters, InsT, Assoc, Depth,
```

[5]If we have the set of parameters {`var(1)`, `var(2)`, `var(3)`} where `var(1)` iterates on {`v1`, `v2`, `v3`}, `var(2)` iterates on {`v1`, `v3`} and `var(3)` iterates on {`v2`, `v4`, `v5`}; the trace expression `finite_composition(|, T, [m(var(1),[]), m(var(2),[]), m(var(3),[])])`, when instantiated by the `apply` predicate, becomes a new trace expression composed by a shuffle operator with `n` branches, where `n = num_of_par(var(1)) * num_of_par(var(2)) * num_of_par(var(3)) = 3 * 2 * 3 = 18`.

```
    DeepestSeq, GlobalPar, Vars).
```

The `finite_composition` predicate is defined as follow.

```
finite_composition(Op, T, CurPar, InsT, Assoc, Depth, DeepestSeq,
GlobalPar, Vars) :-
    select_subset_vars(GlobalPar, Vars, CurPar, SubSetGlobalPar, NewVars),
    findall(
        IT,
        (select_one_assoc(SubSetGlobalPar, SinglePathPar, NewVars),
        apply(Assoc, Depth, DeepestSeq, T, IT, SinglePathPar, GlobalPar)),
        InsTs),
    add_operator(Op, InsTs, InsT).
```

It selects the subset of parameters which appears inside the set of modifiers `Vars`; subsequently, it instantiates the trace expression for each parameter and for each possible value which can take the parameter.

In order to select only the subset of variables currently used in the `finite_composition` operator, we use the `select_subset_vars` predicate,

```
/* if there is no parameters to select */
select_subset_vars([], V, _, [], V).
/* if the parameter is present in the current set of modifiers on which the
finite_composition operator iterates, thus the predicate must only continue
the selection of the other parameters */
select_subset_vars([t(Var,Values)|OtherPar], Vars, CurPar,
[t(Var, Values)|T], NewVars) :-
    /* the modifier belongs to the set of parameters */
    member(m(Var, _), Vars), !,
    /* continue the selection of the other parameters */
    select_subset_vars(OtherPar, Vars, CurPar, T, NewVars).
/* if the parameter is not present in the current set of modifiers but the
corresponding parameter has a fixed value */
select_subset_vars([t(Var,Values1)|OtherPar], Vars, CurPar,
[t(Var, Values)|T], NewVars) :-
    /* select the correct current value for the parameter */
    (member(t(Var, Values2), CurPar), not(is_list(Values2)),
```

```
    Values = Values2); (not(member(t(Var, _), CurPar)),
    not(is_list(Values1)), Values = Values1),
    /* continue the selection of the other parameters */
    !, select_subset_vars(OtherPar, Vars, CurPar, T, NewVars1),
    /* set the finite_coposition modifiers with the corresponding values */
    set_all_modifiers(Var, Values, NewVars1, NewVars).
 /* the parameter is discarded because is not used in the finite_composition
operator */
 select_subset_vars([_|OtherPar], Vars, CurPar, T, NewVars) :-
    select_subset_vars(OtherPar, Vars, CurPar, T, NewVars).
```

which selects the subset of parameters on which the `finite_composition` predicate must iterate.

The `set_all_modifiers` predicate is necessary to replace into each modifier all parameters with the current values associated. For example, if in the current state of the instantiation phase the parameters `var(1)`, `var(2)`, `var(3)` take the values `v1`, `v2`, `v3` respectively, the `set_all_modifiers` predicate applied to `m(add(var(1)))`, `m(remove(var(2)))` and `m(add(var(3)))` modifiers returns `m(add(v1))`, `m(remove(v2))` and `m(add(v3))`.

```
 /* there is no modifiers to set */
 set_all_modifiers(_, _, [], []).
 /* replace into each modifier all parameters with the current values associated
*/
 set_all_modifiers(Variable, Value, [m(Variable1,Modifiers)|Variables],
 [m(Variable1,Modifiers1)|T]) :-
    /* replace inside the single modifier */
    set_modifiers(Variable, Value, Modifiers, Modifiers1),
    /* continue the replacement */
    set_all_modifiers(Variable, Value, Variables, T).

 /* the modifier does not contain parameters to set */
 set_modifiers(_, _, [], []).
 /* add(Variable) -> add(Value) */
 set_modifiers(Variable, Value, [add(Variable)|Modifiers],
 [add(Value)|T]) :-
```

```
    !, set_modifiers(Variable, Value, Modifiers, T).
/* remove(Variable) -> remove(Value) */
set_modifiers(Variable, Value, [remove(Variable)|Modifiers],
[remove(Value)|T]) :-
    !, set_modifiers(Variable, Value, Modifiers, T).
/* add(Variable) modifier remains unchanged */
set_modifiers(Variable, Value, [add(Variable1)|Modifiers],
[add(Variable1)|T]) :-
    set_modifiers(Variable, Value, Modifiers, T).
/* remove(Variable) modifier remains unchanged */
set_modifiers(Variable, Value, [remove(Variable1)|Modifiers],
[remove(Variable1)|T]) :-
    set_modifiers(Variable, Value, Modifiers, T).
```

In this way, we obtain a subset of the variables containing only those which belong to our trace expression and, for each modifier, we have the corresponding current value.

Each iteration (`findall` predicate in `finite_composition` predicate) requires that each parameter is set to a value; so, we can instantiate all different branches of the `finite_composition` operator. The predicate which solves this problem is the `select_one_assoc` predicate.

```
/* there is no parameter to set with a value */
select_one_assoc([], [], _).
/* if the parameter has multiple possible values (it is a parameter on which
the finite_composition operator iterates) select one of them (considering all
the modifiers) */
select_one_assoc([t(var(N), Values)|OtherParameters],
[t(var(N), H)|T], Variables) :-
    is_list(Values),
    ((member(m(var(N), Modifiers), Variables)) ->
        ((member(H, Values); member(add(H), Modifiers)),
        not(member(remove(H), Modifiers))); true),
    /* if the parameter has only one possible value, select it */
    select_one_assoc(OtherParameters, T, Variables).
select_one_assoc([t(var(N), Value)|OtherParameters],
```

```
[t(var(N), Value)|T], Variables) :-
    not(is_list(Value)),
    select_one_assoc(OtherParameters, T, Variables).
```

Finally, the `finite_composition` predicate must call the `add_operator` predicate. It must only compose all branches previously created during the `findall` phase, with the chosen operator `Op` (that is specified as argument of `finite_composition` operator).

```
add_operator(_, [], lambda).
add_operator(_, [InsT|[]], InsT) :- !.
add_operator(Op, [InsT|T], InsTWithOp) :-
    add_operator(Op, T, InsTWithOp1),
    add_operator_aux(Op, InsT, InsTWithOp1, InsTWithOp).
```

The `add_operator_aux` is the predicate which composes two trace expressions: `add_operator_aux(Op, T1, T2, Res)` returns `Res` unified with `T1 Op T2`.

6.3.2 Jason

Jason, as anticipated in Section 1.2, is an interpreter for an extended version of the AgentSpeak language.

The first attempt to implement our framework has been using Jason because it supports natively the Prolog language; in this way, there is no integration problem[6] between the agent interpreter implementation and the Prolog predicates (that have just been presented in the previous section).

6.3.2.1 Integration with the framework

As already highlighted in Section 6.3.1, the most important features of our framework, such that the protocol representation, the protocol instantiation and the protocol projection are implemented using the Prolog language. When we implement our framework using a chosen MAS tool, we must only implement the interpreter using the language provided by the tool, everything else, as just said, is implemented instead in Prolog.

[6] Mainly for this reason we decided to present it first.

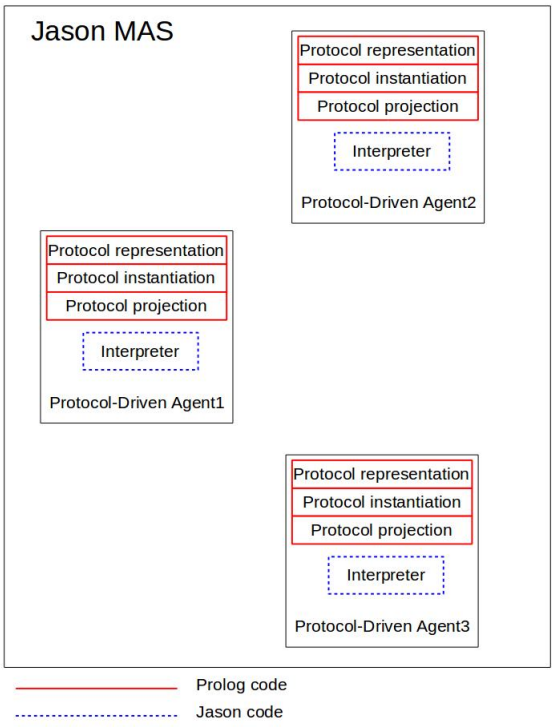


FIGURE 6.1: The implementation of our framework in Jason.

In general, an integration is needed between the two languages, that is the communication between the protocol-driven agent's interpreter and Prolog; however, in the case of Jason, this integration is not necessary because it supports Prolog code inside it. In fact, as already explained in Section 1.2, Jason allows us to write rules that correspond to Prolog predicates, in this way, we can copy the `next`, `apply`, `project`, etc... predicates in Jason, thus calling them directly from the interpreter.

Consequently, the protocol-driven agent's interpreter is the only one which must be implemented using Jason language (plans, goals, annotations and so on), considering that it represents the body of the protocol-driven agent.

Below we present the customization for the Jason tool.

6.3.2.2 The protocol-driven agent's interpreter customization

First of all, each Jason protocol-driven agent must achieve the `!execute` goal[7]. This goal takes one argument, that is the name of the protocol (in this case `pr1`) which the agent wants to follow.

[7]The goals have been already introduced in Section 1.2.

```
/* Initial beliefs */
myId(0). /* agent's id */
prec(send). /* priority to send (prec(rec) priority to receive)*/
timeOut(10). /* it is expressed in seconds */
/* list of all agents which have the power to require
   a protocol switch to the other agents */
empowered([switcherAgent]).
/* Initial goals */
!start.
/* Plans */
+!start : true <- !execute(trace_expression(pr1)).
```

The four beliefs[8] derive directly from the first four requirements expressed in Section 6.1.

Each agent starts with only one goal to achieve: the `!start` goal. The `!start` goal is achieved by the `+!start` plan which tries to achieve the `!execute` goal.

The `+!execute` plan instantiates the protocol, projects the protocol and, after that, runs the interpreter.

```
+!execute(trace_expression(Name), ActualParameters):
  .my_name(MyName) & /* get the agent's name */
  trace_expr(Name, T) & /* recover the trace expression (protocol) */
  apply(T, ActualParameters, InsT) & /* instantiate the protocol */
  project(InsT, [MyName], ProjT) & /* project the protocol */
  <-
  !move_to_state(ProjT); /* save the current state of the protocol */
  !startInterpreter(trace_expression(Name)). /* run the interpreter */
```

The interpreter uses some predicates in order to manage the message queue and the protocol switch requests.

```
/* FIFO queue of messages received by the agent */
messageQueue([]). /* initially empty */
/* predicate which adds a message as last element of the message queue */
push([], Msg, [Msg]).
```

[8]The values reported for the four beliefs are for example, indeed, they depend on the specific agent and the specific MAS context.

```
push([H|T], Msg, Msgs) :-
  push(T, Msg, Msgs1) & Msgs = [H|Msgs1].
/* predicate which removes the message at the top of the message queue */
pop([H|T], H, T).
/* predicate which gets the message at the top of the message queue */
top([H|T], H).

/* at the beginning there is no protocol switch requests */
switchMsg(null).

/* when an agent receives a message, this plan adds the
   message as the last of the message queue */
+!kqml_received(Sender, Performative, Content, _MsgId) :
  messageQueue(Msgs) & /* Get Message Queue */
  .print("Receive: (", Sender, ",", Performative, ",", Content, ")") &
  /* push the message received into the message queue */
  push(Msgs, msg(Sender, MyName, Performative, Content), Msgs1)
  <-
  -messageQueue(_); +messageQueue(Msgs1). /* update the message queue */
```

The interpreter implementation in Jason follows the high level design introduced in Section 6.2.2; consequently, it implements the self-adaptive protocol-driven agent's interpreter, case by case. In particular, in Jason, each case is managed by a single plan, as shown in Appendix B.

As an example, we show only the plan which manages a protocol switch message reception.

```
/* if a protocol switch is required and it is consistent with
   the protocol */
+!executeInterpreter :
  /* in a previous run a protocol switch was required */
  switchMsg(msg(S, MyName, Performative, switch(PrSwitch, Parameters))) &
  current_state(LastState) &
  next(LastState, msg(S, MyName, Performative,
       switch(PrSwitch, Parameters)), NewState)
  <- /* Agent can manage protocol switch now */
  .print("SwitchMsg: ", msg(S, MyName, Performative,
         switch(PrSwitch, Parameters)));
```

```
/* Remove protocol switch request from the agent's knowledge base */
-switchMsg(_); +switchMsg(null);
/* If the agent needs of some operations before it can switch to the
   new protocol (each agent must have its implementation) */
!cleanUp; /* In Jason it is implemented as a goal */
/* Start the new protocol */
!execute(trace_expression(PrSwitch), Parameters).
```

Considering that (see Section 1.2), the plans are defined as:

```
+!triggering_event : context <- body.
```

we comment on each part of the plan shown above:

- the `triggering_event` is the `executeInterpreter`, that is the specific event for which the plan is to be used; in this way, if an event, which was generated by the addition of a `!executeInterpreter` goal, took place matching the triggering event, the plan starts its execution.
- the `context`, which is used for checking the current situation so as to determine whether the plan is likely to succeed in handling the event (e.g. achieving a goal), is divided so:
  1. it is checked if in a previous run a protocol switch was required, this is done using the `switchMsg` belief, which is set when a protocol switch message is received (this is managed by another plan of the interpreter, see Appendix B);
  2. it is checked if the protocol switch is consistent with the current state of the protocol, this is done using the `next` predicate which implements the *next* transition function (considering that we are in Jason, we can use directly the `next` predicate).
- the `body`, which is a sequence of formula determining a course of action, is divided so:
  1. it is printed a message informing that is occurring a protocol switch;
  2. it is updated the `switchMsg` belief (it is removed the old belief `-switchMsg` and it is added the new belief `+switchMsg`);
  3. it is added the goal `!cleanUp`, which is the implementation of the homonym policy;

4. it is called the `!execute` goal, which is the main goal that we have already presented before in the section, in this way, it will start the new protocol passed as argument in the protocol switch message.

Policies implementation. Each agent, which executes the interpreter, must implement all the policies described in Section 6.2.1. In this way, the agent can:

- react to a message reception, for example:

```
+!react(Msg) :
  true
  <-
  /* save the number of messages received */
  -count_messages_received(C);
  +count_messages_received(C+1).
```

- select a message to send, for example:

```
/* there is no message to send */
select([], _, cannot_send, cannot_send, _NewState).
/* choose the first message to send */
select(
[msg(MyName, Receiver, Performative, Content, NewState)|_],
X, can_send, msg(MyName, Receiver, Performative, Content), NewState).
```

- manage an unexpected message, for example:

```
+!unexpected(Msg):
  true
  <-
  /* save the error in order to manage it after */
  +error(Msg).
```

- clean the knowledge base after a protocol switch request, for example:

```
+!cleanUp :
  true
  <-
  /* update the current state */
  -current_state(_);
  +current_state(initial).
```

6.3.2.3 Experiments with Jason

In this section we report some experiments of our work in order to show some examples of results which we can obtain executing our protocol-driven agents inside Jason. All screen shots, which we have reported, show the agent's gui in Jason.

The experiments that we have chosen to report are those related to the examples presented in Chapter 5; in this way, we can focus directly on the main important aspects of the protocols, considering that we have already presented all logical details.

Iterated Contract Net Protocol. In Figure 6.2, we have reported the body of the `initiator` agent. As already said in Section 6.3.2.2, it consists in the call of the main goal `!start` which calls in turn the `!execute` goal passing as argument the name of the protocol and all parameters necessary for the instantiation phase.
The implementation of the participant agent (see Figure 6.3) is very similar.

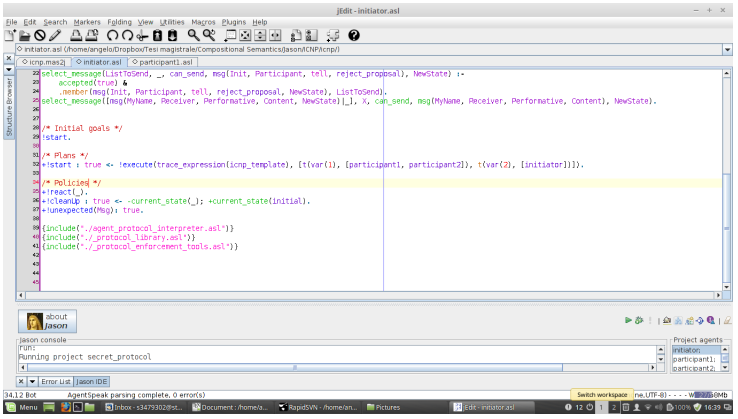


FIGURE 6.2: ICNP - initiator agent in Jason.

After the protocol has been instantiated, it starts the communication among all agents

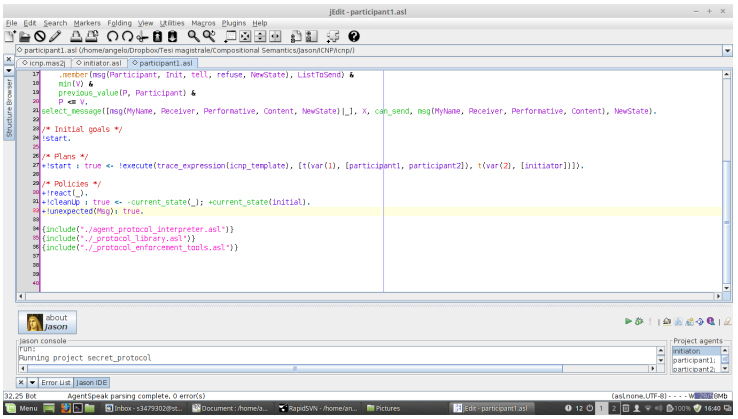


FIGURE 6.3: ICNP - participant agent in Jason.

inside the system; in particular, in Figure 6.4 we can observe the information exchange between `initiator` and `participant2`, where `participant2` proposes the value `10` and `initiator` proposes against the value `9`.

At a certain point of the protocol, in Figure 6.5, `participant2` proposes the value `6`

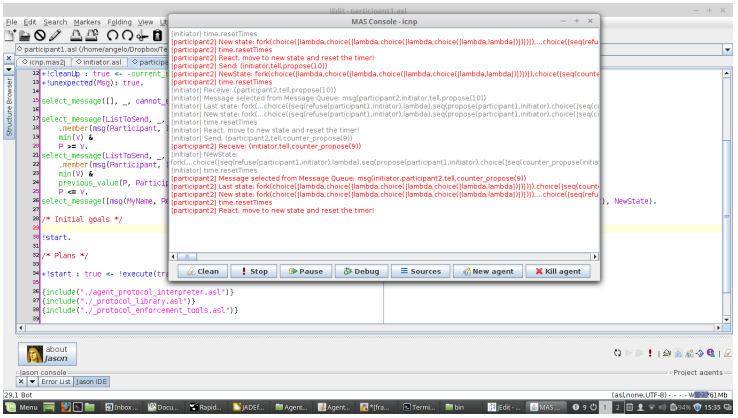


FIGURE 6.4: ICNP - Proposal and counterproposal.

and `initiator` accepts sending the `accept_proposal` message.

The winner agent, in this case `participant2` (Figure 6.6), sends to `initiator` the

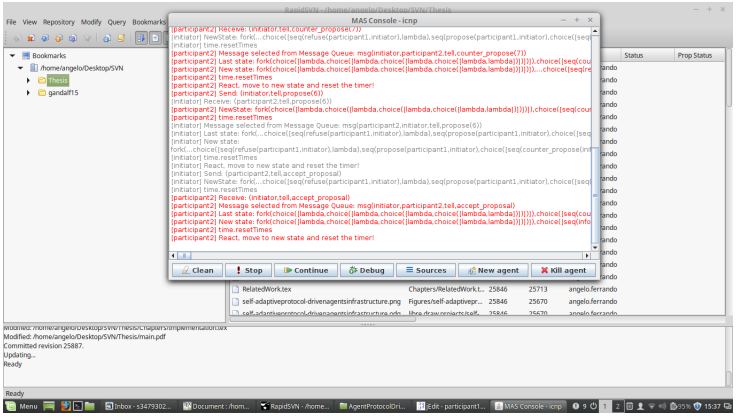


FIGURE 6.5: ICNP - Acceptance and rejection of the proposal.

`inform` message (as expected from the protocol) thus ending correctly the protocol.

Auction Protocol. From the point of view of the trace expression structure, the Auction protocol is very interesting, in fact, it is the only example which uses the `intersection` operator.

The output reported in Figure 6.7 shows `agent1` which sends the `inform_start` and the `cfp_1` messages to `agent2` and `agent3`. Consequently, `agent2` and `agent3` send a proposal to `agent1`. After, `agent1`, accepts `agent2` and rejects `agent3` (in this particular case). All these phases are synchronized, in fact, each phase can start only when the

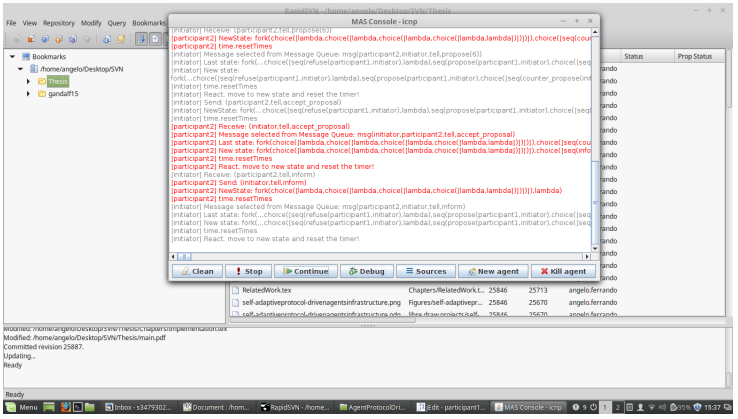


FIGURE 6.6: ICNP - Inform message from the winner agent.

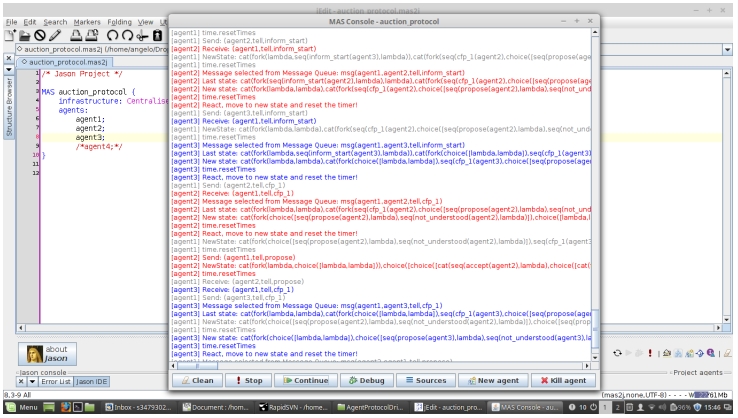


FIGURE 6.7: Auction Protocol - Inform start and call for proposal.

previous one is completed. This is obtained by the presence of the *intersection* operator, as we have seen in Chapter 5.

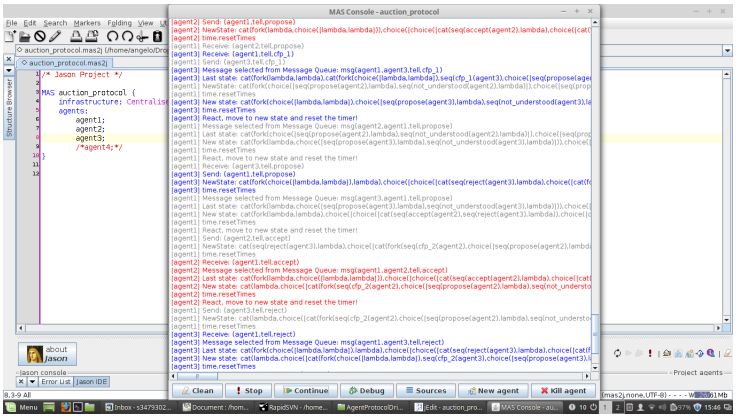


FIGURE 6.8: Auction Protocol - Proposal with acceptance of one participant and rejection of the others.

Secret Protocol. The secret protocol is characterized by two types of agents, the normal (which can become secret) agent, reported in Figure 6.9, and the `boss` agent, reported in Figure 6.10. Initially all agents can communicate among them without

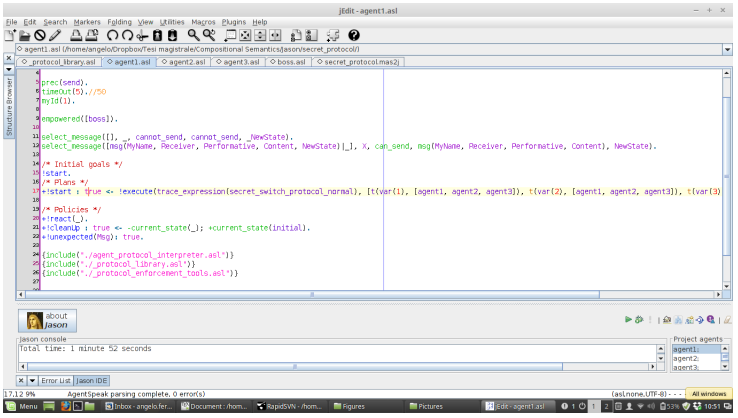


FIGURE 6.9: Secret Protocol - Agent.

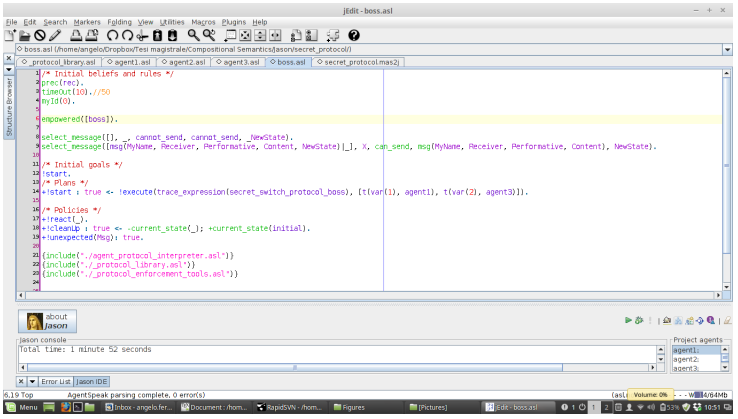


FIGURE 6.10: Secret Protocol - Boss.

limitations; we can observe this in Figure 6.11 where `agent1` receives a `tell_me` message from `agent2` answering with a `say_to` message (no problems with the communication because there are no confidential data). At a certain point of the protocol, an agent can become a secret agent as we can see in Figures 6.12 and 6.13; `agent3` receives the protocol switch message from `boss` becoming a secret agent. In particular, in Figure 6.12 `agent3` manages the protocol switch and starts the preprocessing phase instantiating the new protocol. After that, in Figure 6.13, `agent3` continues projecting the new protocol; in this way, it obtains the local version customized for it. When a normal agent becomes a secret agent could have some confidential information, consequently, it must not release these information to normal agents but only to the other secret agents. In Figure 6.14, in fact, `agent1`, which has received a `tell_me` message from the normal `agent2`, answers with a rejection message (the `shut_up` message in this case). At a certain point of the

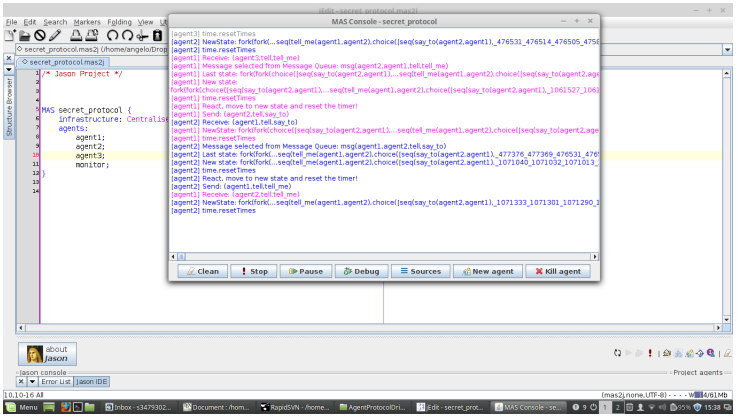


FIGURE 6.11: Secret Protocol - Communication between normal agents.

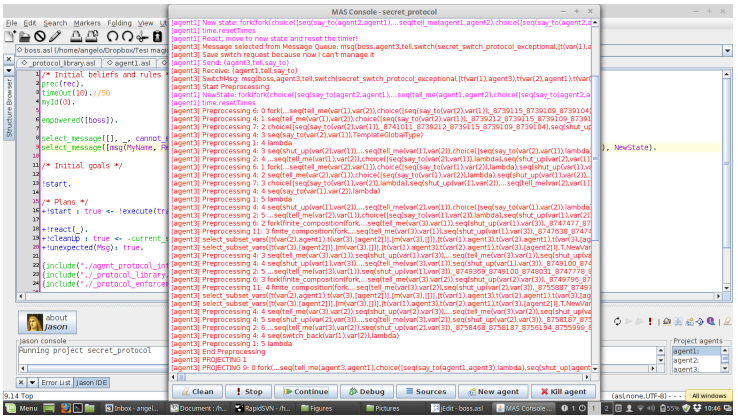


FIGURE 6.12: Secret Protocol - Preprocessing phase - Switch to exceptional behaviour (the `agent3` become a secret agent).

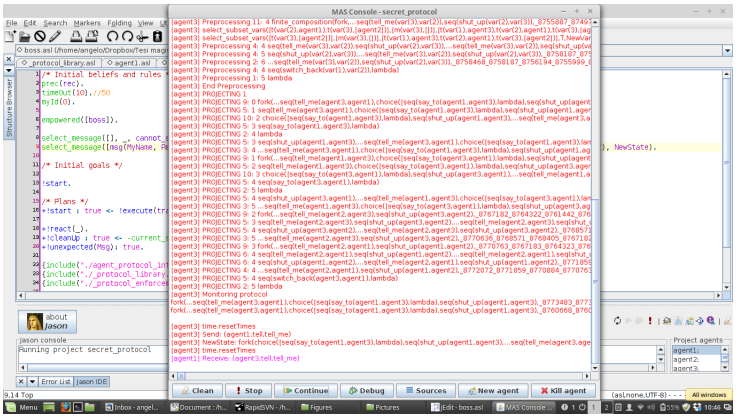


FIGURE 6.13: Secret Protocol - Projection phase - Switch to exceptional behaviour (`agent3` become a secret agent).

protocol, reported in Figure 6.15, a secret agent ceases to be secret and it becomes a normal agent again (it is obtained as a result of a protocol switch).

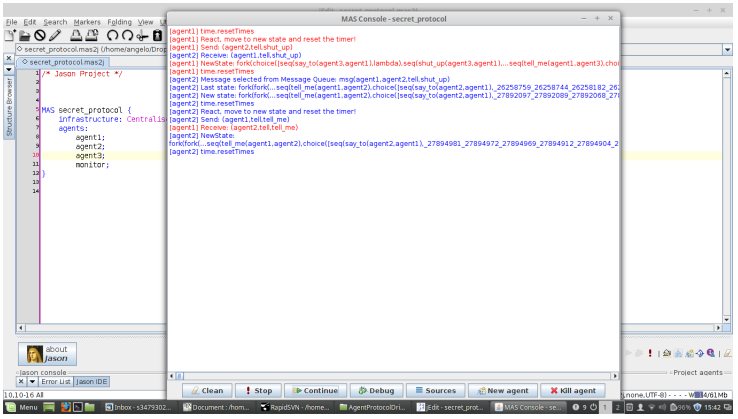


FIGURE 6.14: Secret Protocol - A secret agent can not release confidential information to normal agents.

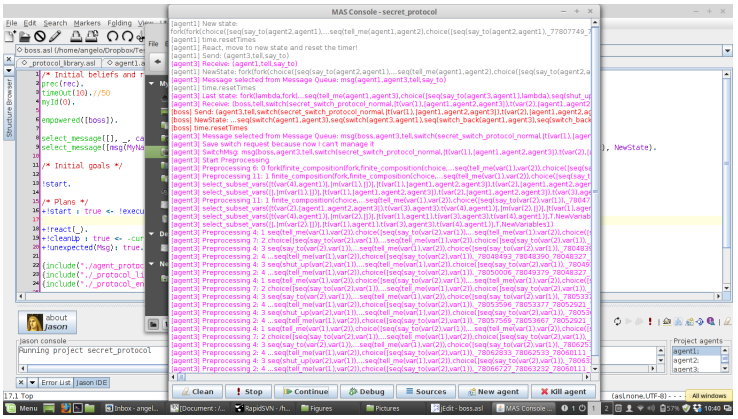


FIGURE 6.15: Secret Protocol - A secret agent which becomes a normal agent.

6.3.3 JADE

JADE, as already anticipated in Section 1.3, is a Java framework to develop MASs, where each agent is an object instance of a class which must extend a common Java class (`Agent` class). In order to create a protocol-driven agent, we have created a class called `AgentProtocolDriven`, which will be extended by the classes that define our protocol-driven agents.

The `AgentProtocolDriven` class defines the protocol-driven agent's interpreter; consequently, each protocol-driven agent class, which extends it, inherits the `setup` method and the cyclic behaviour representing the real interpreter implementation; that is structured following the design level presented in Section 6.2.2, splitting all cases on the base of what the agent can or can not to do in the current state of the protocol.

Before analyzing in more detail the interpreter implementation in Java, we explain how the integration between Java and Prolog was solved.

6.3.3.1 Integration with the framework

To know the actions allowed by the protocol at a given time (namely, which messages the agent can send, which one it is allowed to receive), the agent queries the Prolog library where all the important pieces of information, like the current state of protocol, are maintained; to do this, in JADE, we decided to use a Java library called JPL[9], that makes communication between a Java program and the SWI Prolog engine possible. So, the JADE agent's interpreter can, step by step, ask to Prolog what the agent can or cannot do.

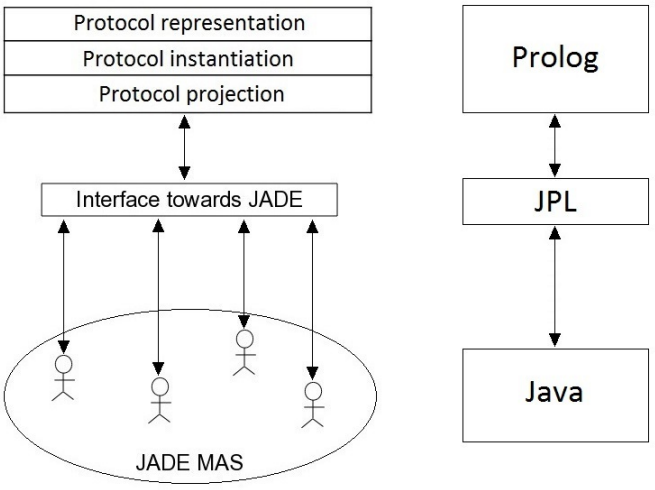


FIGURE 6.16: The implementation of our framework in JADE.

JPL is a set of Java classes and C functions providing an interface between Java and Prolog. JPL uses the Java Native Interface (JNI) to connect to a Prolog engine through the Prolog Foreign Language Interface (FLI), which is more or less in the process of being standardized in various implementations of Prolog. JPL is not a pure Java implementation of Prolog; it makes extensive use of native implementations of Prolog on supported platforms. The current version of JPL only works with SWI-Prolog.
Currently, JPL only supports the embedding of a Prolog engine within the Java VM. Future versions may support the embedding of a Java VM within Prolog, so that, for example, one could take advantage of the rich class structure of the Java environment from within Prolog.
JPL is designed in two layers, a low-level interface to the Prolog FLI and a high-level Java interface for the Java programmer who is not concerned with the details of the Prolog FLI. The low-level interface is provided for C programmers who may wish to port their C implementations which use the FLI to Java with minimal fuss.

[9] `http://www.swi-prolog.org/packages/jpl/`

The initialization of the Prolog engine is carried out with a static block inside the `AgentProtocolDriven` class. The static method `init` of the `JPL` class allows this.

```
/* static block which initializes the Prolog engine */
static{
  JPL.init(); /* Prolog engine initialization */
  /* load the library that contains all implemented protocols */
  createAndCheck("consult", new Atom("prolog/protocollibrary.pl"));
  System.out.println("protocollibrary loaded.");
  /* load the library contains all predicates used
     to instantiate, project, etc... */
  createAndCheck("consult",
                 new Atom("prolog/protocolenforcementtools.pl"));
  System.out.println("protocolenforcementtools loaded.");
}
```

The `createAndCheck` method is also a static method of the `AgentProtocolDriven` class. In particular, there are three versions of this method (overloading).

```
/* method used to execute a predicate represented as a String */
protected static Query createAndCheck(String predicate){
  Query query = new Query(predicate);
  if(!query.hasSolution()){
    throw new PrologException(predicate + " predicate failed");
  }
  return query;
}

/* method used to execute a predicate represented as a functor(term)
   */
protected static Query createAndCheck(String functor, Term term){
  Query query = new Query(functor, term);
  if(!query.hasSolution()){
    throw new PrologException(functor + " " + term + " predicate
        failed");
  }
  return query;
}

/* method used to execute a predicate represented as
   functor(term1, ..., termN) where terms = { term1, ..., termN } */
protected static Query createAndCheck(String functor, Term[] terms){
  Query query = new Query(functor, terms);
  if(!query.hasSolution()){
    throw new PrologException(functor + " predicate failed");
```

```
    }
    return query;
}
```

These methods are necessary to execute all queries in Prolog engine. They create an object of type `Query` (`Query` is a class of JPL library) and, after that, they call the method `hasSolution` which executes in Prolog the predicate passed as argument.

6.3.3.2 The interpreter customization

Each new agent class must extend the `AgentProtocolDriven` class and eventually if necessary overriding the following methods:

- `setup`, method dedicated to initialize the Prolog engine with all predicates necessary to the agent (it is the first method called by JADE on each agent);
- `react`, method dedicated to the agent reaction after a message reception that is expected by the protocol;
- `unexpected`, method dedicated to the agent reaction after a message reception that is unexpected by protocol;
- `select`, method used by the agent in order to select which message to send between those expected by the protocol;
- `cleanup`, method used by the agent to perform all actions necessary before switching to another protocol.

The `react`, `unexpected`, `select` and `cleanup` methods consist in the policies implementation in JADE (we will present an example of methods implementation at the end of this section).

Each agent's setup method must recall the inherited method of its parent (*AgentProtocolDriven* class) in which the main behaviour implementing the interpreter's body is created and added. This is a *Cyclic Behavior* which is executed any time the agent is selected by the JADE schedule. It is like a loop, and in each round the agent can check if can do something coherently with the protocol.

The parent's setup method does not only create a behaviour but it cares about instantiation and projection of the protocol by calling the `execute` method which runs the Prolog predicate: *instantiate_and_project*[10].

[10]It is the very similar to the `execute` plan in Jason.

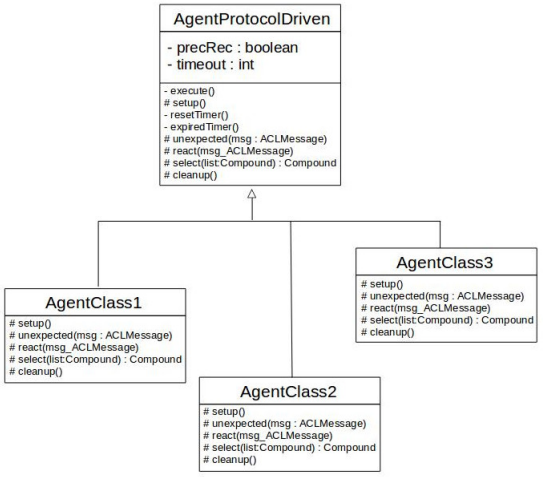


FIGURE 6.17: `AgentProtocolDriven` class hierarchy

```
/* method used by the agent in order to instantiate the protocol
   chosen during the setup phase (see <agentType>.conf) */
private void execute(String protocolName, String protocolParameters){
  /* assert the initial state of the protocol */
  createAndCheck("clean_and_record(" + getLocalName() + ",
     current_state(initial))");

  /* instantiate and project the protocol */
  createAndCheck(
    "instantiate_template_and_project(" +
    protocolName + "," +
    protocolParameters + "," +
    getLocalName() + "," +
    "[" + getLocalName() + "])"
  );
  resetTimer();
}
```

Below we show the Prolog code corresponding to this predicate. It can be easily seen that the code can be broken down into three basic components (as already seen in Figure 6.16):

- the protocol representation;
- the protocol instantiation;
- the protocol projection.

```
/* Predicate that manages the instantiation and
```

```
   the projection of a template trace expression */
instantiate_and_project (Name, ActualParameters,
MyName, ProjectedAgents) :-
  /* get the template trace expression from the library,
     this protocol can have parameter variables */
  trace_expr(Name, T),
  /* preprocessing phase where all the syntactic sugar
     and parameter variables are removed */
  apply(T, CurPar, InsT),
  /* project protocol on this agent */
  project(MyName, InsT, ProjectedAgents, ProjT),
  /* update current state of protocol */
  clean_and_record(MyName, current_state(ProjT)).
```

After this sequence of instructions, inside the Prolog engine the projected protocol of our agent is correctly instantiated and the interpreter can follow it.

The `setup` method of the `AgentProtocolDriven` class ends creating and adding the main behaviour implementing the interpreter's body.

The code of the interpreter implementation in JADE is reported in Appendix A. Below we report only the case which manages a protocol switch message reception.

```
/* setup method that is the first called by JADE when
   the agent is created */
@Override
protected void setup() {
  ...
  /* add the main behaviour that is the interpreter implementation */
  this.addBehaviour(new CyclicBehaviour() {
    @Override
    public void action() {
      /* if a switch is required and consistent with the protocol */
      if(switchMsg != null){
        /* check if it is allowed a switch in the current protocol
           state */
        Query query = new Query("is_time_for_switch(" + getLocalName
            () + "," + switchMsg.getContent() + ", PrSwitch,
            Parameters)");

        Hashtable h; /* get the predicate result */
        if((h = query.oneSolution()) != null){
```

```
            switchMsg = null;
            /* extrapolate all protocol data */
            String protocolName = fromTermToString((Term) h.get("
               PrSwitch"));
            String protocolParameters = fromTermToString((Term) h.get("
               Parameters"));
            System.out.println("\n[" + getLocalName() + "]: SwitchMsg("
                + protocolName + ", " + protocolParameters + ")");
            /* method which implements the cleanup policy */
            cleanUp();
            /* instantiate and project the new protocol */
            execute(protocolName, protocolParameters);
            return; /* continue to the next run */
          }
        }
      ...
      }
  });
}
```

In order to make the interpreter implementation in JADE easier, we had to create some Prolog predicates which allow the Java code to query Prolog in a more compact way.

```
/* if Msg is allowed in the current state of the protocol,
   move to new state and save it */
move_to_next(MyName, Msg) :-
 /* get the current state of the protocol */
 recorded(MyName, current_state(LastState), Ref),
 /* try to do a step, if it is valid in the current state */
 next(LastState, Msg, NewState),
 erase(Ref),
 /* update current state of protocol */
 recorda(MyName, current_state(NewState)).

/* the predicate that checks if is time for a protocol switch
   (depend on the current state of the protocol) */
is_time_for_switch(MyName, switch(PrSwitch, Parameters),
PrSwitch, Parameters) :-
  /* get the current state of the protocol */
  recorded(MyName, current_state(LastState), _),
  next(LastState, msg(_, MyName, _, switch(PrSwitch, Parameters)), _).
```

The `move_to_next` predicate allows the JADE interpreter to change the current state of the protocol (`next` predicate) saving the results as the new protocol state; instead, the `is_time_for_switch` predicate checks if a protocol switch is allowed in the current state of the protocol.

The necessity of these "super predicates" will be discussed in Section 6.3.4.1 where it will be highlighted the JADE implementation problems.

Policies implementation. As already said in Section 6.3.3.2, the policies in JADE are implemented as methods of the `AgentProtocolDriven` class. These methods will be overridden by all classes which extendsit.
The default implementation in `AgentProtocolDriven` class is:

```
/* the default method that manages an unexpected received message */
protected void unexpected(ACLMessage msg){
  /* it prints only a message informing that the message received is
     unexpected by the protocol in the current state */
  System.out.println("\n[" + getLocalName() + "]: Message unexpected
     managed!");
}

/* the default method that manages the agent reaction after the
   reception of a message */
protected void react(ACLMessage msg){
  /* it prints only a message informing that the message received is
     correct and consistent with the current state of the protocol */
  System.out.println("\n[" + getLocalName() + "]: React, move to new
     state and reset the timer!");
}

/* the default method that manages the choice of a message to send (
   default method choose first message of the list) */
protected Compound select(Compound listToSend){
  /* check if the term passed as argument is or not a Prolog list */
  if(listToSend == null || (!listToSend.name().equals(".") && !
     listToSend.name().equals("[]"))){
    throw new IllegalArgumentException("listToSend must be a list");
  }
  /* if the list is empty, return null (which means that there is no
     message to send in the current state of the protocol) */
  if(listToSend.arity() == 0) return null;
  /* return the first message of the list (the simplest possible
     implementation, indeed this is the default behaviour) */
```

```
  return (Compound) listToSend.arg(1);
}

/* the default method that performs all actions necessary before
   switching to another protocol */
protected void cleanup(){ } /* default, there is nothing to do */
```

A class representing a new protocol-driven agent must extend the `AgentProtocolDriven` class overriding these methods.

6.3.3.3 Experiments with JADE

In this section we present some experiments of our work in such a way that we can show the results which have obtained executing our protocol-driven agents inside JADE. All the experiments show the agent's gui in JADE, in this case, the terminal.

The experiments reported below are those related to the examples presented in Chapter 5 (like for Jason implementation); in this way, we can focus directly on the main important aspects of the protocols, considering that we have already presented all logical details.

Iterated Contract Net Protocol. In Figure 6.18 we can see that, as expected by the ICNP protocol, the `initiator` agent sends the `cfp` message to all other agents in the system (in this case the participants `agent2` and `agent3`). After that, a `participant`

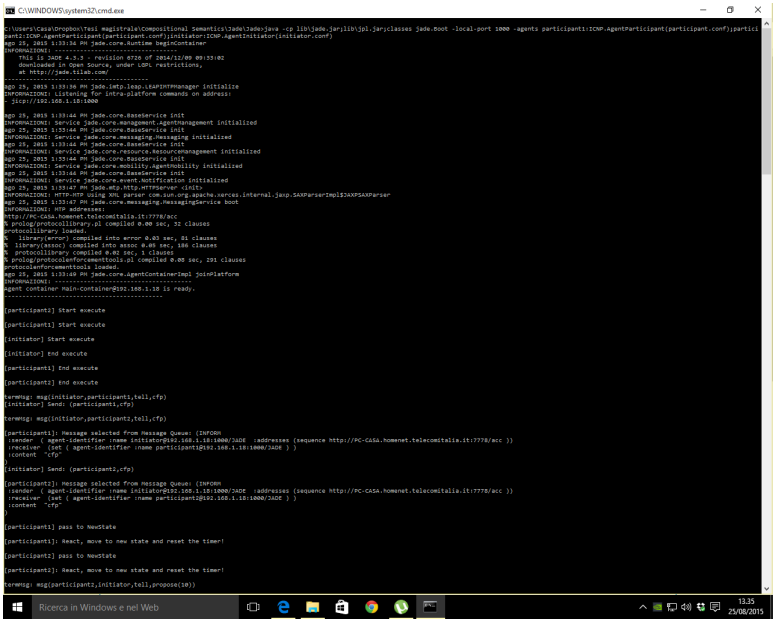


FIGURE 6.18: ICNP - Call for proposal.

agent can send a message containing a proposal to `initiator` (see Figure 6.19); the latter can answer with an against proposal until the value proposed is low enough (in this case the maximum value that can be accepted by the `initiator` is `6`). When the

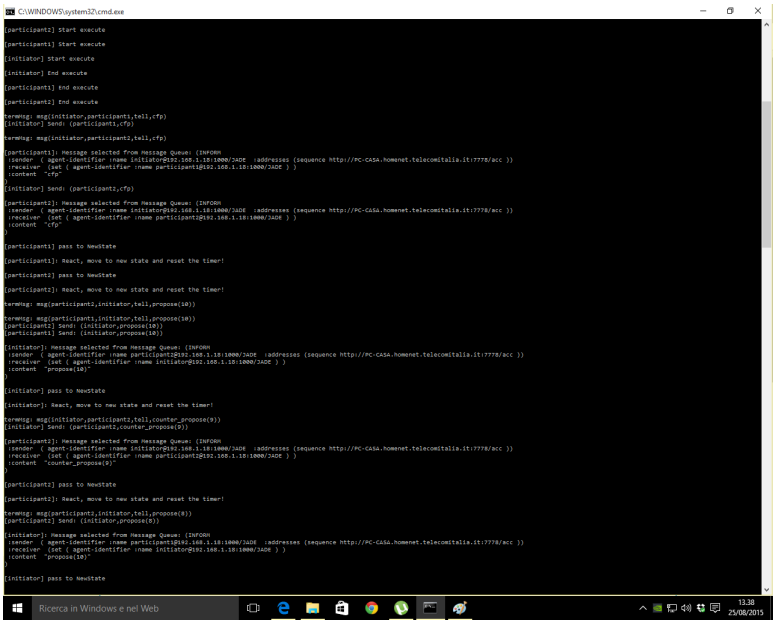


FIGURE 6.19: ICNP - Proposal and counter proposal.

value proposed is less than or equal to `6`, `initiator` accepts the participant who has proposed it (see Figure 6.20) thus ending the protocol.

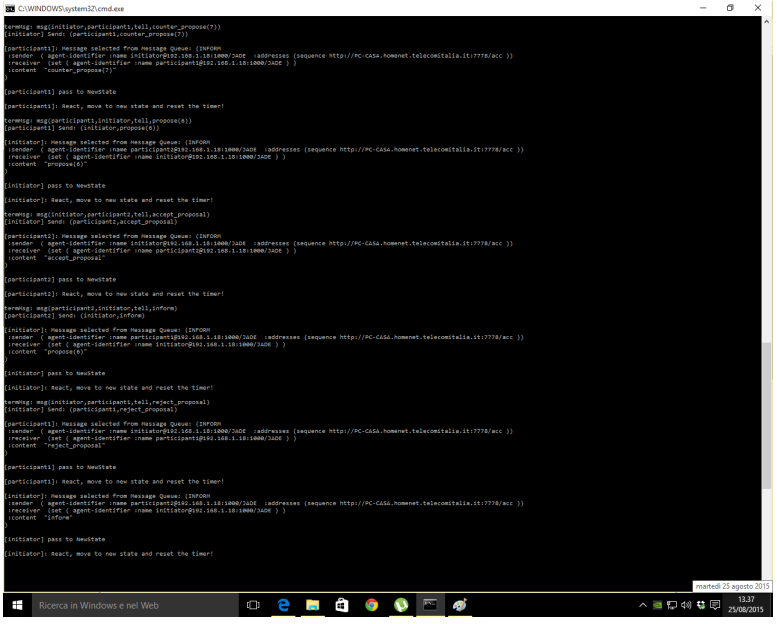


FIGURE 6.20: ICNP - Acceptance and rejection of the proposal.

Auction Protocol. The Auction protocol, as in the case of the Jason implementation, is not really interesting from the point of view of the observable results. As we can see in Figure 6.21, `agent1` sends the `inform_start` and the `cfp_1` messages to `agent2` and `agent3`.

After an exchange of proposals, coherently with the protocol, `agent1` accepts the pro-

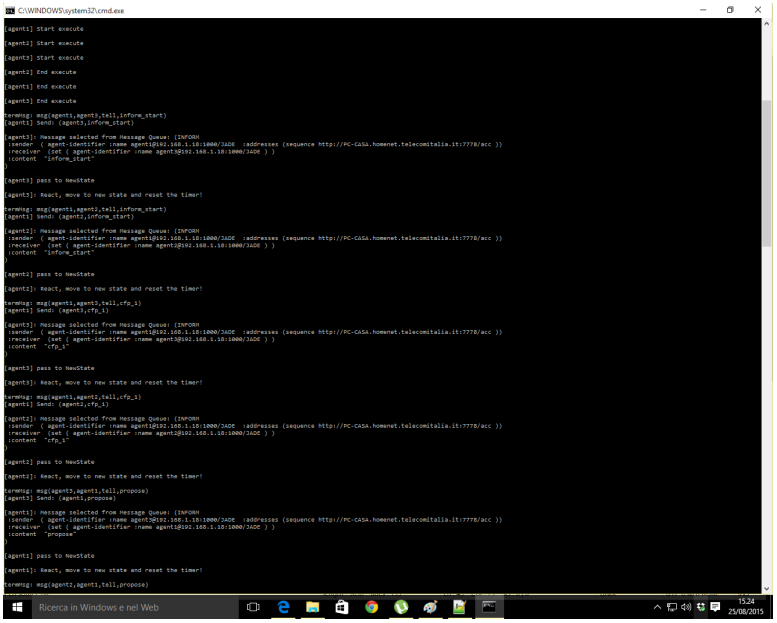


FIGURE 6.21: Auction Protocol - Inform start and call for proposal.

posal of `agent3` rejecting that of `agent2`. Subsequently, `agent1` can either end the

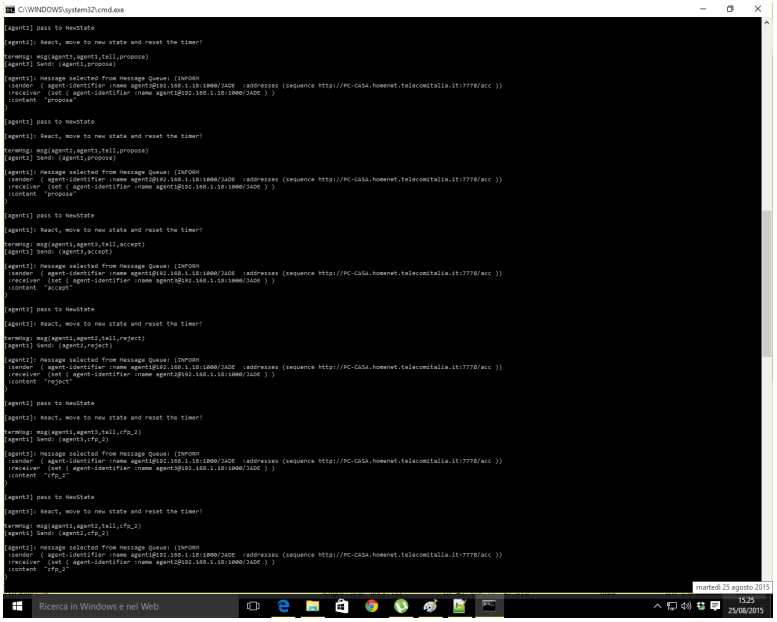


FIGURE 6.22: Auction Protocol - Proposal with acceptance of one participant and rejection of the others.

protocol with the chosen winner, or it can restart the protocol with a `cfp_2` message and, after that, the agents will resend their proposals and so on.

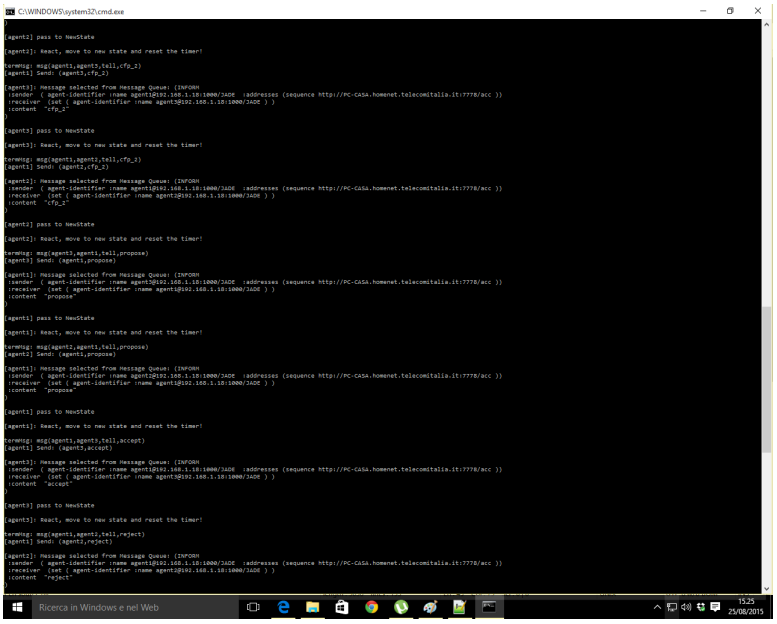


FIGURE 6.23: Auction Protocol - the `cfp_2` message and the restart of the protocol.

Secret Protocol. Initially, see Figure 6.24, all agents are normal and there are no secret agents. The communication happens only among normal agents; consequently, each `tell_me` message is followed by an answer message `say_to` (there are no confidential data). In Figure 6.25, `agent1` changes the protocol after a protocol switch request sent by `boss`, becoming a secret agent. If a normal agent sends a `tell_me` message to `agent1`, it can not answer positively and it rejects the communication with the rejection message `shut_up`. In Figure 6.26, we can see that `agent1` sends a `shut_up` message to `agent2`, which is a normal agent that can not have any confidential information. At a certain point of the protocol, reported in Figure 6.27, after the reception of a protocol switch request from the `boss` agent, `agent1` becomes a normal agent.

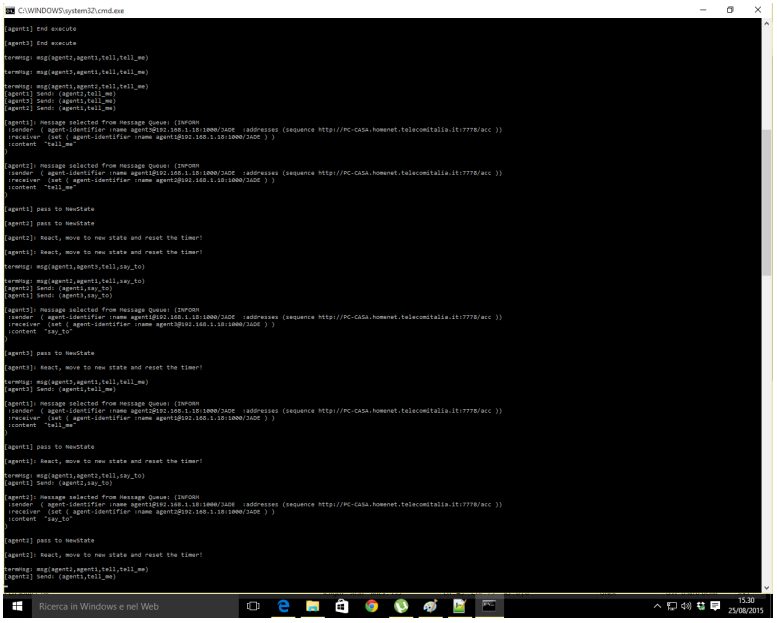


FIGURE 6.24: Secret Protocol - Communication between normal agents.

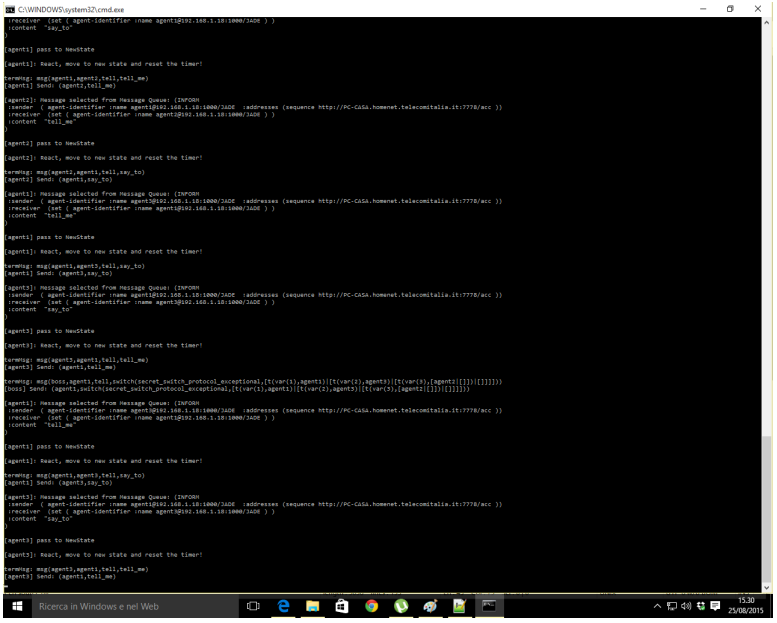


FIGURE 6.25: Secret Protocol - `agent1` becomes a secret agent.

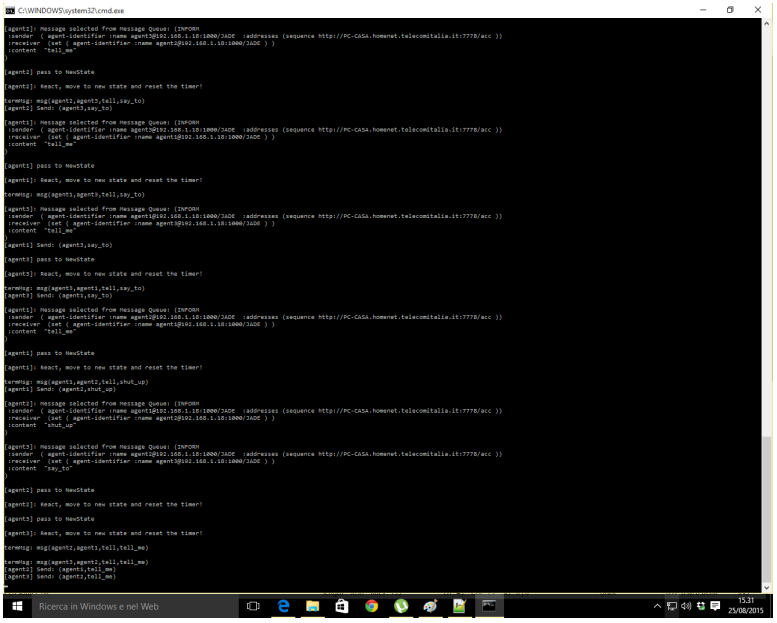


FIGURE 6.26: Secret Protocol - A secret agent can not release confidential information to normal agents.

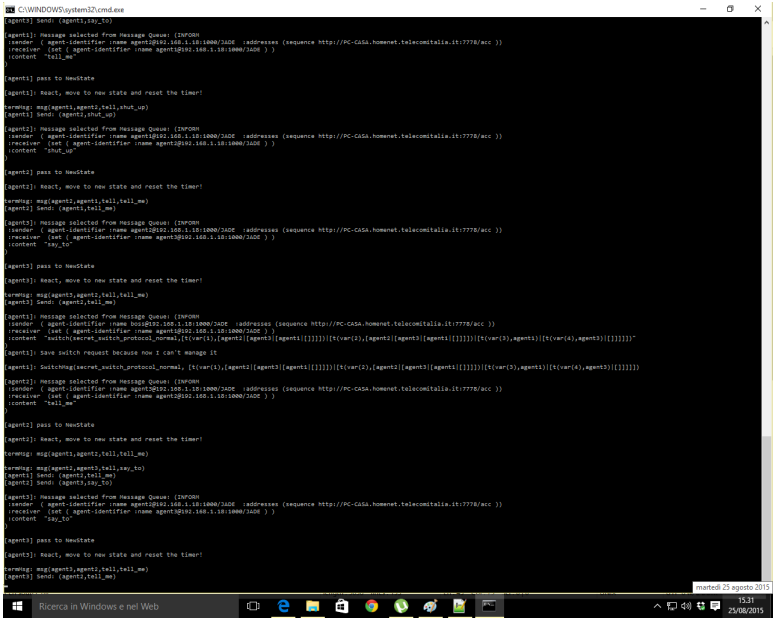


FIGURE 6.27: Secret Protocol - A secret agent which becomes a normal agent.

6.3.4 Comparison between Jason and JADE implementation

Both implementations did not require a huge amount of work, considering that both use the same Prolog predicates; in this way, the workload is totally focused on only the interpreter implementation. As already said, the implementation in Jason has been simpler, above all considering the absence of the need for the integration between the two languages (Prolog code inside).

From the point of view of the interpreter customization, the two implementations are very similar because both follow the same case by case approach, which has been described at design level in Section 6.2.2.

As already mentioned in Section 3.2.5, our framework does not depend on a particular MAS architecture, we can observe that Jason is a tool which allows creating only MASs with a BDI architecture (see Section 1.2), while, JADE is a tool which allows creating MASs that are not related to a special architecture. The only requirements which must be satisfied are those presented in Section 6.1.

6.3.4.1 Problems encountered only with JADE

The interpreter implementation in JADE was more complicated than in Jason, we found many more different problems.

The main problems can be summarized in two specific cases:

- the JPL library does not support cyclic terms;
- SWI Prolog assert predicate does not allow a cyclic term as argument.

It is easy to note that the second problem results from a lack of SWI Prolog in the management of cyclic terms.

In order to solve the first problem, we had to create "super predicates", which are collections of predicates, to ensure that all intermediate executions are made within Prolog and no cyclic term is returned to JADE. We have already presented one of these predicates in Section 6.3.3: the `move_to_next` predicate; this predicate is necessary because the JADE interpreter (Java method) can not call directly the `next` predicate (which instead it is possible in Jason) because the (possible) cyclic term returned is not representable in JPL. In this way, this "super predicates" can call the `next` predicate

saving the result as the new current state of the protocol directly, without the need to return the term to JPL.

The second problem was solved instead using another predicate inside SWI Prolog; the *record* predicate supports cyclic terms and has a behaviour similar to the *assert* predicate.

Chapter 7

Conclusions and Future work

In this thesis we have presented the constrained global type formalism and its extension; namely the trace expression formalism; we have introduced all the basic aspects of this new formalism and, above all, we have compared it with LTL when it is used for runtime verification. The results obtained and discussed in Section 4.4 led us to determine that our formalism is more powerful and suitable to be used for runtime verification respect to LTL (trace expressions were created for this purpose).

Our work does not give only a theoretical contribution, it has a strong impact also at the implementation level; we have described our implementation in the Jason and JADE tools in Sections 6.3.2 and 6.3.3 respectively, reporting our experiments using protocols studied in Chapter 5. Our results reflect those predicted by the theory, in fact, we have created correctly all the MASs corresponding to the protocols reported in the thesis[1].

The most relevant aspect of our approach is that it allows the developer of the MAS to create MASs in a compact and easy way, obtaining as a result a MAS that not only works but that is correct by construction having all the agents which are guided by the same protocol. The originality and significance of this work have been witnessed by many publications where the approach has been discussed:

- D. Ancona, D. Briola, A. Ferrando and V. Mascardi. Global Protocols as First Class Entities for Self-Adaptive Agents. In Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, pages 1019-1029, 2015.

[1]Not all the protocols that we have implemented have been reported.

- A. Ferrando. Protocol-driven agents and their integration in JADE. In Proceedings of the thirtieth Convegno Italiano di Logica Computazionale, CILC 2015, 2015.
- D. Ancona, D. Briola, A. Ferrando and V. Mascardi. Protocols with Exceptions, Timeouts, and Handlers: A Uniform Framework for Monitoring Fail-Uncontrolled and Ambient Intelligence Systems, WOA 2015 special issue, to appear.

In the remainder of this section we discuss some possible extensions of our work towards different directions. For each task we state which targets could be easily achieved ("easy to achieve"), and which are more challenging to achieve ("difficult to achieve").

Section 7.1 presents a theoretical evolution, which consists in studying the expressiveness of trace expressions and in extending the formalism with attributes along the lines of attribute grammars and attribute global types [71]. In Section 7.2 we analyze how to check properties of trace expressions in a static way, thus moving from the original purpose of trace expressions (runtime verification) to a new application domain (static verification and model checking). Section 7.3 presents a more practical research direction, that consists in extending the architecture of our framework for protocol-driven self-adaptive agents in a cooperative way inspired by [10]. Section 7.4 outlines a way to check the suitability of our approach by implementing it on top of MASs frameworks different from JADE and Jason, based on different MAS models.

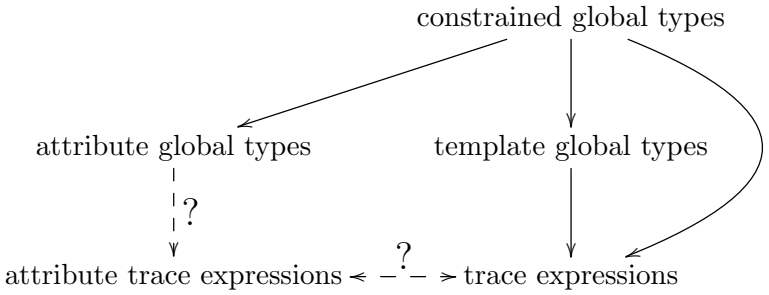
7.1 Trace expressions$_{++}$

Trace expressions can be used both to verify at runtime that the system behaves as expected, as discussed in [8], and to drive the system's behaviour, as discussed in the thesis. In this section we present some research directions related with extensions of trace expressions.

7.1.1 Attribute trace expressions

In [71] **attribute global types**, namely an extension of constrained global types with attributes, were introduced with the purpose of representing more expressive protocols to be used for runtime verification (not for protocol-driven behaviour). Global types equipped with attributes are more expressive, since they allow parametric specifications of protocols, but despite their expressive power they can be still effectively used for dynamic checking of protocols.

Porting the ideas described in [71] into the trace expression formalism, thus obtaining **attribute trace expressions**, would join the benefits of trace expressions with those of attributes.



While we suppose that these attribute trace expressions could still be suitable for runtime verification, we are not sure that they could still drive the behaviour of an agent. With respect to a "generate and test" approach, a monitor used for runtime verification must only be able to test, whereas a protocol driven agent must be able to test, but also to generate. The main difference between these two approaches is thus that the first takes a somehow "passive" point of view because the monitor "simply" monitors a flow of information where actual messages are ground and can be used to unify the attribute values with the actual perceived values: the monitor is not required to have generative capabilities; in the second approach, each protocol-driven agent must also be able to generate new events. Attributes might make the generation stage very difficult or even not possible.

Easy to achieve:
To extend trace expressions using attributes. From the theoretical point of view, to analyze all advantages and disadvantages caused by extending the syntax and semantics. From the practical point of view, to implement this extension in different working prototypes, such that in Jason, in JADE and others, at least as far as runtime verification is concerned.

Difficult to achieve:
To implement prototypes that use attribute trace expressions not only for runtime verification, but also for protocol-driven behaviour.

7.1.2 Expressiveness

In Section 4.4 we compared trace expressions with LTL in the context of runtime verification. However we did not perform a deep analysis of the expressive power of trace

expressions.

We presented many examples in order to compare trace expressions with LTL and we observed that trace expressions can represent some context-free expressions. It is not obvious to state something about the position of the trace expression formalism in Chomsky's hierarchy; we suppose that it could be at least equivalent to context-free languages given the presence of operators like concatenation, shuffle, intersection and conditional, but we have no formal proof.

Considering trace expressions with the attribute extension, it can be expected a higher expressive power.

Another aspect which would be important to analyze is the time complexity arising from the use of trace expressions instead of other formalism such that context-free grammar, LTL, Büchi automatons and so on. In the case of deterministic trace expressions we had good empirical results but we have to better formalize and also studying the case of non-deterministic trace expressions.

Based on these formally proved relationships:

- LTL is equivalent to star-free regular languages, and
- Büchi automatons are equivalent to ω-regular languages (consequently languages recognized by LTL are strictly contained in those recognized by Büchi automatons)

it should be found an algorithm which allows us to move from a trace expression to something like LTL but more expressive in order to give substance to our claim that trace expressions are supposed to be more expressive of LTL and also of Büchi automatons.

These formalisms can be also compared in the context of static analysis, for example, in the case of Büchi automatons we can recognize only traces with infinite length, while using trace expressions, we can represent traces with both infinite and finite length.

Easy to achieve:
To study trace expressions expressiveness, properties (for instance closure properties) and relations with other formalism in addition to LTL.

Difficult to achieve:
To find the exact position of trace expressions in Chomsky's hierarchy.

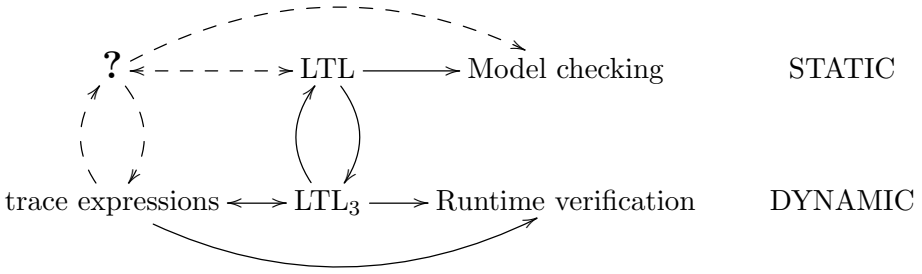
7.2 Static Verification and Model checking of MASs

In Section 4.4 we compared trace expressions with LTL in the context of runtime verification in order to demonstrate the power of our approach.

LTL is basically used in model checking, in fact in order to pass to runtime verification we had to consider LTL_3 which is a three-valued extension of LTL. In Section 4.4 we demonstrated that it is always possible, given an LTL_3 monitor, to extrapolate a trace expression so that its monitor is equivalent.

An interesting development of our work could be to try to directly verify some properties on a trace expression. This would be extremely useful because it would allow us to check if a protocol has some property, such that deadlock absence, determinism, contractiveness and so on, before building a MAS which is driven by that protocol.

In some sense, this would be the converse of what we have done in [9] where we started from LTL, which is used in static verification, moving after on trace expressions, which are used in runtime verification.



As we can see in the diagram above, we will try to retrace our steps so that, given a trace expression, we can consequently obtain the corresponding LTL property (if there is one, i.e there are some trace expressions that cannot be represented using LTL).

Easy to achieve:
To understand, given a trace expression, which properties we could be able to verify (checking them accordingly).

Difficult to achieve:
To bring trace expressions in the model checking world by comparing them with LTL and Büchi automatons. It would be extremely useful to completely verify trace expressions and not only some predefined properties.

7.3 Coo-PDA

Another interesting research direction could be that of Cooperative Protocol-Driven agents (Coo-PDA). As mentioned at the beginning of the chapter, this evolution derives from [10]; in that work the authors described a MAS architecture where all agents, implementing a BDI architecture, can exchange plans (namely, behavioral knowledge).

The most important difference to note is that the Coo-PDA extension would not allow agents to exchange **plans** but **protocols**; this is crucial because it allows working with several MASs architectures and not only with BDI agents.

The idea consists in extending our work promoting an agent to super agent which knows many protocols (expressed for instance as trace expressions). All the other agents ask it to follow a protocol whenever is necessary.

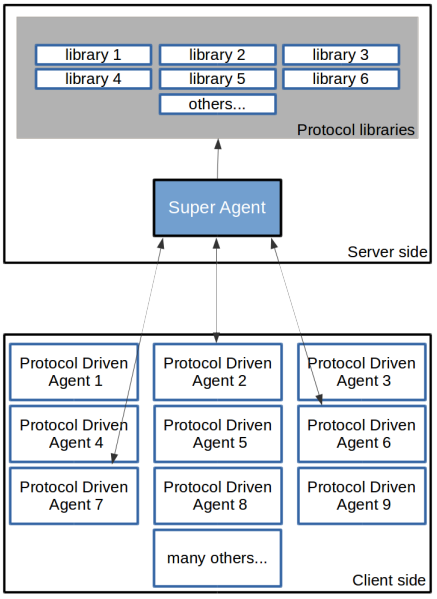


FIGURE 7.1: Coo-PDA architecture.

The server side is divided into two main parts:

1. Protocol libraries part, where all the libraries used by the super agent are defined. Each library contains a set of protocols written in some formalism, for example trace expressions.

2. Administrator part, that is where the Super Agent, or rather the agent which is dedicated to providing the necessary protocols to client agents, is defined.

In Figure 7.2, we can see a short and simplified protocol exchange flow between a protocol driven agent (client) and the super agent (server). Here we take advantage of the protocol switching (introduced in Section 3.2.5). In some sense, the Coo-PDA approach is a direct consequence of this; in fact, it keeps all the features presented in Section 3.2.5 unchanged except for the architecture, where there is a super agent (special agent) and all the others which contact him in order to obtain new protocols or change some existing one.

Easy to achieve:
To create the Coo-PDA architecture starting from our work in order to obtain a generic and modular implementation, maintainable and easy to extend. Agents will directly require each protocol only by "name".

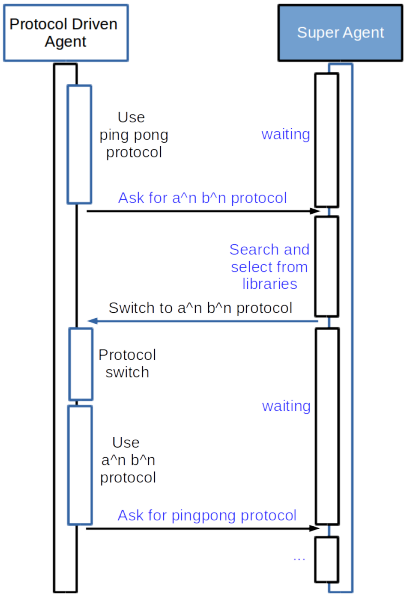


FIGURE 7.2: Coo-PDA flow example.

Difficult to achieve:
To extend the Coo-PDA architecture so that each agent can require a protocol passing a set of properties which it claims to be maintained and some other information, receiving a trace expression that meets the requirements (Figure 7.3: the client agent does not need to know explicitly the name of a protocol but it can simply state what it expects the protocol can do or can achieve).

7.4 Porting on other frameworks

Two working prototypes for protocol-driven agents exist, demonstrating the feasibility of our approach (as already presented in Sections 6.3.2 and 6.3.3). While integrating

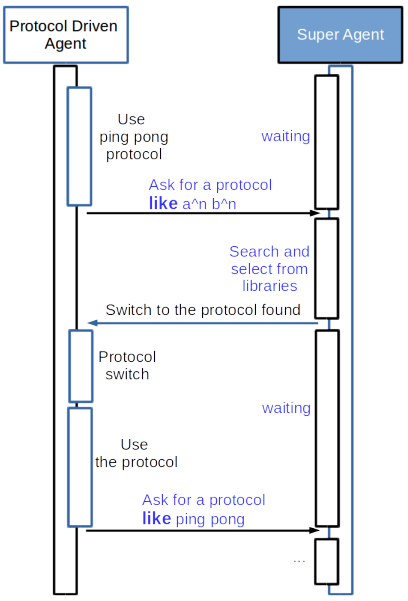


FIGURE 7.3: Coo-PDA flow evolution example

our protocol-driven agents into Jason was easy because of Jason's native support to Prolog, integrating them into JADE was not. However, that attempt - which in the end was successful - makes us confident that it can be possible to integrate our approach into almost any agent framework, provided that an interface between the framework's language and Prolog is provided.

Easy to achieve:
To implement our work on top of at least one other MAS framework such as 2APL [45], 3APL [61], GOAL [62], Jadex [25].

Difficult to achieve:
To implement our work on top of many other MAS frameworks, thus demonstrating its generality and portability.

Appendix A

JADE interpreter implementation

```
/* setup method that is the first called by Jade when
   the agent is created */
@Override
protected void setup() {
  ...
  execute(protocolName, protocolParameters);
  ...
  /* add the main behaviour that is the interpreter implementation */
  this.addBehaviour(new CyclicBehaviour() {
    @Override
    public void action() {
      /* if a switch is required and consistent with the protocol */
      if(switchMsg != null){
        /* check if it is allowed a switch in the current protocol
           state */
        Query query = new Query("is_time_for_switch(" + getLocalName
            () + "," + switchMsg.getContent() + ", PrSwitch,
            Parameters)");

        Hashtable h; /* get the predicate result */
        if((h = query.oneSolution()) != null){
          switchMsg = null;
          /* extrapolate all protocol data */
          String protocolName = fromTermToString((Term) h.get("
              PrSwitch"));
          String protocolParameters = fromTermToString((Term) h.get("
              Parameters"));
          System.out.println("\n[" + getLocalName() + "]: SwitchMsg("
               + protocolName + ", " + protocolParameters + ")");
```

```
      /* method which implements the cleanup policy */
      cleanUp();
      /* instantiate and project the new protocol */
      execute(protocolName, protocolParameters);
      return; /* continue to the next run */
    }
  }

  /* get the first message in the message queue (null if there
     are not messages) */
  ACLMessage msg = receive();

  /* if the agent gives priority to receive messages and the
     message queue is empty and it does not catch up the timeout
     */
  if(precRec && msg == null && !expiredTimer()){
    return; /* it is a foo run */
  }

  /* statement which manages the error case where the agent
     receives a message but the protocol does not allow it */
  if(msg != null && new Query("inMsgs(" + getLocalName() + ",[])"
     ).hasSolution()){
    /* method which implements the unexpected policy */
    unexpected(msg);
    return; /* continue to the next run */
  }

  /* statement which manages the agent reception */
  if(msg != null && (precRec || (!precRec && new Query("outMsgs("
      + getLocalName() + ",[])").hasSolution() && expiredTimer())
     )){
    System.out.println("\n[" + getLocalName() + "]: Message
       selected from Message Queue: " + msg);
    /* if the message selected is not a protocol switch request
       */
    if(!msg.getContent().contains("switch")){
      /* if the message selected is allowed in the current state
         of the protocol, move to new state and save it */
      Query q1 = new Query("move_to_next(" + getLocalName() + ",
         msg(" + msg.getSender().getLocalName() + "," +
         getLocalName() + "," + toKQMLPerformative(msg.
         getPerformative()) + "," + msg.getContent() + "))");
      if(q1.hasSolution()){
```

```
        System.out.println("\n[" + getLocalName() + "] pass to
            NewState");
        /* method which implements the react policy */
        react(msg);
        resetTimer();
      }
      /* the message selected is unexpected by the protocol */
      else{
        /* method which implements the unexpected policy */
        unexpected(msg);
      }
      return; /* continue to the next run */
    }
    /* if the message selected is a protocol switch request */
    else{
      boolean found = false;
      for(String str : empoweredAgents){
        if(msg.getSender().getLocalName().equals(str)){
          found = true;
          break;
        }
      }
      /* if the agent that requires a protocol switch can ask for
          it */
      if(found){
        System.out.println("[" + getLocalName() + "]: Save switch
            request because now I can't manage it");
        /* save the protocol switch message */
        switchMsg = msg;
      }
      else{ /* otherwise */
        System.out.println("[" + getLocalName() + "]: Discard
            switch request because Sender can't request it");
      }
      return; /* continue to the next run */
    }
  }

  /* statement that manages the message sending */
  Query q = new Query("outMsgs("+getLocalName()+",ListToSend)");
  if(q.hasSolution()){
    Compound listToSend = (Compound) q.oneSolution().get("
        ListToSend");
    Compound termMsg = select(listToSend);
```

```
        if(termMsg == null){
          /* re-put the message on top of the message queue */
          if(msg != null) putBack(msg);
          return; /* continue to the next run */
        }
        /* the agent can send a message */
        System.out.println("\ntermMsg:" + fromTermToString(termMsg));

        /* if the message selected is allowed in the current state of
            the protocol move to new state and save it */
        q = new Query("move_to_next", new Term[]{ new Atom(
           getLocalName()), termMsg });
        Hashtable h;
        if((h = q.oneSolution()) != null){
          String receiver = termMsg.arg(2).toString();

          ACLMessage msgToSend = new ACLMessage(toFipaPerformative(
             termMsg.arg(3).toString()));
          msgToSend.setSender(getAID());
          msgToSend.addReceiver(new AID(receiver, AID.ISLOCALNAME));

          msgToSend.setContent(fromTermToString(termMsg.arg(4)));
          /* send the message selected */
          send(msgToSend);
          System.out.println("["+getLocalName()+"] Send: ("+receiver+
             ","+fromTermToString(termMsg.arg(4))+")");

          resetTimer();
        }
      }
      /* re-put the message on top of the message queue */
      if(msg != null) putBack(msg);
    }
  });
}
```

Appendix B

Jason interpreter implementation

```
/* the main plan that resets the agent's timer, then initializes
   the protocol and executes the interpreter */
+!startInterpreter(Pr) :
  myId(Id) /* get the agent's identifier */
  <-
  time.resetTimer(Id); /* reset the timer timeout */
  -protocol(_); +protocol(Pr); /* save the current protocol used */
  !executeInterpreter.  /* run the interpreter */

/* if a protocol switch is required and it is consistent with
   the protocol */
+!executeInterpreter :
  /* in a previous run a protocol switch was required */
  switchMsg(msg(S, MyName, Performative, switch(PrSwitch, Parameters))) &
  current_state(LastState) &
  next(LastState, msg(S, MyName, Performative,
       switch(PrSwitch, Parameters)), NewState)
  <- /* Agent can manage protocol switch now */
  .print("SwitchMsg: ", msg(S, MyName, Performative,
         switch(PrSwitch, Parameters)));
  /* Remove protocol switch request from the agent's knowledge base */
  -switchMsg(_); +switchMsg(null);
  /* If the agent needs of some operations before it can switch to the
     new protocol (each agent must have its implementation) */
  !cleanUp; /* In Jason it is implemented as a goal */
```

```
  /* Start the new protocol */
  !execute(trace_expression(PrSwitch), Parameters).

/* if the agent gives priority to receive messages but the message queue
   is empty (however it does not catch up the timeout) */
+!executeInterpreter :
  myId(Id) & /* get the agent's identifier */
  messageQueue(Msgs) & .empty(Msgs) & /* The message queue is empty */
  /* the agent gives priority to receive messages, this information is
     maintained thanks to a belief which each agent must have */
  prec(rec) &
  timeOut(T) & not time.expired(T, Id)
  <-
  /* this run is foo but maybe in the next
     the agent will receive a message */
  !executeInterpreter.

/* the plan that manages an error case where the agent receives a message
   but the protocol does not allow it (the priority does not matter) */
+!executeInterpreter :
  messageQueue(Msgs) &
  not .empty(Msgs) & /* the message queue is not empty */
  inMsgs(ListToReceive) &
  .empty(ListToReceive) & /* the agent can not receive a message now */
  pop(Msgs, Msg, Msgs1) /* remove the message from the message queue */
  <-
  -messageQueue(_); +messageQueue(Msgs1); /* update the message queue */
  .print("Message unexpected managed!");
  /* goal which manages an unexpected message
     (each agent must have its implementation) */
  !unexpected(Msg); /* In Jason it is implemented as a goal */
  !executeInterpreter. /* continue to run the interpreter */

/* the plan that manages the agent message reception */
+!executeInterpreter :
  myId(Id) & /* get the agent's identifier */
  messageQueue(Msgs) &
  not .empty(Msgs) & /* the message queue is not empty and */
  (prec(rec) | /* the agent gives either priority to receive messages or */
```

```
  (
    prec(send) & /* the agent gives priority to send messages and */
    outMsgs(ListToSend) &
    /* the protocol allows the agent to send some messages and */
    .empty(ListToSend) &
    timeOut(T) & time.expired(T, Id) /* the timeout is expired */
  )) &
  pop(Msgs, Msg, Msgs1) /* remove the message from the message queue */
  <-
  -messageQueue(_); +messageQueue(Msgs1); /* update the message queue */
  .print("Message selected from Message Queue: ", Msg);
  /* the sub goal created in order to distinguish the switch cases */
  !subExecuteInterpreter(Msg);
  !executeInterpreter. /* continue to run the interpreter */

+!subExecuteInterpreter(Msg) :
  myId(Id) & /* get the agent's identifier */
  Msg = msg(Sender, MyName, Performative, Content) &
  /* the message is not a protocol switch request */
  not Content = switch(PrSwitch, _) &
  current_state(LastState) &
  .print("Last state: ", LastState) &
  next(LastState, Msg, NewState) & /* the message matches the protocol */
  .print("New state: ", NewState)
  <-
  /* goal which changes the agent's knowledge base
     (each agent must have its implementation) */
  !react(Msg); /* In Jason it is implemented as a goal */
  !move_to_state(NewState); /* save the current state of the protocol */
  time.resetTimer(Id); /* reset the timer timeout */
  .print("React, move to new state and reset the timer!").

+!subExecuteInterpreter(Msg) :
  /* the message is a protocol switch request */
  Msg = msg(Sender, MyName, Performative, switch(PrSwitch, Parameters)) &
  /* the sender has the power to require a protocol switch */
  empowered(AgentsEmpowered) &
  .member(Sender, AgentsEmpowered)
  <-
```

```
  .print("Save switch request because now I can't manage it");
  /* the agent manages protocol switch when the current protocol
     will provide it */
  -switchMsg(_); +switchMsg(Msg).

+!subExecuteInterpreter(Msg) :
  /* Message is a switch request */
  Msg = msg(Sender, MyName, Performative, switch(PrSwitch, Parameters)) &
  /* but the sender has not the power to request a protocol switch */
  empowered(AgentsEmpowered) &
  not .member(Sender, AgentsEmpowered)
  <- /* the agent does nothing, it discards simply the message */
  .print("Discard switch request because Sender can't request it").

/* if the agent executes this goal, the message into the message queue
   is not expected from the protocol in the current state */
+!subExecuteInterpreter(Msg) :
  true <-
   /* goal which manages an unexpected message
     (each agent must have its implementation) */
  !unexpected(Msg). /* In Jason it is implemented as a goal */

/* the plan which manages the message sending */
+!executeInterpreter :
  myId(Id) & /* get the agent's identifier */
  outMsgs(ListToSend) &
  /* goal which manages the message selection */
  select(ListToSend, _, can_send, msg(MyName, Receiver, Performative,
         Content), NewState) & /* In Jason it is implemented as a goal */
  /* check the message selected */
  has_type(msg(MyName, Receiver, Performative, Content), _) &
  <-
  .send(Receiver, Performative, Content); /* send the message */
  .print("Send: (", Receiver, ",", Performative, ",", Content, ")");
  !move_to_state(NewState); /* save the current state of the protocol */
  .print("NewState: ", NewState);
  time.resetTimer(Id); /* reset the timer timeout */
  !executeInterpreter. /* continue to run the interpreter */
```

```
/* in this round the agent did nothing but maybe the
   next round it will do something */
+!executeInterpreter :
  true <- !executeInterpreter.
```

Bibliography

[1] ACM transactions on autonomous and adaptive systems (taas).

[2] IEEE standard for software verification and validation. *IEEE Std 1012-2004 (Revision of IEEE Std 1012-1998)*, pages 1–110, June 2005.

[3] D. Ancona, M. Barbieri, and V. Mascardi. Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In S. Y. Shin and J. C. Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1377–1379. ACM, 2013.

[4] D. Ancona, D. Briola, A. El Fallah Seghrouchni, V. Mascardi, and P. Taillibert. Efficient verification of MASs with projections. In F. Dalpiaz and J. D. M. B. van Riemsdijk, editors, *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Revised Selected Papers*, volume 8758 of *Lecture Notes in Computer Science*, pages 246–270. Springer, 2014.

[5] D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Global protocols as first class entities for self-adaptive agents. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015*, pages 1019–1029, 2015.

[6] D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Protocols with exceptions, timeouts, and handlers: A uniform framework for monitoring fail-uncontrolled and ambient intelligence systems. Submitted to WOA'15 special issue, to appear.

[7] D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In *DALT 2012*, volume 7784 of *LNAI*, pages 76–95. Springer, 2012.

[8] D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In M. Baldoni,

L. A. Dennis, V. Mascardi, and W. Vasconcelos, editors, *Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers*, volume 7784 of *Lecture Notes in Computer Science*, pages 76–95. Springer, 2012.

[9] D. Ancona, A. Ferrando, and V. Mascardi. Runtime verification with trace expressions and LTL logics. In *Technical report DIBRIS*, 2015.

[10] D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI model with cooperativity. In J. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies*, volume 2990 of *Lecture Notes in Computer Science*, pages 109–134. Springer Berlin Heidelberg, 2004.

[11] J. Austin. *How to Do Things with Words*. Oxford, 1962.

[12] M. Baldoni, C. Baroglio, and F. Capuzzimati. 2COMM: A commitment-based MAS architecture. In M. Cossentino, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Engineering Multi-Agent Systems*, volume 8245 of *Lecture Notes in Computer Science*, pages 38–57. Springer Berlin Heidelberg, 2013.

[13] M. Baldoni, C. Baroglio, and F. Capuzzimati. A commitment-based infrastructure for programming socio-technical systems. *ACM Trans. Internet Technol.*, 14(4):23:1–23:23, Dec. 2014.

[14] M. Baldoni, C. Baroglio, and F. Capuzzimati. Typing multi-agent systems via commitments. In F. Dalpiaz, J. Dix, and M. van Riemsdijk, editors, *Engineering Multi-Agent Systems*, volume 8758 of *Lecture Notes in Computer Science*, pages 388–405. Springer International Publishing, 2014.

[15] M. Baldoni, C. Baroglio, F. Capuzzimati, and R. Micalizio. Programming with commitments and goals in JaCaMo+. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '15, pages 1705–1706, Richland, SC, 2015. International Foundation for Autonomous Agents and Multiagent Systems.

[16] A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *ASWEC'06. IEEE*, 2006.

[17] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009. in press.

[18] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[19] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[20] K. Bhargavan, S. Chandra, P. J. Mccann, and C. A. Gunter. What packets may come: Automata for network monitoring. In *Proc. POPL*, pages 206–219. ACM Press, 2001.

[21] K. Bhargavan and C. A. Gunter. Requirements for a practical network event recognition language. *Electronic Notes in Theoretical Computer Science*, 70(4):1 – 20, 2002. RV'02, Runtime Verification 2002 (FLoC Satellite Event).

[22] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.

[23] M. Bratman. *Intention, Plans, and Practical Reason*. Center for the Study of Language and Information, 1987.

[24] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355, 1988.

[25] L. Braubach, W. Lamersdorf, and A. Pokahr. Jadex: Implementing a BDI-infrastructure for JADE agents, 2003.

[26] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[27] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[28] N. Bulling, M. Dastani, and M. Knobbout. Monitoring norm violations in multi-agent systems. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 491–498, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[29] G. Cabri, M. Puviani, and F. Zambonelli. Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In *Collaboration Technologies and Systems (CTS), 2011 International Conference on*, pages 508–515, 2011.

[30] D. Capera, J. George, M.-P. Gleizes, and P. Glize. The AMAS theory for complex problem solving based on self-organizing cooperative agents. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, pages 383–388, 2003.

[31] G. Casella and V. Mascardi. West2East: Exploiting WEb Service Technologies to Engineer Agent-Based SofTware. *Int. J. Agent-Oriented Softw. Eng.*, 1(3/4):396–434, 2007.

[32] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multiparty session. *Logical Methods in Computer Science*, 8(1), 2012.

[33] L. Cernuzzi and F. Zambonelli. Dealing with adaptive multi-agent organizations in the Gaia methodology. In J. Müller and F. Zambonelli, editors, *Agent-Oriented Software Engineering VI*, volume 3950 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin Heidelberg, 2006.

[34] F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA 2007*, pages 569–588, 2007.

[35] C. Chopinaud, A. El Fallah-Seghrouchni, and P. Taillibert. Automatic generation of self-controlled autonomous agents. In *Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, pages 755–758, 2005.

[36] E. M. Clarke. 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[37] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[38] J. Cohen, D. Perrin, and J.-E. Pin. On the expressive power of temporal logic. *J.Comput. Syst.Sci. 46*, pages 271–294, 1993.

[39] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Self-adaptive monitors for multiparty sessions. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014*, pages 688–696. IEEE, 2014.

[40] M. Cossentino. From requirements to code with the PASSI methodology. *Agent-oriented methodologies*, 3690:79–106, 2005.

[41] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Comput. Sci.*, 25:95–169, 1983.

[42] N. Criado, E. Argente, P. Noriega, and V. J. Botti. Reasoning about constitutive norms in BDI agents. *Logic Journal of the IGPL*, 22(1):66–93, 2014.

[43] G. Cugola, C. Ghezzi, and L. Pinto. DSOL: a declarative approach to self-adaptive service orchestrations. *Computing*, 94(7):579–617, 2012.

[44] F. Dalpiaz, P. Giorgini, and J. Mylopoulos. Adaptive socio-technical systems: a requirements-based approach. *Requirements Engineering*, 18(1):1–24, 2013.

[45] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[46] G. De Giacomo, Y. Lespérance, and C. Muise. On supervising agents in situation-determined ConGolog. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '12, pages 1031–1038, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.

[47] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, Dec. 2004.

[48] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP'12 (part of ETAPS 2012)*, LNCS. Springer, 2012.

[49] G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos. Self-organization in multi-agent systems. *Knowl. Eng. Rev.*, 20(2):165–189, 2005.

[50] 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*, CIKM '94, pages 456–463, New York, NY, USA, 1994. ACM.

[51] A. Fip. *FIPA English Auction Interaction Protocol Specification (XC00031F)*. FIPA TC Communication, Aug. 2001.

[52] A. Fip. *FIPA Iterated Contract Net Interaction Protocol Specification*. FIPA, 2001.

[53] FIPA. FIPA ACL message structure specification. Approved for standard, Dec. 6^{th}, 2002, 2002.

[54] N. Fornara, F. Viganò, and M. Colombetti. Agent communication and artificial institutions. *Autonomous Agents and Multi-Agent Systems*, 14(2):121–142, 2007.

[55] N. Fornara, F. Viganò, M. Verdicchio, and M. Colombetti. Artificial institutions: a model of institutional reality for open multiagent systems. *Artificial Intelligence and Law*, 16(1):89–105, 2008.

[56] S. N. Gerard and M. P. Singh. Evolving protocols and agents in multiagent systems. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 997–1004, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[57] M.-P. Gleizes. Self-adaptive complex systems. In M. Cossentino, M. Kaisers, K. Tuyls, and G. Weiss, editors, *Multi-Agent Systems*, volume 7541 of *Lecture Notes in Computer Science*, pages 114–128. Springer Berlin Heidelberg, 2012.

[58] A. Goodloe and L. Pike. Monitoring distributed real-time systems: A survey and future directions, 2010.

[59] Z. Guessoum, M. Ziane, and N. Faci. Monitoring and organizational-level adaptation of multi-agent systems. In *Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on*, pages 514–521, 2004.

[60] C. Hahn, I. Zinnikus, S. Warwas, and K. Fischer. Automatic generation of executable behavior: A protocol-driven approach. In M.-P. Gleizes and J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering X*, volume 6038 of *Lecture Notes in Computer Science*, pages 110–124. Springer Berlin Heidelberg, 2011.

[61] K. V. Hindriks, F. S. De Boer, W. Van Der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, Nov. 1999.

[62] K. V. Hindriks and J. Dix. *GOAL: A Multi-Agent Programming Language Applied to an Exploration Game*. Springer-Verlag, 2014.

[63] IBM Corp. *An architectural blueprint for autonomic computing*. IBM Corp., USA, 2004.

[64] N. R. Jennings, K. P. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[65] T. Juan and L. Sterling. The ROADMAP meta-model for intelligent adaptive multi-agent systems in open environments. In P. Giorgini, J. Müller, and J. Odell, editors, *Agent-Oriented Software Engineering IV*, volume 2935 of *Lecture Notes in Computer Science*, pages 53–68. Springer Berlin Heidelberg, 2004.

[66] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[67] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, and G. Rosu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In *RV'14*, volume 8734, pages 285–300. Springer, 2014.

[68] C. B. M. Baldoni and F. Capuzzimati. Reasoning about social relationships with Jason. *Autonomous Agents and Multi-Agent Systems*, 2014.

[69] A. U. Mallya and M. P. Singh. Modeling exceptions via commitment protocols. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, pages 122–129, New York, NY, USA, 2005. ACM.

[70] M. Mansouri-samani and M. Sloman. Monitoring distributed systems (a survey), 1992.

[71] V. Mascardi and D. Ancona. Attribute global types for dynamic checking of protocols in logic-based multiagent systems. *TPLP*, 13(4-5-Online-Supplement), 2013.

[72] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In *ATAL*, pages 347–360. Springer Verlag, 1995.

[73] S. Merz. Model checking: A tutorial overview. In *Modeling and Verification of Parallel Processes*, pages 3–38. Springer-Verlag, 2001.

[74] T. Miller and P. McBurney. Using constraints and process algebra for specification of first-class agent interaction protocols. In G. M. P. O'Hare, A. Ricci, M. J. O'Grady, and O. Dikenelli, editors, *Engineering Societies in the Agents World VII, 7th International Workshop, ESAW 2006, Dublin, Ireland, September 6-8, 2006 Revised Selected and Invited Papers*, volume 4457 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2006.

[75] T. Miller and P. McBurney. Annotation and matching of first-class agent interaction protocols. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '08, pages 805–812, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[76] T. Miller and P. McBurney. Propositional dynamic logic for reasoning about first-class agent interaction protocols. *Computational Intelligence*, 27(3):422–457, 2011.

[77] M. Morandini, F. Migeon, M.-P. Gleizes, C. Maurel, L. Penserini, and A. Perini. A goal-oriented approach for modelling self-organising MAS. In H. Aldewereld, V. Dignum, and G. Picard, editors, *Engineering Societies in the Agents World X*,

volume 5881 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin Heidelberg, 2009.

[78] G. J. Myers and C. Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[79] D. Okouya, N. Fornara, and M. Colombetti. An infrastructure for the design and development of open interaction systems. In M. Cossentino, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Engineering Multi-Agent Systems*, volume 8245 of *Lecture Notes in Computer Science*, pages 215–234. Springer Berlin Heidelberg, 2013.

[80] L. Padgham, J. Thangarajah, and M. Winikoff. AUML protocols and code generation in the Prometheus Design Tool. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS'07, pages 270:1–270:2, New York, NY, USA, 2007. ACM.

[81] J. G. Quenum, S. Aknine, O. Shehory, and S. Honiden. Dynamic protocol selection in open and heterogeneous systems. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Hong Kong, China, 18-22 December 2006*, pages 333–341. IEEE Computer Society, 2006.

[82] J. G. Quenum, F. Ishikawa, and S. Honiden. Protocol selection alongside service selection and composition. In *2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*, pages 719–726. IEEE Computer Society, 2007.

[83] R. de Lemos, H. Giese, H. A. Müller, et al. Software engineering for self-adaptive systems: A second research roadmap. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2013.

[84] A. 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.

[85] G. Salvaneschi, C. Ghezzi, and M. Pradella. An analysis of language-level support for self-adaptive software. *ACM Trans. Auton. Adapt. Syst.*, 8(2):7:1–7:29, 2013.

[86] J. R. Searle. *Speech acts: An essay in the philosophy of language*, volume 626. Cambridge university press, 1969.

[87] M. P. Singh. Interaction-oriented programming: Concepts, theories, and results on commitment protocols. In *Proceedings of the 19th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, AI'06, pages 5–6, Berlin, Heidelberg, 2006. Springer-Verlag.

[88] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White. A multi-agent systems approach to autonomic computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, pages 464–471, Washington, DC, USA, 2004. IEEE Computer Society.

[89] S. Warwas and C. Hahn. The DSML4MAS development environment. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1379–1380, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[90] G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[91] D. Weyns and M. Georgeff. Self-adaptation using multiagent systems. *Software, IEEE*, 27(1):86–91, 2010.

[92] Y. Brun, G. Di Marzo Serugendo, C. Gacek, et al. Engineering self-adaptive systems through feedback loops. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009.

[93] P. Yolum and M. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and Artificial Intelligence*, 42(1-3):227–253, 2004.

[94] F. Zambonelli, N. Bicocchi, G. Cabri, L. Leonardi, and M. Puviani. On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*, pages 108–113, 2011.

[95] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.