UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS INFORMÁTICOS

EJASON: A FRAMEWORK
FOR DISTRIBUTED AND
FAULT-TOLERANT
MULTI-AGENT SYSTEMS

PHD THESIS

ÁLVARO FERNÁNDEZ DÍAZ

April 2018

DEPARTAMENTO DE LENGUAJES Y SISTEMAS
INFORMÁTICOS E INGENIERÍA DEL SOFTWARE

—

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS INFORMÁTICOS

EJASON: A FRAMEWORK
FOR DISTRIBUTED AND FAULT-TOLERANT
MULTI-AGENT SYSTEMS

Submitted in partial fulfilment of the requirements for the degree of Doctor en
Software y Sistemas

Author:

ÁLVARO FERNÁNDEZ DÍAZ

European Master in Computational Logic
Universidad Politécnica de Madrid

Advisors:

PH.D. CLARA BENAC EARLE

Profesor Contratado Doctor
Universidad Politécnica de Madrid

PH.D. LARS-ÅKE FREDLUND

Profesor Contratado Doctor
Universidad Politécnica de Madrid

2018

To my Grandma Juana, let this step towards my maturity be a tribute to all her love and care to me from my childhood through her return to innocence. I love you abuela, to me “you will never grow old”.

Dedicada a mi abuela Juana, que este paso en mi madurez sirva de homenaje a todo su amor y cariño hacia mí desde mi infancia hasta su retorno a la inocencia. Te quiero abuela, para mí “tú nunca serás vieja”.

Thesis Committee:

- **Dr. Juan José Moreno Navarro**
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid
- **Dr. Rafael Heitor Bordini**
Faculdade de Informática
Pontifícia Universidade Católica de Rio Grande do Sul
- **Dr. Javier Carbó Rubiera**
Escuela Politécnica Superior
Universidad Carlos III de Madrid
- **Dr. Laura Milagros Castro Souto**
Facultad de Informática
Universidade da Corua
- **Dr. Susana Muñoz Hernández**
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid
- **Dr. Francisco Javier López Fraguas**
Facultad de Informática
Universidad Complutense de Madrid
- **Dr. María Elena Gómez Martínez**
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contents

List of Figures	v
List of Tables	vi
Summary	ix
Resumen	xi
Acknowledgements	xiii
Agradecimientos	xv
1 INTRODUCTION	1
1.1 Contributions of the Thesis	5
1.1.1 Summary of Contributions	6
1.1.2 List of publications	8
1.2 Fault-Tolerance for Multi-Agent Systems	9
1.3 Formal semantics of agent-oriented programming languages	11
2 BACKGROUND	13
2.1 The programming language Jason	14
2.1.1 Origins	14
2.1.2 Syntax and Informal Semantics of Jason	20
2.1.3 Formal Semantics: Jason's Grammar	32
2.2 The programming language Erlang	33
2.2.1 Erlang as a functional programming language	33
2.2.2 Concurrency and distribution	34
2.2.3 Fault tolerance	35

3	GOVERNING JASON AGENTS	39
3.1	Synchronisation mechanisms	39
3.1.1	Jason: Atomic Plans	41
3.1.2	eJason: Critical Sections	44
3.2	Intention selection function	46
3.2.1	Jason: customisation of the intention selection function	49
3.2.2	eJason: consecutive evaluation and execution of a plan	50
3.3	Achievement Goals	51
3.3.1	Jason: maintaining a record of intentions to retry	52
3.3.2	eJason: requeuing not applicable achievement goals	53
3.4	Test Goals	55
3.4.1	Jason: implementing busy waiting	55
3.4.2	eJason: providing different operators	56
3.5	The Reasoning Cycle of eJason agents	57
4	DISTRIBUTION AND FAULT TOLERANCE	59
4.1	Distribution	60
4.1.1	Distribution Schema	60
4.1.2	Connectivity	61
4.1.3	Multi-Agent System Dynamics	62
4.1.4	Improvements over Jason	63
4.1.5	Distribution API	65
4.2	Fault Tolerance	71
4.2.1	Monitoring Mechanism	72
4.2.2	Supervision Mechanism	76
5	FORMAL SEMANTICS	87
5.1	Agent Evolution: Reasoning Cycle	88
5.1.1	Accepted Programs	88
5.1.2	Agent Configurations	89
5.1.3	Notational Conventions and Auxiliary Functions	90
5.1.4	Transitions of the Reasoning Cycle	94
5.2	Distribution and Fault Tolerance	108
5.2.1	eJason Multi-Agent Systems	108
5.2.2	Relevant Internal Actions	112
5.2.3	Monitoring Notifications	119
5.2.4	Supervision	122
5.3	System Properties	130

6	INTERPRETING eJASON PROGRAMS	133
6.1	The Interpreter of eJason	134
6.1.1	High-Level Design	134
6.1.2	Scope of the implementation	135
6.1.3	Overview of the implementation	138
6.1.4	Installing and Running eJason	141
6.2	Examples	143
6.2.1	Peano Arithmetic	143
6.2.2	Token Ring	148
6.2.3	Fault Tolerant Client-Server System	154
6.2.4	Overcoming Race Conditions	159
6.2.5	Overcoming Divergence	161
7	CONCLUSIONS	163
	Bibliography	169

List of Figures

Figure 2.1:	Preformed plans executed by a commuter to achieve its goal G_1 in a) an uneventful day and b) in the presence of a disruption in line 1 (disabling the execution of plan $P_{1.2}$).	18
Figure 2.2:	Possible state transitions within a reasoning cycle in Jason. . .	29
Figure 2.3:	An Erlang multi-node system	34
Figure 2.4:	Sample Erlang Supervision Tree	38
Figure 3.1:	Jason plan for the file counter	40
Figure 3.2:	Possible interleavings for the counter update example.	42
Figure 3.3:	File counter with an atomic plan	42
Figure 3.4:	Sample use of the Jason atomic plans	43
Figure 3.5:	Possible interleavings for the counter update example using synchronisation mechanisms.	43
Figure 3.6:	File counter with eJason critical sections	45
Figure 3.7:	Jason code for the client agents	46
Figure 3.8:	Jason code for the file server agent	47
Figure 3.9:	Jason code for the file server agent using critical sections . . .	50
Figure 3.10:	Final Jason code for the client agents	57
Figure 3.11:	Possible state transitions within a reasoning cycle in eJason. .	58
Figure 4.1:	eJason multi-agent system for a simplified e-commerce service	61
Figure 4.2:	Connection of Multi-Agent Systems	67
Figure 4.3:	Sample eJason supervision tree	76
Figure 5.1:	Labelled state transitions within a reasoning cycle. Labels omitted for readability: a) <u>achvG11</u> , <u>achvG12</u> , <u>testG12</u> , <u>testG13</u> ; b) <u>action</u> , <u>testG11</u> , <u>addBel</u> , <u>delBel</u> , <u>critExecInt2</u> ; c) <u>critAddG12</u> , <u>critAction</u> , <u>critAddBel</u> , <u>critDelBel</u> ; d) <u>critAddG11</u> ; e) <u>clrInt1</u> , <u>clrInt3</u> ; f) <u>critExecInt1</u> ; g) <u>critAddIM</u> ; h) <u>critRel3</u>	94

Figure 6.1: High Level Design of the eJason interpreter	135
Figure 6.2: Inference rules of the Peano agent	144
Figure 6.3: Inference rules implementing number conversion	145
Figure 6.4: Jason plan that runs the different Peano addition tests	146
Figure 6.5: Jason plan that runs the different Peano multiplication tests .	146
Figure 6.6: Complete Jason code of the worker agents	151
Figure 6.7: Complete Jason code for the tokenRing agent	152
Figure 6.8: Variations to the Worker agents for the second test batch . . .	153
Figure 6.9: Code for the server agent	156
Figure 6.10: Code for the client agents	157
Figure 6.11: Code for the supervisor agent	159
Figure 6.12: Sample business layer for the server agent	160
Figure 6.13: Sample business layer for the client agents	160

List of Tables

Table 6.1:	Execution time of the Peano addition test batch ($X=10$) . . .	147
Table 6.2:	Execution time of the Peano multiplication test batch ($X=10$)	148
Table 6.3:	Execution time of the Token Ring test batch ($W=50$)	153
Table 6.4:	Execution time of the second Token Ring test batch	154

Summary

This thesis elaborates on the synergies between the agent-oriented programming language Jason and the general-purpose programming language Erlang. These synergies are leveraged by the novel *Distributed and Fault-Tolerant Multi-Agent Systems Development Framework* eJason, which is the core contribution of this thesis work.

The syntax of the programming language Jason incorporates high-level abstractions that enable the programmer to efficiently manage the different facets of software agents in general, and, more specifically, the building blocks of the Belief-Desire-Intention architecture. Moreover, the formal semantics of this programming language are available and well-studied. However, Jason lacks native support for the distribution of multi-agent systems across several computers and built-in fault-tolerance mechanisms for the detection and recovery of the system from faults that incapacitate an agent.

The programming language Erlang provides an excellent support for the development of distributed and highly-concurrent software. Erlang provides the programmer with constructs to implement fault-detection and fault-recovery techniques that are fully-supported by the Erlang runtime system. However, Erlang lacks high-level abstractions for the development of multi-agent systems as well as any constructs related to building blocks of the Belief-Desire-Intention software architecture.

Building on the strengths of Jason and Erlang, while removing their weaknesses listed above, the multi-agent system development framework eJason brings-in several relevant contributions.

The first of the contributions of eJason includes a set of language extensions to Jason that enhance the control of the programmer on the behaviour of the Jason agents. These extensions tackle the negative impact of several race conditions

inherent to the concurrent execution of the set of intentions within an agent's mental state.

The second contribution of eJason is a distribution schema for the Jason agents, which enables their distribution across several computers. This distribution schema is programmer-transparent and is seamlessly integrated into the syntax of Jason.

The third contribution of eJason is embodied by two fault-tolerance mechanisms that are inspired by those of Erlang but which improve them in several ways. The monitoring mechanism of eJason enables the detection of different agent faults while the supervision mechanism enables the automatic execution of different fault recovery actions.

The fourth contribution is a formalisation of the semantics of eJason. This formalisation builds on that of Jason, while providing a wider perspective that now includes the whole multi-agent system.

Finally, the last contribution of eJason is a free and open-source implementation of an interpreter of eJason. This interpreter enables the validation of the benefits introduced by eJason, as is described by several examples and use cases throughout the thesis document.

Resumen

La presente tesis doctoral se fundamenta en el aprovechamiento de las sinergias existentes entre el lenguaje de programación para sistemas de agentes Jason y el lenguaje de programación de propósito general Erlang. El “Entorno de Trabajo para el Desarrollo de Sistemas Multi-Agente Distribuidos y Tolerantes a Fallos” conocido como eJason, y que representa la contribución novedosa principal de este proyecto de tesis doctoral, realiza el máximo aprovechamiento de las mencionadas sinergias.

El lenguaje de programación Jason incorpora en su sintaxis diversas abstracciones de alto nivel que permiten realizar un uso eficiente, por parte del programador, de los diferentes aspectos que caracterizan a los agentes software en general. De manera más específica, estas abstracciones facilitan el manejo de los componentes esenciales de la arquitectura software conocida como BDI (de sus siglas en inglés Beliefs, Desires, Intentions), es decir, del conjunto de creencias, deseos e intenciones de un agente. La semántica formal de Jason se encuentra disponible y ha sido ampliamente estudiada, lo que proporciona una descripción precisa del comportamiento esperado de los programas desarrollados con Jason. Sin embargo, Jason carece de soporte nativo para la distribución de los distintos agentes en diferentes computadores. Este lenguaje carece, además, de mecanismos nativos para la detección y resolución de aquellos fallos que, aún incapacitando a un único agente, afecten negativamente al conjunto del sistema.

El lenguaje de programación Erlang proporciona un soporte excelente al desarrollo de software distribuido y altamente concurrente. Erlang le proporciona al programador directivas para la implementación de mecanismos de detección y resolución de fallos que cuentan con el soporte completo del entorno de ejecución. El lenguaje de programación Erlang carece de abstracciones de alto nivel para el desarrollo de sistemas multi-agente así como cualquier mecanismo nativo para el manejo de los componentes de la arquitectura BDI.

El entorno de desarrollo eJason, aprovechando las fortalezas de Jason y Erlang, y eliminando sus debilidades y carencias anteriormente mencionadas, proporciona y se fundamenta en distintas contribuciones relevantes.

La primera contribución de eJason se encuentra representada por una serie de extensiones al lenguaje de programación Jason cuyo objetivo es la mejora en el control del comportamiento de los agentes por parte del programador. Estas extensiones minimizan o eliminan los efectos adversos de las distintas condiciones de carrera derivadas de la multiplicidad de focos de atención permitida por la semántica de Jason.

La segunda contribución de eJason es la provisión de un esquema de distribución propio para los agentes Jason que posibilita su distribución en diferentes computadores. Este esquema de distribución es transparente para el programador y su integración en la sintaxis de Jason no es disruptiva.

La tercera contribución de eJason está representada por dos mecanismos de tolerancia a fallos inspirados por aquéllos de Erlang pero que los superan en diversos aspectos. El mecanismo de monitorización de eJason permite la detección de diferentes fallos, mientras que el mecanismo de supervisión posibilita la corrección automatizada de diversos fallos.

La cuarta contribución de eJason es una formalización de su semántica. Esta formalización extiende la de Jason proporcionando, a su vez, una perspectiva más amplia que permite especificar no sólo el comportamiento de un único agente, sino aquél del sistema multi-agente en su totalidad.

Finalmente, la contribución última de eJason es un intérprete gratuito y de código abierto para este lenguaje. Este intérprete permite validar los beneficios introducidos por eJason, como se describe en diversos ejemplos y casos de uso a lo largo del documento de tesis.

Acknowledgements

Although it is virtually impossible to express all the gratitude that I owe to the different people that have contributed to this achievement in one way or another, I will try. Firstly, I would like to thank my advisors Dr. Clara Benac Earle and Dr. Lars-Åke Fredlund for the chance to undertake this doctoral project. They have shown themselves not only available and resourceful, but also friendly and supportive to me all the steps of the path. Their constant assistance provided me with all the stepping stones necessary to grow as a researcher and an IT professional since my very graduation.

Secondly, I would like to thank all my colleagues from Babel Research Group (UPM) with whom I have shared a great deal of my PhD studentship: Juanjo, Julio, Ángel, Susana, Pablo, Guillem, Álvaro, Emilio, Salvador, Ricardo and two more astounding people that deserve their own lines in this section.

The first of them is Ana María Fernández, who was the first member to make me feel welcome during my first weeks at Babel. Since her very first *“Switch the lights on, the screen is gonna blind you badly!”*, I knew we were in for a long, enjoyable companionship. Her determination and resolve were always an inspiration in moments of doubt.

The second person that I want to thank is Dr. Elena Gómez with whom I shared the second half of my stay at Babel. Her example has always been a subject for my admiration and her teachings, within both the professional and personal dimension, are uncountable and invaluable. Her contribution to my development as a professional and individual is, in many facets, key to my (life) career.

Besides, I also want to thank the members of my Thesis Committee for their time and availability.

Out of the research community, I would like to thank my family and friends for their encouragement and, at times much needed, words of support. They have borne part of the burden, specially when it took the best of my mood. I want to thank also my colleagues at the Spanish Central Bank for their frequent inquiries: "*when are you going to defend your Thesis?*", they reinforced my determination to be able to provide them with a date.

I also want to thank my former boss Carmen Clares (now happily retired) for her encouragement to focus on my thesis when overwhelmed by my day to day responsibilities. Her question "*what about your stuff?*" helped me keep on track of the hard work to achieve my doctorate.

I want to thank Dr. Óscar Sánchez for his constant pressure towards the completion of this thesis. Almost every afternoon from the last two years ended with an "*will you work on your thesis this afternoon?*" to be followed by a similar "*did you work on your thesis?*" the following morning. His opinion on what contents a thesis document shall mandatorily address encouraged me to write some additional paragraphs.

Last but not least, I want to thank my sister Sara and my brother-in-law Sergio for their constant support and understanding on the implications of writing a thesis dissertation. Finally, to my parents, I just want them to know that, as is the case since kindergarten, all my academic steps are also theirs.

Agradecimientos

Aunque resulta completamente imposible expresar toda la gratitud que albergo hacia las diferentes personas que, de un modo u otro, han contribuido a que alcance esta meta, lo intentaré. En primer lugar, quiero agradecerles a mis supervisores de tesis, los doctores Clara Benac Earle y Lars-Åke Fredlund la oportunidad que me brindaron de emprender mi proyecto doctoral. Durante todo el recorrido, se han mostrado no sólo disponibles y plenos de recursos, sino amigables y comprensivos hacia mí. Su apoyo constante me ha proporcionado todas las herramientas necesarias para crecer como investigador y profesional de las tecnologías de la información desde el mismo momento de mi graduación.

En segundo lugar, quiero agradecerles a mis compañeros del Grupo de Investigación Babel (UPM) el tiempo que hemos colaborado durante el desarrollo de este proyecto: Juanjo, Julio, Ángel, Susana, Pablo, Guillem, Álvaro, Emilio, Salvador, Ricardo y, de manera muy especial, dos personas que, por su brillantez excepcional, merecen su propia mención.

La primera de ellas es Ana María Fernández, quien fue la primera persona del grupo de investigación que me hizo sentir parte de él durante las primeras semanas de mi incorporación. Desde su primer *“Enciende las luces, ¡te vas a quedar ciego con la pantalla!”* supe que el futuro nos regalaría una alargada y satisfactoria trayectoria común. Su determinación y claridad de visión siempre fueron una fuente de inspiración para mí en momentos de duda.

La segunda persona a la que quiero agradecerle su contribución es la doctora Elena Gómez, con quien compartí buena parte de la segunda mitad de mi trayectoria en Babel. Su ejemplo siempre ha merecido mi mayor admiración, tanto en el ámbito profesional como el personal. De ella he aprendido incontables e inestimables lecciones que, incorporadas durante mi desarrollo personal y profesional, resultan determinantes en diversos aspectos de mi carrera.

Por otro lado, quisiera agradecerles a todos y cada uno de los miembros de mi Tribunal de Defensa su tiempo y disponibilidad.

Ajenos a la comunidad investigadora, quisiera agradecerles a mi familia y amigos su apoyo y, a veces muy necesarias, palabras de apoyo. Ellos han soportado parte de la carga, especialmente cuando ésta se apoderaba de mi estado de humor. También quiero agradecerles a mis compañeros del Banco de España su habitual pregunta: “¿Cuándo vas a defender la tesis?”, ya que han redoblado mi determinación de poder proporcionarles una respuesta.

Entre estos compañeros, quisiera destacar a mi anterior responsable y mentora Carmen Clares (felizmente jubilada) su frecuente consejo de centrarme en finalizar esta tesis cuando, desbordado por la responsabilidad, acudía a ella en busca de guía. Su pregunta “¿Qué hay de lo tuyo?” me ha ayudado a continuar con el duro esfuerzo necesario para finalizar mi doctorado.

Quiero agradecerle también al doctor Óscar Sánchez su presión constante para que terminara esta tesis. En los últimos dos años, la mayoría de las tardes concluían con un “¿Te vas a poner hoy con la tesis?” seguido de un “¿Te pusiste con la tesis?” a la mañana siguiente. Su opinión con respecto a qué debe contener obligatoriamente una tesis ha motivado la inclusión de algunos párrafos a la misma.

En penúltimo lugar, pero no menos importante, quiero agradecerles a mi hermana Sara y a mi cuñado Sergio su apoyo constante y su comprensión sobre las implicaciones de llevar a cabo una tesis doctoral. Finalmente, a mis padres tan sólo recordarles que, como ocurre desde mi etapa preescolar, todos mis pasos académicos también son sus pasos.

Chapter 1

INTRODUCTION

In the latest years, the widespread use of devices with increasing computing capabilities accelerates the switch from a monolithic, single-threaded computation paradigm to a distributed, highly-concurrent one. This switch to distributed computing requires of the development of programming languages that provide the proper mechanisms and high-level abstractions to support the implementation of this kind of software. Among the different existing programming language paradigms¹ for distributed systems there is an increasing interest in those that address the development of multi-agent systems (MAS). In a nutshell, these systems are pieces of software whose architecture is composed by a series of entities with computational capabilities, named agents, that, in most cases, interact among them. MAS and the well-known actor model [Agha and Hewitt, 1987] have many characteristics in common. The key difference is that agents normally impose extra requirements upon the actors, typically a rational and motivational component such as the Belief-Desire-Intention architecture [Rao and Georgeff, 1995, Wooldridge, 2000].

Although not widely known outside the artificial intelligence community, the belief-desire-intention software model (BDI for short) for programming multi-agent systems has a number of interesting features. Basically it provides a higher-level software architecture for separating a number of program concerns:

- *beliefs* are the facts an agent (thinks it) knows
- *desires* are the goals an agent wants to realise, and
- *intentions* represents the strategies the agent has chosen to realise its desire; these are often called *plans*

¹Roughly, the common set of programming styles and language constructs by which the programming languages can be classified.

Moreover, as BDI systems are generally reactive multi-agent systems, *events* (occurrences of actions outside of the control of the agent itself, but which potentially impact the agent) are key.

Although much of the BDI literature is inspired by the philosophical aspects of artificial intelligence, the concepts above can be reinterpreted to speak about a particular software architecture for multi-process systems. This interpretation is adopted in this thesis work.

As a matter of example, consider a classical simple communication protocol using acknowledgement messages (such as the alternating bit protocol or TCP/IP) for transmitting a sequence of bytes, with a sender and receiver agent (process). What are the beliefs, desires, intentions and events of the agents? Clearly the overall *desire* of both the sender and receiver agents is that the sequence of bytes gets correctly transferred and received at the receiver agent. *Beliefs* concern basic facts, as well as more complex facts concerning protocol negotiation. For instance, a basic belief of the sender is that it has sent some subsequence of bytes on the communication channel. A more complex belief is that the sender “knows” that the receiver agent has correctly received a sent subsequence of bytes; this knowledge comes from observing the *event* that an “acknowledgement” message was received from the receiver agent. Similarly a receiver agent may possess the more complex belief that it “knows” that the sender agent knows that the receiver agent has correctly received a subsequence of bytes (usually this knowledge comes from having received a later subsequence of bytes as an event). The *intentions*, or *plans*, describe the concrete protocol, i.e., for an agent, when a new event occurs, and given a certain set of beliefs, what is the next set of actions of an agent. An action may be a communication or that the agent updates its beliefs. As an example, the sender agent may send a new sequence of bytes upon receiving an acknowledgement message (event) from the receiver agent, and also update its beliefs to reflect the fact that it knows that the receiver agent has correctly received the previous part of the message.

Compared to a more traditional way of implementing a communication protocol, here the main difference is that the knowledge of the agents (processes) is structured and made explicit in a declarative fashion.

There are numerous different BDI programming language systems available; some of the more popular ones are 2-APL [Dastani, 2008], 3-APL [Hindriks et al., 1999], Jason [Bordini et al., 2007] and

GOAL [Hindriks et al., 2001]. As a core part, most of the programming language systems use a Prolog inference engine to resolve basic queries regarding beliefs, to store beliefs, and to select plans. Although all these language systems do provide support for multi-agent systems, i.e., adequate communication facilities, it is fair to say that the major implementation effort for systems like e.g. Goal has been on improving basic inference speed by utilising a high-performing Prolog implementation. Rather less attention has been spent on improving multi-agent aspects, resulting in multi-agent system implementations that do not scale to more than at most a few thousand concurrent agents.

Apart from the implementation of agents that follow a specific architecture, such as BDI, the more difficult challenges faced by the multi-agent systems community, i.e., how to develop scalable and robust, fault-tolerant systems, are the same fundamental challenges that any concurrent and distributed system faces. Consequently, the agent-oriented programming languages provide mechanisms to address these issues, typically borrowing from mainstream frameworks for developing distributed systems. As a result, new programming languages and tools capable of supporting complex MAS development have appeared on the last decade. Most of these languages (and all the aforementioned ones) are implemented using the widespread Java programming language [Gosling, 2000], hence leveraging the capabilities of the Java Virtual Machine. The performance and scalability of the systems implemented with these programming languages highly depend then on those of Java.

Finding the BDI software architecture interesting, while finding wide room for improvement of the support for large agent systems, inspired the hypothesis that the agent-development community will benefit from the integration of a high-performing runtime system into an agent-development framework. The widespread usage of a general-purpose language, such as Java, as the basis of a framework for the development of distributed and highly-concurrent software systems, instead of a programming language well-known for its excellent capabilities on these matters, such as Erlang [Armstrong, 2007, Cesarini and Thompson, 2009], motivated the work presented in [Fernández-Díaz et al., 2010]. This paper shows how the Erlang implementation of a simple, whereas communication intensive, agent system (i.e. the extended Contract Net Protocol [Aknine et al., 2004]) outperforms the implementation of the very same system in Java. The average performance improvement is of four orders of magnitude. Hence supporting the hypothesis that multi-agent systems may benefit from the capabilities provided by the Erlang programming language. This hypothesis represents the seed of the present thesis work.

Erlang is a programming language best-suited for the development of highly-concurrent, robust, soft real-time systems that are also highly-scalable to make an efficient use of the underlying multiprocessor architecture. Any Erlang system is typically composed by hundreds or thousands of lightweight processes. These processes are handled by Erlang's Virtual Machine (BEAM) and are hence created and scheduled with a very small cost of computational resources, specially when it is compared with the cost of traditional Operating System processes. These processes do not share memory space and communicate with each other using asynchronous message-passing techniques that implement mailboxes. The processes that compose an Erlang system can be distributed across several computers without a penalty to their communication abilities. A distributed Erlang system is composed by several Erlang nodes, where each of them hosts several Erlang processes. Erlang can then be used to implement efficient agents on top of lightweight processes which are able to interact using Erlang message-passing capabilities. Moreover, such Erlang agents could be distributed across several machines without an impact on their ability to communicate.

Additional useful features of Erlang are its fault-tolerance techniques. These include the ability of processes to monitor the execution state of other processes, in order to react to failures in them and easily contain their propagation, e.g., avoiding deadlocks when a process expects some message from another process that has stopped being executed. Another fault-tolerance technique introduced by Erlang is the supervisor behaviour, a design pattern intended to orchestrate the Erlang processes in an orderly manner so that they are started and stopped gracefully, or even restarted if they crash. The Erlang programmer can leverage these techniques for fault detection and recovery to design software systems where crashing processes, while still undesirable, do not represent an irrecoverable vulnerability. Actually, Erlang promotes a philosophy known as 'let it crash', where processes are meant to either succeed or crash. Putting the aforementioned techniques into place, the programmer can easily recover then from a process crash.

Note that Erlang follows the actor model and provides excellent support for concurrency, distribution and fault tolerance. However, Erlang lacks some of the concepts, like the Belief-Desire-Intention architecture, that are relevant to the development of MAS. Therefore, in order to support the aforementioned hypothesis, several alternatives were considered. One alternative implies the implementation of an agent development framework anew, which would leverage the capabilities of Erlang. This alternative would require an implementation that added support for

beliefs and plans to Erlang itself. Hence providing a set of Erlang libraries (modules in Erlang terminology) and best practices in order to implement multi-agent systems in Erlang. A similar approach is followed by the agent implementation platform eXAT [Stefano and Santoro, 2003, Stefano and Santoro, 2004]. Another alternative would be developing a new programming language that provided the necessary constructs to implement multi-agent systems. The third alternative is the reimplementing of the core functionality of a successful and already-existing programming language in Erlang. In a nutshell, the first and second alternatives consist in the introduction of yet another agent-oriented implementation framework. Such alternatives shall attract enough interest from the community as to justify the switch from a different agent-oriented development technology. On the contrary, the third alternative does not require any switch for the agent development community, provided that the programming language selected is already successful. Moreover, by maintaining the well-defined syntax and semantics of a mature programming language, its inherent benefits carry over to the new implementation. Hence, the third alternative has been adopted.

One remarkable member of the set of agent-development technologies is the aforementioned programming language Jason. This language successfully addresses, among others, the communication mechanisms that enable the interaction between agents as well as the high-level abstractions that enable the programmers to implement complex behaviours for the agents, whose programs can be executed by an interpreter written in Java. The core contribution of this thesis is embodied by an interpreter of Jason in Erlang. However the contributions of the thesis are not limited to that, as Jason itself has been provided with extended capabilities for distribution and fault-tolerance. These extensions, while inspired by the aforementioned Erlang distribution and fault-tolerance techniques, provide several relevant improvements over the Erlang ones, specially tailored for the development of multi-agent systems. Moreover, these extensions accommodate to both the BDI architecture implemented by Jason as well as the syntax of this programming language. The contributions of the thesis are summarised in the next section.

1.1 Contributions of the Thesis

The design and implementation of a Jason interpreter in Erlang, namely eJason, intends to provide a development framework for multi-agent systems that embodies a combination of the strengths of both programming languages in a synergic manner. This framework implements and extends the syntax and semantics of Jason and

leverages the excellent capabilities for distribution and fault-tolerance of the Erlang Virtual Machine. The achievement of a performance improvement in-line with that reported in the early stages of the research, summarised in [Aknine et al., 2004], is also a desirable feature. The next subsections provide a summary of the contributions of this thesis as well as a list of relevant publications in peer-reviewed international research conferences and journals that present a subset of the herein described contributions.

1.1.1 Summary of Contributions

As it is described throughout the present thesis document, the contributions of this work are far greater:

- **The Jason programming language is extended in order to enhance the control of the programmer on the behaviour of the agents in the system.** A thorough analysis of the syntax and semantics of the Jason programming language exposes some behaviours that, while allowed by its semantics, may be considered as vulnerabilities of the language. These vulnerabilities include:
 - a) Classical race conditions, that, uncontrolled, can result in the undesired behaviour of the agents. Jason provides language constructs that, while effectively tackling this problem, dramatically decrease the readability of the code. In contrast, eJason provides a more elegant and concise approach to deal with this problem.
 - b) Jason semantics implements a loose coupling between the execution time of a piece of code and the moment at which the guard that enables its execution is resolved. The consequence of this behaviour is the lack of guarantees on the truth value of the guard when the guarded code is actually executed. On the contrary, eJason provides a means for the programmer to implement a strong coupling of the two, allowing such programmer to specify which prefix (if not its entirety) of the guarded code must be executed right after the evaluation of the guard.
 - c) When a guarded code cannot be executed (because its guard cannot be validated), the Jason interpreter desists on its execution, requiring the programmer to elaborate cumbersome mechanisms to retry its execution later. On the contrary eJason allows the programmer to specify whether this code must be retried or desisted on with the introduction of a single operator.

These vulnerabilities are described in Chapter 3. This chapter also elaborates on their implications as well as how each interpreter tackles their related threats. In a nutshell, this chapter shows how Jason requires the programmer to implement complex mechanisms (or even re-implement parts of the interpreter itself using the Java programming language) to cope with the aforementioned vulnerabilities. In contrast, the extensions introduced by eJason deal with these issues in an elegant way.

For readability, in the remainder of the document, the term “Jason” is used stand-alone to refer to either the Jason programming language or the implementation of an interpreter for this language in Java, when its meaning is unambiguous on the context. Similarly, the term “eJason” refers to the extended version of the Jason programming language, as well as its Erlang interpreter.

- **eJason introduces language constructs to implement distributed and fault-tolerant multi-agent systems.** Jason does not provide language constructs to implement distributed systems. Instead, it allows the development of distributed multi-agent systems by interfacing with JADE [Bellifemine et al., 2007, Bellifemine et al., 2005]. This approach introduces several drawbacks. First of all, it exposes the fact that a programming language for the development of MAS (i.e. an inherently distributed and highly-concurrent software) lacks native mechanisms to enable the interaction of agents distributed across several machines. Besides, interfacing with a third-party software increases the complexity of the implementation as it requires the programmer to accommodate yet another technology to the design. On the other side, eJason introduces language constructs that enable an agent distribution that is transparent to the programmer. With regards to fault-tolerance, Jason focuses its efforts at a low level. It provides mechanisms to enable an agent to retry a procedure that fails at some step but lacks mechanisms to recover from the failure of the agent itself. The constructs introduced by eJason enable the programmer to implement mechanisms to detect and recover from failures on the agents that compose the system. Moreover, offering a wide variety of alternatives on how to react to such failures. While the fault-tolerant mechanisms of eJason are inspired in the ones provided by Erlang, they improve their capabilities in several aspects. All these mechanisms, along with recommendations on their use, are described in Chapter 4.
- **The formal semantics for eJason is available.** Despite its high value, a formalisation of the semantics of most agent-oriented programming languages

is not available. A brief description of formalisations available for these kind of languages is provided in Section 1.3. The formal semantics of Jason is described in [Bordini et al., 2007]. For clarity, the formalisation of eJason follows the same style. This formal semantics specifies all the language extensions as well as all the mechanisms for distribution and fault-tolerance introduced by eJason. Hence this formalisation provides the unambiguous description of the functionality introduced by eJason. The formal semantics of eJason is provided in Chapter 5.

- **An open-source interpreter for eJason has been implemented, evaluated and made available for download.** The contributions of this thesis work are not limited to the theoretical domain. The interpreter for eJason has been uploaded to an online software repository can be downloaded as open-source at:

git : //github.com/avalor/eJason.git

This interpreter makes it possible to develop fault-tolerant distributed systems fully in eJason. Apart from the source code, the high-level design of the implementation is provided in Chapter 6. This design helps understand the complexity of the solution as well as provides useful insights for programmers interested in the implementation of similar interpreters. Finally, this chapter introduces several use cases that help evaluate both the improvement of the performance with regards to Jason and the benefits of using eJason to implement distributed and fault-tolerant multi-agent systems.

1.1.2 List of publications

A subset of the contributions of this thesis has been already presented in international peer-reviewed research conferences and journals. The corresponding publications, in chronological order, are the following:

- **Implementing a multiagent negotiation protocol in Erlang** by Álvaro Fernández Díaz, Clara Benac Earle and Lars-Åke Fredlund in *Erlang Workshop 2010*. This article describes the experiments carried out in the early stages of the research. The results described in this article provide insights that encourage the hypothesis leading to the present thesis work.
- **eJason: An Implementation of Jason in Erlang** by Álvaro Fernández Díaz, Clara Benac Earle and Lars-Åke Fredlund in *Programming Multi-Agent*

Systems (ProMAS) 2012. This article describes the implementation of the first version of the eJason interpreter. Its contents represent a previous version of the contents in Section 6.1. The target audience is the multi-agent systems community.

- **Erlang as an implementation platform for BDI languages** by Álvaro Fernández Díaz, Clara Benac Earle and Lars-Åke Fredlund in *Erlang Workshop 2012*. This article describes at a low level the implementation of the building blocks of every BDI architecture. Its contents are also described in Section 6.1. The target audience is the Erlang community.
- **Adding distribution and fault tolerance to Jason** by Álvaro Fernández Díaz, Clara Benac Earle and Lars-Åke Fredlund in *AGERE! 2012 (co-located with SPLASH 2012)*. This paper briefly introduces the distribution and fault-tolerance mechanisms of eJason described in Chapter 4.
- **Enhancing control over Jason agents** by Álvaro Fernández Díaz, Clara Benac Earle and Lars-Åke Fredlund in *Jornadas sobre Programación y lenguajes (PROLE) 2014*. This article summarises the contents described in Chapter 3.
- **Adding distribution and fault tolerance to Jason** by Álvaro Fernández Díaz, Clara Benac Earle and Lars-Åke Fredlund in *Journal of Science of Computer Programming V.98 2015*. This journal article describes virtually all of the contents in Chapter 4.

In order to better evaluate some of the contributions of this thesis work, the next two sections elaborate on works related to this one. Section 1.2 provides a brief description of different fault-tolerance mechanisms implemented by several MAS development frameworks. This description provides a broad-grained classification for these techniques. Section 1.3 provides a review on how the formal semantics for different agent-oriented programming languages have been addressed. Besides, it helps situate the wider perspective provided by the formal semantics for eJason.

1.2 Fault-Tolerance for Multi-Agent Systems

Several existing techniques try to facilitate the development of robust multi-agent systems. According to the classification introduced in [Klein et al., 2003], these techniques follow either a *survivalist* or *citizen* approach.

A survivalist approach implies that the agents are designed to preserve their own existence. Survivalist agents inspect their own state and handle the possible exceptions that could arise, applying the corresponding corrective measures. The plan failure mechanism of Jason [Bordini et al., 2007] is an example of a survivalist approach. The replication of agents [Kumar and Cohen, 2000, Ductor et al., 2011], i.e. maintaining different copies of the same agent in the system, is a technique that also follows the survivalist approach. In active replication techniques the different replicas of an agent carry out the same computations. Then a single output is determined using e.g. a proxy [Fedoruk and Deters, 2002] that implements some aggregation function or voting system. Passive replication techniques maintain all but one replicas dormant. A dormant replica takes over the responsibilities of the active replica if a failure of the latter occurs. The DARX framework [Marin et al., 2001] allows the use of both active and passive replication techniques. The use of survivalist techniques requires the programmer to implement complex coordination and exception handling behaviours. Moreover, the replication of agents can consume an excessive amount of resources if not used carefully.

On the contrary, the citizen approach implies that the agents rely on different entities (e.g. system services or other agents) in order to handle different exceptions and agent deaths. Citizen approaches often require the existence of sentinels [Klein and Dellarocas, 1999, Haegg, 1997]. The sentinels are agents or services that monitor the behaviour of the different agents in order to take the appropriate corrective measures in the presence of faults. Sentinel agents require a description of the expected behaviour of the agents that they monitor. The sentinels in [Haegg, 1997] infer the behaviour of the agents by listening to their broadcast messages. A sentinel can even alter (repair) a message sent between two other agents if it is considered wrong. Conversely, each agent in [Klein and Dellarocas, 1999] possesses a normative description of its own behaviour. The agent uses this description to create a sentinel that monitors its behaviour.

The HADES System [Olsen et al., 2008] follows a hybrid (survivalist and citizen) approach inspired by biology. The agents in HADES actively replicate themselves in a way that seeks to preserve the equilibrium of the system. These agents monitor their own behaviour and repair, or even kill, themselves if necessary. The agents acknowledge their “citizenship” to the system, hence monitoring the agents in their environment. If an agent identifies some persistent anomaly they entice the malfunctioning agents to kill themselves. New replicas of these self-killed agents are then created to replace them.

The platform SyMPA [Suna, 2005] uses replication to maintain the functionality of its central system. Besides, the state of individual agents is preserved by writing it on the disk periodically. Similarly, the platform JADE [Bellifemine et al., 2007] introduces a Main Container Replication Service (MCRS) to replicate the main containers that compose an agent platform. The MCRS appoints a master main container, while the other main containers are replicas for backup purposes. The master container stores the information of the Directory Facilitator in a database. In the case of a failure of the master main container, a backup replica takes over that information from the database and replaces the failed one.

Note that, as described in Chapter 4, the fault-tolerance mechanisms introduced by eJason follow a citizen approach.

1.3 Formal semantics of agent-oriented programming languages

In spite of the wide variety of agent programming languages and platforms, there is relatively few work available on the formalisation of their semantics. Among the different formal semantics available, there can be found those for the programming language 3APL [D’Inverno et al., 2000] and the architecture dMars [D’Inverno et al., 2004]. They provide specifications using the Z Specification Language with the aim of facilitating the reduction of such specifications into implementations of an interpreter or a deployed system, respectively. A more recent work included in [Sardina and Padgham, 2011] provides a transition-based operational semantics for the CANPlan (Conceptual Agent Notation) programming language, with a special focus on the semantics of its declarative goals and its failure handling and planning mechanisms. Similarly, the works in [van Riemsdijk et al., 2005] and [Morandini et al., 2009] propose specifications for the concepts of declarative goals and goal models, respectively, in multi-agent environments. These formal semantics are not tailored to any specific agent programming language or platform.

The structural operational semantics for AGENTSPEAK and its interpreter Jason can be found in [Vieira et al., 2007] and [Bordini et al., 2007], respectively. They specify the transition rules that define the execution of the different phases of an agent’s reasoning cycle. Besides, they provide semantics for the speech-act based communication between agents as well as for the BDI logical modalities in AGENTSPEAK programs. As eJason is an extension to Jason, the work presented in this document is built on top of these specifications. While all the mentioned works are

given from an agent perspective (i.e. focusing on different aspects of the agent building blocks and agent communication), the formal semantics for eJason provided in Chapter 5 follows a multi-agent system perspective (i.e. focuses on the evolution of the system with regards to both its distribution and fault-tolerance mechanisms).

Chapter 2

BACKGROUND

This chapter introduces the programming languages that form the basis of eJason. The agent-oriented programming language Jason provides the syntactical and semantical core of eJason, as the latter can be considered an extension of the former. The general-purpose programming language Erlang provides the execution layer of eJason, while its mechanisms for distribution and fault-tolerance inspire similar extensions to Jason implemented by eJason.

Section 2.1 provides a thorough description of the philosophy behind Jason as well as the syntax of the language, which fully applies to eJason as well. This section begins with a review of the relevant literature which traces the origins of Jason to the enunciation of the classical Belief-Desire-Intention model of rational agency. This review elaborates on how the evolution of different technologies and theories led to the implementation of Jason as an agent-oriented programming language. For the sake of clarity, the description of the different predecessors of Jason is illustrated with simple, yet novel, examples. Next, an analysis of the syntax, along with an informal semantics, of the main language constructs of Jason is provided.

Section 2.2 describes the main features of the Erlang programming language that are leveraged by eJason. This section is more brief than the previous one, as it focuses on the elements of Erlang that have inspired the extensions to Jason. This description highlights the strengths of the Erlang mechanisms for distribution and fault-tolerance. These mechanisms are integrated seamlessly into the syntax of eJason (i.e. adapting the syntax of Jason) in order to facilitate their use by the Jason programmer.

2.1 The programming language Jason

This section describes the programming language Jason for programming Multi-Agent Systems.

2.1.1 Origins

The origins of the Jason programming language can be traced back to the enunciation of the Belief-Desire-Intention (BDI) model of rational agency [Bratman, 1987]. In the following, a brief description of the BDI model is given.

The Belief-Desire-Intention Model of Rational Agency

The BDI agent model is based upon the model of human behaviour, specifically on the role that the intentions play in order to determine such behaviour. This model derives from two related theories about the concept of intention. The *intentional stance* [Dennett, 1989] establishes a level of abstraction that enables the characterisation of the behaviour of an entity in the terms of its mental attitudes, i.e., its beliefs, its desires and its intentions (described below). The theory of *practical reasoning* [Bratman, 1987], introduces a psychological framework to describe how the aforementioned attitudes determine the actions that a rational agent takes. The postulates of these theories crystallised into the BDI architecture [Bratman et al., 1988], that lays the foundations for the development of software systems populated by rational agents.

The consideration that certain software system adheres to the BDI model implies that some of the computational entities (namely the agents) composing such system possess a so-called *mental state*. The mental state of an agent is defined by the following three building blocks of the motivational stance:

- **Beliefs:** the agent's beliefs represent the information that the agent possesses about the world (i.e. about its environment or even about the agent itself). For instance, a possible belief may be represented by a variable *Door1* with value *open*, meaning that some physical door is open. Note that there is no requirement about the accuracy of this information, as it represents what an agent believes.
- **Desires:** the agent's desires are the situations or states that the agent might like to bring about (frequently referred to as goals in the related literature, as their semantics is virtually the same). These desires just represent a set of

options for the agent, but the fact that an agent possesses some desire does not imply that such agent will act in order to fulfil it. However, the agent's desires influence the agent behaviour as the agent's intentions are selected from them via the deliberation process described below. Note that the different desires of an agent may be conflicting (i.e. mutually exclusive) or even not realisable. For instance two conflicting desires may be accomplishing that *Door1 = open* and *Door1 = closed*.

- **Intentions:** the intentions are the desires that an agent has committed itself to accomplish. Therefore, the intentions drive the behaviour of an agent, as they determine the actions it takes. For instance, adopting the intention to accomplish the desire *Door1 = open* above, may require the execution of some protocol that, in the end, causes the corresponding physical door to be open.

The mental state of an agent is used as input to the practical reasoning process that determines its execution flow. The practical reasoning carried out by an agent is composed by two main activities: *deliberation* and *means-ends reasoning*. Deliberation refers to the process by which an agent decides which of its desires it will pursue (i.e. which desires become intentions). The means-end reasoning is the procedure followed by an agent in order to determine by which means (i.e. available actions) it will achieve an end (i.e. an intention). Traditionally, the means-ends reasoning process had been implemented by planning techniques. A planner is a component of the agent that receives as input: an intention or goal, a set of actions available to the agent and the set of agent's beliefs. This planner then searches the solution space and outputs an execution flow that the agent should follow in order to eventually achieve the goal given as input. However, the planning approach to means-ends reasoning suffers from a series of drawbacks.

Planning activities frequently assume that the agents possess accurate information and that they are not resource-bounded (e.g. computationally or by real-time constraints). This assumption is not realistic for even the simplest systems, as it is very difficult (if not impossible) for an agent to predict all the effects of its actions over time. Besides, assuming that the information that the agent handles is accurate requires the agent's environment to freeze during the time that the planning computations take. Therefore, the BDI architecture proposes an alternative approach to planning based on the utilisation of *partial plans*. These partial plans are alternatives provided by the programmer to the agent in order to fulfil its intentions, hence constraining the solution space to be searched.

An agent must first check the applicability of the different alternatives, with respect to its own conception of the world. Then, the agent weighs the different applicable alternatives and, ideally, selects one depending on its expected utility. Note also that these plans are partially instantiated (i.e. may contain free variables that are bound at execution time), which means that they are not completely defined in advance and can adapt to the specific state of affairs at the moment of their execution. For instance, consider a plan aimed at acquiring a book that, among other steps, requires paying for the book. This plan may be partially defined and the exact method of payment (cash or using different credit cards) may be delayed until the moment of the purchase.

In summary, the BDI model provides a useful high-level abstraction for the underlying software system (the motivational stance). The concepts of belief, desire and intention are intuitive to humans and, hence, their use for the design and description of complex systems facilitates the understanding about such systems for, ideally, a much broader audience. From a programmer's perspective, this abstraction also brings the classical advantages of declarative programming as results from the implementation of a practical reasoning process (equivalent to an interpreter). The programmer does not need to specify each single step (in contrast to traditional imperative languages), but just provide a representation of the different BDI ingredients and partial plans. The programmer then relies on the underlying BDI run-time environment (implementing some BDI architecture) to carry out the proper deliberation and means-ends activities. As a result, a language claiming to implement the BDI model must provide a framework that enables the implementation of agents that are able to deal with incomplete information, to adapt to a rapidly changing environment and to be responsive.

Even though the BDI model describes the rationale behind each of its theoretical ingredients, it does not specify how each of them should be computationally represented. It leaves then room for a relatively high degree of flexibility for the different agent-oriented technologies that implement the BDI model. One of the first realisations of a BDI architecture was the formalism known as the Procedural Reasoning System (PRS), which is a predecessor of Jason.

The Procedural Reasoning System

The PRS, described in, e.g., [Georgeff and Lansky, 1986, Ingrand et al., 1992], proposes a formalism for the specification and execution of BDI-agents. The work in [Georgeff and Lansky, 1986] implements the BDI approach to means-ends

reasoning. The programmer must provide a series of preformed plans (also known as Knowledge Areas) or *procedures* among which the agent will have to choose in order to achieve its goals. This approach is inspired by day-to-day human behaviour, where the humans have similar preformed plans to, for instance, commute by public transport. In this case, a human knows in advance which railway lines to take and probably even the most time-efficient combination of these (i.e. a preformed plan), which contrasts to checking the railway map everyday to calculate such route (i.e. classical planning).

Another contribution of the preformed plans in the PRS derives from the fact that they not only determine a sequence of actions to be executed, but may as well introduce new (sub-)goals for the agent to pursue during the execution of the plan. The preformed plans to achieve these subgoals can, in turn, introduce new subgoals, and so on. The introduction of a subgoal at some step of a plan implies that such subgoal must be reached prior to the execution of the subsequent step. Continuing with the commuting example above, consider a plan, P_1 , to achieve the goal, G_1 , of “getting to work”. This plan may be decomposed in subgoals to, for instance, “get from home to Home Station” (the railway station), $G_{1.1}$, “travel along the railway network from Home Station to Office Station”, $G_{1.2}$, and “get from Office Station to the office”, $G_{1.3}$. A plan, $P_{1.2}$ to accomplish the subgoal $G_{1.2}$ (travelling along the railway network) may be further subdivided into the subgoals of: “getting into the next train of line 1”, $G_{1.2.1}$, “changing at Main Station to the next train of line 2”, $G_{1.2.2}$, and, finally, “getting out of the train at Office Station”, $G_{1.2.3}$. Therefore, the execution of plan P_1 may require the execution of the plans ($P_{1.1}, P_{1.2}, P_{1.3}, P_{1.2.1}, P_{1.2.2}, P_{1.2.3}$), for its different subgoals.

The modularity of the PRS plans, by their decomposition into subgoals, highly increases the flexibility of such plans, facilitating their adaptation (and, by extension, that of the agent’s behaviour) to dynamic environments. Consider the case that some unexpected eventuality prevents a commuter from taking his/her usual route (i.e. a preformed plan), e.g. a disruption of railway line 1 due to a power shortage. Then, any route that requires travelling along the railway line 1 cannot be followed, like that of plan $P_{1.2}$ above. Instead, a less-efficient, alternative route, i.e. an alternative preformed plan, $P'_{1.2}$, must be taken. Then, the execution of the same plan P_1 , requires the execution of a different set of plans ($P_{1.1}, P'_{1.2}, P_{1.3}$). For clarity, this example is graphically represented as a tree-like structure in Figure 2.1. In this figure, the different goals and subgoals are represented as circular branch nodes. Each goal has one and only one direct descendant, which corresponds to the plan chosen for execution in order to achieve such goal. The different plans are

Figure 2.1: Preformed plans executed by a commuter to achieve its goal G_1 in a) an uneventful day and b) in the presence of a disruption in line 1 (disabling the execution of plan $P_{1.2}$).

represented by rectangular nodes. A rectangular branch node corresponds to a plan that introduces one or more subgoals, represented as its direct descendants using a dashed edge. A rectangular leaf node corresponds to a plan whose execution does not introduce any new subgoal. Note that in both cases a) and b), the plan to achieve the goal G_1 is the same plan P_1 . The only difference is the different plans $P_{1.2}$ and $P'_{1.2}$ executed in order to reach the subgoal $G_{1.2}$ introduced by P_1 .

The PRS implements a BDI architecture and, therefore, it introduces a computational representation for the building blocks of the BDI model. Each agent in the PRS is then composed by:

- 1) A database that contains the **beliefs** of the agent as Prolog-like facts. These beliefs are atomic formulas of first-order logic. For instance, considering the physical door referenced in Section 2.1.1, the existence of a belief *door(open)* within the database represents the fact that the agent believes this door to be open.
- 2) A set of logical expressions that represents the agent's **goals**. The PRS introduces, and gives semantics to, a series of symbols (i.e. $!$, $?$, $,$, $\#$, $\rightarrow$, $\sim$, $>$) that are

prefixed to some condition C in order to compose a goal. For instance, given the condition $valve1(open)$, the expression $!valve1(open)$ represents the desire of the agent to open the aforementioned door. Similarly, the expression $?valve1(open)$ is used to test whether that door is open or not.

- 3) A library containing the preformed **plans** (recall that they are implemented by the agent's programmer) to achieve the different goals. Each of these PRS plans contains the following elements:
 - The *goal* that the plan can be used to achieve i.e. the situation that the execution of the plan will (potentially) bring about. For simplicity, this goal can be considered as the post-condition of the execution of the plan.
 - The *context* that specifies a situation that must be fulfilled for the plan to be executable i.e. under which circumstances the agent may start the execution of the plan. The context is then a set of pre-conditions necessary for the plan to be applicable.
 - The *body* of the plan specifies the course of action to take for the agent to achieve the plan's goal. The introduction of new subgoals is possible (as mentioned above) if it is specified in the plan body. These subgoals shall be fulfilled before the plan execution concludes.
- 4) An **intention** structure that contains all the plans selected to accomplish the agent's goals. Each of the plans in this structure will be eventually executed by the agent.

These elements of the architecture, along with an interpreter that accesses and updates them, defines the behaviour of the agent. In a nutshell, once some agent's goal in 2) is adopted as intention (deliberation), the interpreter selects, with the aid of the set of beliefs in 1), an appropriate plan from the plan library in 3) (means-ends reasoning) and adds such a plan to the intentions in 4) for its future execution.

Over the years, several implementations of the PRS, as well as extensions to it, have been developed, using different programming languages. The Stanford Research Institute developed a Lisp implementation known as PRS-CL [Myers, 1997] [Myers, 2001]. The University of Michigan implemented UM-PRS [Lee et al., 1994] using the programming language C++. A Java implementation, JAM [Huber, 1999], inspired by PRS-CL and UM-PRS is also available. The first implementation in the C programming language, known

as C-PRS [Ingrand et al., 1996], has recently evolved into the open-source tool OpenPRS [Ingrand et al., 2007].

Of special relevance to Jason is the C++ implementation by the Australian AI Institute, known as the Distributed Multi-Agent Reasoning System, dMARS [Kinny, 1993]. The dMARS system has served as basis for the development of the commercially available agent development framework JACK [Busetta et al., 1999] (that extends JAVA with elements whose functionality is inspired by the BDI software model) and the abstract programming language AGENTSPEAK(L), a predecessor of Jason.

The programming language AGENTSPEAK(L)

The programming language AGENTSPEAK(L) [Rao, 1996], from now on referred to as AgentSpeak, was introduced with the aim of bridging the gap existing between the different theoretical formalisations of the BDI (like, e.g., the BDI modal logics [Wooldridge, 2000]) and the existing practical implementations of BDI architectures (e.g. the PRS). AgentSpeak provides a textual specification language that, while capturing the most relevant functionalities of the PRS and dMARS within its execution model, assists the ascription of mental attitudes to the agent. Prior to AgentSpeak, the mental attitudes of an agent had to be explicitly specified as logic formulas that complemented their computational representation in a language. AgentSpeak, instead, allows the identification of these attitudes with the different constructs of the language and its execution model. This change of perspective facilitates, among others, the formal verification of properties specified in BDI logics.

The program that determines the behaviour of an AgentSpeak agent is composed by a set of base beliefs (equivalent to facts in traditional logic programming languages) as well as a set of plans. For readability, these constructs are described next, along with the new ones introduced by the first implementation of an interpreter of AgentSpeak: the programming language Jason, which is the centrepiece of the remainder of this Section.

2.1.2 Syntax and Informal Semantics of Jason

Jason was first introduced in [Bordini et al., 2005, Bordini and Hübner, 2006] as an interpreter, implemented in JAVA [Gosling, 2000], for an extension of the programming language AgentSpeak. This language extension has since been known as the Jason programming language. Jason is the first practical implementation of

an interpreter of AgentSpeak, as the latter remained an abstract language. While AgentSpeak focuses mainly in the internal structure of an agent, Jason introduces, among others, mechanisms for the inter-communication of agents as well as their distribution over a network. Besides, Jason facilitates the customisation of the interpreter as well as the interaction of the agents with their environment (all these requiring some JAVA programming).

Next, the most relevant language constructs and their informal semantics are described, addressing which parts are native to AgentSpeak or introduced by Jason. A formal semantics is available in Chapter 5. For clarity, the commuting example from the previous section is used to illustrate the different examples.

Terms

The most basic terms of AgentSpeak's syntax are similar to Prolog [Colmerauer and Roussel, 1996, Clocksin and Mellish, 2003] terms. An *atom* is an alphanumeric symbol starting with a lower-case letter, e.g. `symbol`. A *string* is a concatenation of characters in quotations, e.g. `"I am a string"`. Integers and floating point numbers are also allowed. A *list* is a structure of the form `[H|T]`, being *H* and *T* the header and tail of the list, respectively. A *variable* is an alphanumeric expression starting by an upper-case letter. A variable is said to be *unbound* or *uninstantiated* if it has not yet been unified to a value. A predicate has the form `functor( $t_1, \dots, t_n$ )` for $n \geq 0$. The functor is an alphanumeric expression starting by a lower-case letter. The terms $t_1, \dots, t_n$ are any of the aforementioned terms. Predicates and atoms may be suffixed with a list structure whose elements are known as annotations, e.g. `functor( $t_1, \dots, t_n$ )[ $annot_1, \dots, annot_m$ ]` (see below). Both atoms and predicates (annotated or not) are referred to as *literals*.

Beliefs and Rules

Each *belief* within an agent's belief base in AgentSpeak is a *ground predicate*, i.e. either an atom or a predicate whose terms are not variables. For instance, a belief `at(home)` within the belief base of some agent, represents the fact that such agent believes to be at home. Recall that there is no guarantee about the accuracy about the information of the agent's beliefs.

In AgentSpeak, the truth value of a predicate *p* is determined by the existence of a most general unifier θ such that $p\theta$ belongs to the agent's belief base. For instance, consider an agent that possesses only one belief, `at(home)`, and consider then the two predicates $p_1 = \text{at(Place)}$ and $p_2 = \text{at(station)}$. The truth value of

p_1 is *true* as the unification of the free variable **Place** to the value **home** is possible. On the contrary, no unification is possible for the predicate p_2 and, consequently, its truth value is *false*.

The classical logic operators for disjunction and conjunction are represented as $|$ and $\&$, respectively. Jason implements two kinds of negation operators. The negation operator **not** follows a closed-world assumption such that, given some predicate p , the disjunction $p|\text{not}p$ is a tautology (i.e. the law of excluded middle applies). The negation operator $\sim$ follows an open-world assumption such that $\sim p$ is true if and only if the agent explicitly believes p (or a predicate that can be unified to it) to be false. Note that, then, a disjunction $p|\sim p$ is not always true.

The belief base of a Jason agent also contains inference rules that allow the generation of new knowledge. For instance, the following Jason rule can be used to compute the factorial of a given natural number:

```
factorial(1,B) :- B = 1.

factorial(A,B) :- A > 1 & factorial(A-1,C) &
                  B = A * C.
```

Goals

Given a predicate p , two types of *goals* can be defined: an *achievement goal* $!p$ that represents the agent's desire of bringing about a state of affairs such that it believes p to be true; and a *test goal* $?p$ used to assess the truth value of p . As a matter of example, consider the goals $g_1 = !\text{at}(\text{home})$, $g_2 = ?\text{at}(\text{home})$ and $g_3 = ?\text{at}(\text{Place})$. The achievement goal g_1 represents the agent's desire to go home (i.e. to believe itself to be at home). The test goal g_2 is used to check whether the agents believes to be at home. The test goal g_3 is used to check the location of the agent by finding an unification for the variable **Place**.

Events

Whenever a new belief b is added to or removed from the agent's belief base and whenever a goal $!g$ or $?g$ is adopted by the agent, a new *event* is generated. Such event is of the form $\{te, i\}$ where te is called the *triggering event* and i is the *related intention*. This triggering event would be $+b$, $-b$, $!g$ and $?g$, respectively, for the aforementioned cases. The related intention is the intention (described below) whose execution generated the event. This related intention is empty, represented

as $\top$, if the event is external (as results of perception or message processing) or if it corresponds to a belief addition or deletion. Therefore, $i \neq \top \Rightarrow te \in \{+?g, +!g\}$. Note that the agent's events potentially influence its behaviour, as they may trigger the execution of plans.

Apart from the events described above, Jason introduces failure events that are generated when either there are no applicable plans for a goal addition event or some failure occurs during the execution of a plan (e.g. the attempt to bind an already bound variable with a different value). These events are either $\{-!g, te\}$ or $\{-?g\}$ for the failure of plans relevant to achievement or test goals, respectively. The inclusion of failure events increments the fault-tolerance of the agent, as they enable the implementation of failure handling plans whose execution is triggered when such failures occur.

Annotations

Jason introduces an annotation mechanism for the different belief and goal predicates that is mainly intended to determine the source of the belief addition/deletion or goal adoption. Given a belief b or goal g , their possible annotations, $a_1, \dots, a_n$ are appended to them within brackets: $b[a_1, \dots, a_n]$ or $g[a_1, \dots, a_n]$. At least one of these annotations corresponds to the source of the belief/goal and is of the form **source(Source)**. **Source** is either the atom *percept* if the belief/goal was acquired via perception; the atom *self* if the agent introduced it itself during the execution of a plan; or any other atom *name* that corresponds to the name of the agent that introduced the belief/goal via communication (the different communication directives are described below). As a matter of example, consider the beliefs $b_1 = \text{at}(\text{home})[\text{source}(\text{self})]$, $b_2 = \text{at}(\text{home})[\text{source}(\text{percept})]$ and $b_3 = \text{at}(\text{home})[\text{source}(\text{alice}), \text{source}(\text{bob})]$ that only differ from each other in their source of information. Other kinds of predicates can be used as annotations at the programmer's discretion as a convenience to, for instance, associate additional information. For instance, a goal $\text{!at}(\text{home})[\text{source}(\text{alice}), \text{time}(13,10)]$ may represent an achievement goal delegated by an agent named *alice* at 13:10h (i.e. it carries a timestamp).

With regards to the unification of annotated predicates (necessary to, e.g. determine their truth value), i.e. to unify $p[a_1, \dots, a_n]$ and $q[b_1, \dots, b_m]$, the most general unifier θ must be such that the following conditions hold:

- $p\theta = q$, i.e. their functor can be unified

- For each annotations a_i of p , there exist some annotation b_j of q such that $a_i\theta = b_j$

For instance, consider an agent whose belief base contains the following belief $bel_1 = \text{belief}[\text{source}(\text{alice}), \text{source}(\text{bob})]$, and the predicates $p_1 = \text{belief}[\text{source}(\text{alice})]$ and $p_2 = \text{belief}[\text{source}(\text{Source})]$ and $p_3 = \text{belief}[\text{source}(\text{alice}), \text{source}(\text{charles})]$. In this case, p_1 can be unified to bel_1 . The predicate p_2 can be also unified to bel_1 if the variable **Source** is bound to either **alice** or **bob**. Finally, the predicate p_3 cannot be unified to bel_1 , as the annotation **source(charles)** cannot be unified to any of the annotations of bel_1 . Note also that unification is not reciprocal, i.e. while p_1 unifies to bel_1 it is not the case that bel_1 unifies to p_1 . Therefore, an assignation $\text{belief}[\text{source}(\text{alice})] = \text{belief}[\text{source}(\text{alice}), \text{source}(\text{bob})]$ will succeed while $\text{belief}[\text{source}(\text{alice}), \text{source}(\text{bob})] = \text{belief}[\text{source}(\text{alice})]$ will fail.

Plans

A *plan*, as in the PRS, is of the form $te : ctx \leftarrow body$. where te is the triggering event (representing what the plan is relevant to, i.e. what it can help achieve), ctx is the plan context (a series of conditions that should hold for the plan to be applicable) and the plan $body$, which is a sequence of formulas $f_1, \dots, f_n$ that, executed in order, represent the recipe provided by the plan. These plans are partially instantiated, i.e. they contain variables that are dynamically bound at execution time. For instance, consider the following plan for a commuter to get from home to work:

```
+!get_to(work): at(home) & date(D,M,Y) & not (weekend(D,M,Y) | holiday(D,M,Y)) <-
  leave(home);
  !walk_to(homeStation);
  !travel_railway(homeStation,officeStation);
  !walk_to(office).
```

Note that the triggering event is represented in the same way as the triggering events of the events described before. Actually, a plan $p = te_{pl} : ctx \leftarrow body$. is considered to be relevant to some event $e = \{te_{ev}, i\}$ if and only if te_{ev} unifies to te_{pl} . In the plan above, the possible triggering events that unify are e.g. $\text{get_to}(\text{Somewhere})$ and $\text{get_to}(\text{home})[\text{source}(\text{self})]$.

The context of a plan may contain several logical expressions as well as predicates. This context must be satisfied for the plan to be applicable. In the example above, the context specifies the following:

- 1) Check that the agent believes itself to be at home (i.e. the predicate `at(home)` has a truth value of true).
- 2) Check the date and bind variables `D`, `M` and `Y` (day, month and year, respectively). Note that, in this case, the condition is used to retrieve information from the belief base.
- 3) Ensure that it is not weekend or a holiday. In this case, this condition requires the truth value of the predicates `weekend(D,M,Y)` and `holiday(D,M,Y)` to be false.

In summary, this context states that an agent can execute this plan to go from home to work only if such agent is at home on a workday.

With regards to the formulas in a plan body, there are several kinds of them:

- *Goal additions*, represented as `!g` or `?g` for achievement and test goals, respectively. Jason introduces a special operator, `!!` for the addition of achievement goals. A formula `!!g` generates an event $\{+!g, \top\}$, i.e. whose related intention is empty.
- *Mental notes* correspond to the addition or deletion of a belief (the source of these beliefs is the agent itself), written `+b` and `-b`, respectively.
- *Expressions* that represent variable assignation, e.g. `A = value`, or that impose conditions to be satisfied for the plan to continue its execution, e.g. `A == value`, `A \== value` or `A > B*3`.
- *Actions*, that can be *external* or, as introduced by Jason, *internal*. An external action is represented by a predicate `p` and corresponds to the execution of some operation available for the agent outside the scope of AgentSpeak. In the case of Jason, the external actions provide a hook for the execution of the code (frequently written in JAVA), that interfaces to some actuator, which potentially affects the agent's environment. Jason also allows the implementation, in JAVA (or in a different language using the Java Native Interface), of user-defined internal actions. Internal actions are represented by a symbol `.p`, i.e., a predicate `p` preceded by a full stop. The agent's internal actions can be used as a convenience to equip the agents with the capabilities to carry out useful, recurrent tasks like, e.g., manipulating strings or scanning a list. Jason provides a standard library of such internal actions.

In the sample plan above, the first formula in the body is an external action that the agent executes to leave home. The remaining formulas are achievement goal additions that introduce further subgoals.

Intentions

The agent's intentions are represented as a stack of partially instantiated plans. Each plan in the stack, except for the one at the bottom, corresponds to the means to accomplish a subgoal introduced by the plan immediately underneath it. The plan at the bottom of the stack is then the one from which all other stem. As it is later described, this plan is the intended means to accomplish either an initial goal or a goal acquired through perception (which includes communication).

When an agent decides to execute an intention, it does so by executing the actions of the plan on top of the stack. When all the actions in the body of this plan have been executed, it is popped from the stack. Therefore, the size of the stack varies over time until it is empty.

For instance, consider again the example plan for a commuter to get from home to work:

```
+!get_to(work): at(home) & date(D,M,Y) & not (weekend(D,M,Y) | holiday(D,M,Y)) <-
  leave(home);
  !walk_to(homeStation);
  !travel_railway(homeStation,officeStation);
  !walk_to(office).
```

The corresponding intention will always have this plan at the bottom of the stack until it is empty. The next position, if not empty, will correspond to the plan for either `!walk_to(homeStation)`, `!travel_railway(homeStation, officeStation)` or `!walk_to(office)`, and so on.

Agent communication

The agents in a Jason system interact with each other via asynchronous message-passing. Messages sent to an agent are stored in a mailbox associated with the agent. This communication is based on the speech-act [Searle, 1969], like other well-known agent communication languages as the Knowledge Query and Manipulation Language [Finin et al., 1994] (KQML) or the FIPA ACL [Fipa, 2002]. Following the aforementioned speech act, the programmer does not only specify the receiver and the content of a message, but also the intended purpose of such message (known as *illocutionary force* or *performative*). For instance, a message whose content is

`today(tuesday)` may be informative (i.e. transmits the information “today is Tuesday”) or interrogative (i.e. transmits the question “is Tuesday today?”), depending on the performative used (see performatives below). In Jason, the selection of a performative translates into different effects on the receiver. Consider an agent *s* that sends a message with content **Cont** to an agent *r*. The effects of this utterance on the receiver with respect to the different performatives implemented in Jason are the following:

- **tell**: adds a new belief, **Cont**[**source(s)**] to the belief base of the receiver.
- **untell**: removes a belief **Cont**[**source(s)**] from the belief base of the receiver (if it exists).
- **achieve**: delegates an achievement goal to the receiver, i.e. it receives a goal addition event $\{+!\mathbf{Cont}[\mathbf{source(s)}], \top\}$.
- **unachieve**: requests the receiver to drop a goal (desire) as if it had invoked the internal action `.drop_desire(Cont)` (see [Bordini et al., 2007]).
- **askOne**: delegates a test goal **?Cont** to the receiver and provides the sender with a reply.
- **askAll**: similar to *askOne* but, instead of a single matching result for **?Cont**, it returns a list of all possible matchings.
- **tellHow**: adds a new plan **Cont** to the plan base of the receiver.
- **untellHow**: removes a plan, labelled as **Cont**, from the plan base of the receiver.
- **askHow**: requests the receiver to provide a list with all the relevant plans for **Cont**, which are added to the plan base of the sender.

Interpretation: the reasoning cycle

The initial state of an agent is composed by:

- The sets of *initial beliefs* and *inference rules*, which determine the agent’s belief base in its initial state.
- The set of *initial goals*, which are the achievement goals that the agent pursues from its initial state.

- The plans that compose the agent's initial plan base.

From this initial state, the agent evolves with respect to the interpretation procedure implemented by Jason, known as the *reasoning cycle*. This reasoning cycle characterises the rational behaviour of the agents as, in each iteration, an agent perceives its environment (including the processing of incoming messages), deliberates on which desires to pursue (i.e. events that have not yet been processed), decides how to achieve such desires by means-ends reasoning (i.e. selects an applicable plan from its plan base) and acts according to such decisions (i.e. executes formulas from the partially instantiated plans on top of an intention stack). The different steps that compose the reasoning cycle of a Jason agent, where the possible transitions between them are mapped into Figure 2.2, are the following:

- **ProcMsg:** during this initial step, the agent obtains information from its environment (perception) and from the messages received from other agents. This information may update the belief base and provide new goals, generating the corresponding events in each case.
- **SelEv:** one of the unprocessed events, if any (otherwise take the transition *b*), is chosen to be processed during the current iteration. Such event is selected using an agent-specific *event selection function*, S_e , that incarnates the agent's deliberation process. Then, take the transition *a*.
- **RelPl:** the set of relevant plans for the event selected in the previous step is computed. Then take the transition *c*. If there is no relevant plan either the event is discarded (in the case of belief additions/deletions) or a failure event is generated (in the case of goal additions). In this case, take the transition *d*.
- **ApplPl:** the set of applicable plans is computed from the set of relevant plans. Then, take the transition *e*. If the set of applicable plans is empty, either the selected event is discarded or a failure event is generated (for the same cases as before). In such case, take the transition *f*.
- **SelAppl:** one, and only one, of the applicable plans is selected using the agent-specific *option selection function*, S_o .
- **AddIM:** if the selected event possesses a related intention (i.e. it is a subgoal added during the execution of an instruction in the body of another plan), the selected applicable plan is put on top of such intention (recall that an intention is a stack of plans). Otherwise, a new intention, only containing the selected applicable plan, is added to the set of intentions.

- **SelInt:** if the set of intentions is empty, take the transition g . Otherwise, an agent-specific *intention selection function*, S_I , selects one intention from the set of intentions of the agent. Note that each intention represents a different focus of attention of the agent. Then, take the transition h .
- **ExecInt:** the first formula in the body of the plan on top of the intention stack is executed, triggering some modification of the agent's mental state or its environment (e.g. adding a new belief or sending a message), along with the generation of the corresponding event. If the formula executed is not a goal addition of type $!g$ or $?g$, such formula is removed from the body of the plan. Otherwise, the execution of the intention is suspended until the subgoal introduced is fully executed (the suspended intention appears as a related intention to the corresponding goal addition event). Note that only one formula is executed in each iteration of the cycle.
- **ClearInt:** if the body of the plan on top of the intention is not empty (i.e. the plan has not been fully executed) the intention is returned to the set of intentions and transition i is taken, starting a new iteration of the cycle. Otherwise, such plan is removed from the top of the intention stack. If there are more plans left in the intention, transition j is taken. If the intention is empty, it is completely removed and transition i is taken.

Figure 2.2: Possible state transitions within a reasoning cycle in Jason.

Internal Actions

The following list includes some of the internal actions that Jason implements and which are frequently used within the code snippets provided throughout the document:

- **.create_agent**

An agent can create (and name) another agent using the internal action

```
– .create\_agent(AgentName, Source,[Customisations])
```

where **AgentName** is an atom that corresponds to the name of the newly created agent. The parameter **Source** is a string that represents a path to the file that contains the source code of the agent (typically, a file with extension **.asl**). Finally, some optional customisations can be specified in order to use alternative versions of the Java classes that implement the agent behaviour (agentClass), the agent's architecture (agentArchClass) and the agent's belief base (beliefBaseClass).

- **.fail**

An agent can force the generation of a failure event by executing an internal action that does not take any input parameter

```
– .fail
```

The execution of this internal action always fails.

- **.kill_agent**

An agent can kill another agent using its unique name by using the internal action

```
– .kill\_agent(AgentName)
```

where the parameter **AgentName** is an atom that corresponds to the name of the agent that is killed.

- **.my_name**

An agent can unify a variable with its own name using the internal action

```
– .my\_name(Name)
```

The execution of this internal action fails if the input parameter is already unified to anything other than the atom that corresponds to the name of the agent.

- **.send**

The communication between agents in Jason requires the use of the internal action

– `.send(AgentName, Performative, Message [Reply, Timeout]),`

where the parameters in brackets are optional and **AgentName** is the symbolic name of the agent receiving the message. The parameter **Performative** represents the intention of the communication, whose value is one of the illocutionary forces already described in Page 20 . The parameter **Message** is a literal which, along with the performative compose the main content of the communication. The parameters **Reply** and **Timeout** can only be used when the performative is of type **askOne** or **askAll**. In such case, the variable **Reply** will be bound to the reply to the message, if any is provided before **Timeout** milliseconds. Otherwise, the variable **Reply** is bound to the atom **undefined**.

For instance, note the difference between the outcome of the execution of the following actions by some agent *sender*:

- a) `.send(receiver, tell, at(home))`
- b) `.send(receiver, askOne, at(Home), Reply)`
- c) `.send(receiver, achieve, at(home))`

In this case, *a*) will include a new belief `at(home)[source(sender)]` into the belief base of the agent *receiver*. When *b*) is executed, the agent *receiver* adopts a test goal `?at(Home)[source(sender)]`. Similarly, the execution of *c*) causes the adoption of an achievement goal `!at(Home)[source(sender)]` by the agent *receiver*.

- **.wait**

The execution of an intention can be suspended for **Time** milliseconds using the internal action

– `.wait(Time)`

Similarly, the execution of an intention can be suspended until some event **event** belongs to the agent's set of events using the action

– `.wait('Event')`

Note that, in this case, the parameter is a string.

Finally, it is also possible to suspend the execution of an intention until either an event occurs or some time (again in milliseconds) has passed, whichever occurs first, using:

– `.wait('Event', Time)`

2.1.3 Formal Semantics: Jason's Grammar

On the basis of the formal semantics for AgentSpeak given in [Vieira et al., 2007], a formal semantics for the interpretation of Jason programs was published in, e.g. [Bordini et al., 2007]. This semantics were further refined in [Bordini and Hübner, 2010] including the specification of plan failures along with the formal semantics for several internal actions. The Jason semantics is represented as a series of transition rules between agent states. The initial state of an agent being determined by its Jason program, which follows the following grammar (it is slightly simplified for the sake of readability):

$$\begin{aligned}
ag &::= bs \ ps \\
bs &::= b_1 \dots b_n & (n \geq 0) \\
ps &::= p_1 \dots p_n & (n \geq 1) \\
p &::= te : ct \leftarrow h \\
te &::= +at \mid -at \mid +g \mid -g \\
ct &::= ct_1 \mid \top \\
ct_1 &::= at \mid \neg at \mid ct_1 \wedge ct_1 \\
h &::= h_1; \top \mid \top \\
h_1 &::= a \mid g \mid u \mid h_1; h_1 \\
at &::= P(t_1, \dots, t_n)[s_1, \dots, s_m] & (n \geq 0, m \geq 0) \\
s &::= percept \mid self \mid id \\
a &::= A(t_1, \dots, t_n) & (n \geq 0) \\
g &::= !at \mid !!at \mid ?at \\
u &::= +b \mid -at
\end{aligned}$$

This grammar represents an agent as *ag*, composed by a set of beliefs *bs* and plans *ps*. Each belief, *b*, is a ground atomic formula, i.e. a possibly annotated predicate which does not contain any variable. Conversely, an atomic formula, *at*, represents a possibly annotated predicate which may either be a variable itself or contain variables in some of its arguments, *t_i*, and/or annotations, *s_i*, but not only in its predicate functor *P* (i.e. an unbound variable *V* cannot be annotated, making

$V(t_1, \dots, t_n)$ an invalid atomic formula). Each individual plan, p , is composed by a triggering event, te , a context, ct , and a plan body, h , that may be empty ($h = \top$). The context of a plan is represented by a conjunction of literals (where a literal is either at or $\neg at$). Each instruction h_1 in a plan body may either be an action to execute, a (with functor A and arguments t_i), an addition or deletion of a goal, g and g' , or a belief update, u . Finally, the annotations s of the different atomic formulas are themselves possibly annotated predicates (i.e. annotations can be nested).

For readability, the formal semantics of Jason are provided in Section 5.1.2, immediately preceding those of eJason as they are subsumed by that of the latter.

2.2 The programming language Erlang

Erlang [Armstrong, 2007, Cesarini and Thompson, 2009] is a functional concurrent programming language created by Ericsson in the 1980s which follows the actor model [Hewitt et al., 1973]. Erlang is a dynamically typed and strictly evaluated functional programming language that provides high support to the development of dependable and reliable software. It enables programmers to write robust and clean code for modern multiprocessor and distributed systems.

2.2.1 Erlang as a functional programming language

As a functional programming language, Erlang implements, among others, the following features typical to this programming paradigm:

- **Higher order functions:** which, in a nutshell, are functions that may take other functions as arguments.
- **Immutability of data:** variables, once bound to a value, cannot be modified. This immutability of data implies the single-assignment of variables, which disallows assignments like $x = x + 1$ relatively frequent in imperative languages.
- **Recursion:** Erlang functions may invoke themselves, hence iterating their execution. Note that tail-recursion (including a call to itself as the last instruction of a functions) is optimised by the Erlang compiler.

2.2.2 Concurrency and distribution

The chief strength of the language resides in its excellent support for concurrency and distribution. All these features are inherent to the language and, therefore, Erlang provides the proper constructs to leverage them. Most importantly, these constructs can be used in a declarative way, hence alleviating the responsibility of their control from the programmer and unto the Erlang Runtime System.

An Erlang system (see Fig. 2.3) is a collection of Erlang nodes. An Erlang node (or Erlang Runtime System) is a collection of processes (actors), with a unique node name. These processes run independently from each other and do not share memory. They interact via communication. Communication is asynchronous and point-to-point, with one process sending a message to a second process identified by its process identifier (pid). Messages sent to a process are put in its message queue, also referred to as a mailbox. Note that, by using the pid of a process in order to interact with it (e.g. sending a message to its mailbox), the actual distribution of the system at runtime is made transparent to the programmer.

Figure 2.3: An Erlang multi-node system

As an alternative to addressing a process using its pid, there is a convenience for associating a symbolic name with a pid. The name, which must be an atom¹, is automatically unregistered when the associated process terminates, so that it can be reused. Message passing between processes in different nodes is transparent when pids are used, i.e., there is no syntactical difference between sending a message to a

¹An atom is a literal, i.e. a constant starting with a lowercase letter

process in the same node, or to a remote node. However, the node must be specified when sending messages using registered names, as the pid registry is local to a node. For instance, in Fig. 2.3 let the process Proc C be registered with the symbolic name `procC`. Then, if the process Proc B wants to send a message to the process `procC`, the message will be addressed to `procC@Node2`.

2.2.3 Fault tolerance

A unique feature of Erlang that greatly facilitates building fault tolerant systems is its provision of fault detection and fault recovery mechanisms.

To enable the detection of faults, the Erlang monitoring mechanism can be utilised. One process can monitor another process in order to detect its termination. This capability is especially useful when this termination comes unexpectedly (i.e. in case of an abnormal termination of a process). It is possible to monitor processes at remote nodes. However, note that if the symbolic name of the monitored process, instead of its pid, is used to establish the monitor, the name of the node must be also specified. This functionality is provided by an Erlang function named `erlang:monitor`. If a process P_1 monitors another process P_2 , and P_2 terminates, process P_1 is automatically informed by the Erlang runtime of the termination of P_2 .

The following features characterise the monitoring mechanism of Erlang:

- Each process P_1 may enable as many monitors to the very same process P_2 . In case of a termination of process P_2 , the process P_1 will receive as many notifications as monitors to P_2 where enabled. Each of these monitors is uniquely identified by an Erlang reference². These references represent the result value of the execution of each call to the function `erlang:monitor`.
- A monitor can be disabled by invoking the function `erlang:demonitor` including the corresponding Erlang reference as an argument to the function call.
- There is no restriction to the number of monitors that a given process P_1 may setup to other processes P' .

²An Erlang reference is a convenience introduced by the Erlang programming language used to label distinguishable Erlang terms. An Erlang reference created in an Erlang node is only repeated after 2^{82} calls to the functions that generate them. Therefore, they are considered unique. See, e.g. [Cesarini and Thompson, 2009] for further details

- The notification received by the process P_1 , with regards to its monitor to the process P_2 , is a message that informs about one of:
 - The exit reason of the process P_2 , which may provide information of the causes of its termination.
 - The absence of process P_2 , i.e., informs that no process P_2 can be found in order to setup the monitor.
 - The absence of a connection to the node where P_2 resides (as provided by parameter to the function call to `erlang:monitor`).
- A monitor is disabled when it generates a notification (due to any of the causes above). Therefore, for any given a process P_1 that monitors another process P_2 , whenever P_2 dies, a notification to P_1 is sent. However, should P_2 be respawned and die again, no further notification is sent to P_1 due to the aforementioned monitor.

Another interesting feature of Erlang is the *supervisor behaviour*. A behaviour is an abstraction, fully supported by the runtime environment, for common capabilities of processes. These behaviours were introduced by the Open Telecom Platform (OTP). The OTP is a set of libraries and design principles for the development of distributed systems in Erlang. The supervisor behaviour provides useful software mechanisms to implement fault tolerant systems. It enables the implementation of the supervisor process in a supervision tree. A supervision tree defines a hierarchical structure of processes for fault tolerance purposes. Each process (node) in a supervision tree is either a supervisor (branch nodes) or a worker (leaf nodes). The basic idea is that a supervisor keeps all its child processes alive. The worker processes carry out different computations, i.e. they do the actual work. The supervisor processes monitor the behaviour of their child processes (either workers or other supervisors) and restart them when necessary. A supervisor process spawns and monitors all its child processes. If a supervisor process terminates, all its child processes terminate as well.

The main features of the supervisor mechanism of Erlang are the following:

- A supervisor process is responsible of starting all its supervised processes. Therefore, within the code for a supervisor process, instructions of how to spawn all its children shall be provided. These instructions are enclosed within the so-called *child specification*. A child specification is a list of tuples that provides the following mandatory information for each child process:

- *Id*: the child identifier, formerly referred to as name
- *Start*: the instructions to spawn the child process. They amount to the module (source file), function (defined in the preceding module) and arguments to invoke in order to start the child process.

Optionally, a child specification may determine whether a child process shall be restarted if it stopped. The possible values for this *restart* are:

- *Temporary*: the child process is never restarted, even if it stopped as a side-effect of a restart strategy (see below).
 - *Permanent*: the child process will always be restarted
 - *Transient*: the child process is restarted unless it stops normally (Erlang defines the conditions for a normal termination of processes, whose exact details are unimportant to this work).
- The child processes are spawned in the same order as they appear in the child specification.
 - In contrast to the monitoring mechanism, all the processes in a supervision tree are strongly coupled, as their existence is linked to that of the process immediately above in the structure. As a matter of example, consider the supervision tree depicted in Figure 2.4. The process *Supervisor*₁ spawns the processes *Worker*₁ and *Supervisor*₂ which, in turn, spawns the processes *Worker*₂ and *Worker*₃.
 - Stopping a supervisor process also stops all its children. If any of these processes is a supervisor process, its children are also stopped, and so on. Note that this mechanism enables the orderly termination of the processes in the supervision tree.
 - Stopping a child process (either a worker or a supervisor) triggers the so-called *restart strategy*. A restart strategy applies only to a supervisor process and its immediate children. Therefore, in a supervision structure, there exist as many restart strategies as supervisor processes. The restart strategy is an attribute of a supervision whose value, along with its corresponding meaning, is one and only one of:
 - *one_for_one*: the process that stopped is restarted the same initial arguments used to spawn it.

- *one_for_all*: all child processes are stopped and restarted as above.
- *rest_for_all*: the child process that stopped is restarted, along with the child processes whose start instructions appear after those of the stopped one in the child specification (i.e. the so-called “rest”).
- *simple_one_for_one*: similar to *one_for_one*, but allowing the addition of child processes dynamically. Note that all these children share the same start instructions.

Recall that, in all restart strategies above, for a process to be restarted, the restart field of its child specification should be either permanent or transient.

Note that the default value for a restart strategy is *one_for_one*.

Figure 2.4: Sample Erlang Supervision Tree

The supervisor behaviour is widely used in order to implement fault-tolerant Erlang system. Its different restart strategies enable the implementation of fault-recovery for Erlang processes in a concise and elegant way, which are part of the design goals of the Erlang programming language.

Chapter 3

GOVERNING JASON AGENTS

This chapter describes the first of the contributions of this thesis. It provides a thorough analysis of the control mechanisms provided by the Jason programming language, and proposes a number of improvements. The different sections elaborate on different vulnerabilities introduced by the multiplicity of foci of attention of Jason agents. These vulnerabilities relate to those of the distributed systems (e.g. race conditions and data dependencies). For each vulnerability, the current language constructs provided by Jason are analysed and an alternative approach is proposed. The proposed alternatives are implemented in eJason. In a nutshell, these solutions aim at decreasing the non-determinism inherent in some of the Jason constructs.

Section 3.1 elaborates on the synchronisation mechanisms available, and analyse how the different possible alternatives to address the classical problem of data dependencies impact the readability of the code. Section 3.2 identifies the vulnerabilities derived from a loose coupling between the evaluation of a plan's context (i.e. a guard) and the execution of its body (i.e. the guarded code). Section 3.3 describes the approach of eJason to help the programmer deal with the weak time guarantees provided by the reasoning cycle of Jason. Section 3.4 elaborates on the mechanism introduced by eJason to allow the programmer to requeue test goals as desired. Finally, Section 3.5 describes how these extensions impact the reasoning cycle of Jason, i.e., describes the resulting reasoning cycle of eJason agents.

3.1 Synchronisation mechanisms

As introduced in Section 2.1.2, in the section where the Jason reasoning cycle is presented, the interpreter of Jason allows each agent to possess several foci of attention, corresponding to the different intentions of the agent. These intentions compete for the attention of the agent and the decision on which intention to execute, in each

iteration of the reasoning cycle, is determined by the agent's intention selection function. The execution order of the different intentions is not always irrelevant. The plans in the different intentions access and update the information stored in the agent's belief base. Therefore, the modification of the set of beliefs, derived from the execution of an intention, may affect the outcome (or even totally prevent the execution) of the rest of intentions available. The programmer must then consider these data dependencies between the different intentions of an agent's mental state and, when necessary, control the synchronisation of the execution of such intentions.

Data dependencies are more likely (whereas not exclusive) among intentions that contain different instances of the same plan, where that plan reads and updates some belief. As a matter of example, consider an agent which maintains a counter of, e.g., the number of files that it has uploaded. The most simple approach is updating such counter (a belief `files_loaded(Num)`) every time that a file is uploaded. This behaviour can be implemented by the Jason code provided in Figure 3.1.

```

+!load(File) <-
  load(File);           // a
  ?files_loaded(Num);    // b
  -->files_loaded(Num+1). // c

```

Figure 3.1: Jason plan for the file counter

Consider an agent with only this plan in its plan base and with initial goals $g_1 = !load(file1)$ and $g_2 = !load(file2)$. The intentions corresponding to these goals are composed by one instance of the plan above, i.e. $I_1 = [p_1]$ for g_1 and $I_2 = [p_2]$ for g_2 with plan bodies $\{a_1; b_1; c_1\}$ and $\{a_2; b_2; c_2\}$, respectively, where a_1 represents the formula a applied to g_1 , and so on.

Several executions of the agent code, using the standard intention selection function, show that the counter is not always properly updated, as sometimes it only records the upload of one file, while, in fact, two files have been uploaded. A simple exploration of all possible execution traces, represented by the different possible interleavings of the actions in the plan bodies, exposes the root causes of the problem. These interleavings are depicted in Figure 3.2. This figure shows a graph where each node, labelled $I_1 I_2 Ctr$, represents a different configuration of the agent's mental state such that $I_i \in \{a_i, b_i, c_i\} \cup X$ for $i \in \{1, 2\}$ corresponds to the action to be executed if the intention I_i gets selected by the intention

selection function (the symbol X is used as a placeholder if the corresponding intention has been fully executed) and Ctr is the value of the counter (i.e. a belief `file_loaded(Ctr)`). For instance, the node b_1X1 corresponds to a mental state such that the selection of the intention I_1 implies the execution of the formula b_1 , the intention I_2 has been fully executed and the belief base contains a belief `files_loaded(1)`. Note then that the node $XX2$ corresponds to the outcome desired (i.e. the counter is properly updated), while the node $XX1$, shadowed in the graph, corresponds to an undesirable one (i.e. when the counter only records the upload of a single file, instead of two). The different edges in the graph represent the transition between mental states, and their label corresponds to the formula executed during that transition. For instance, the outgoing edge a_1 (resp. a_2) from the node a_1a_20 to the node b_1a_20 (resp. a_1b_20) represents the transition triggered by the execution of the formula $a_1=\text{load}(\text{file1})$ (resp. $a_2=\text{load}(\text{file2})$). The analysis of the execution traces shows that the undesired outcome occurs when the actions b_1 and b_2 (i.e. the actions where the value of the counter is read) have been executed without the execution of neither c_1 nor c_2 (i.e. the actions that update the counter) in-between (i.e. all traces containing the state c_1c_20). The reason is that, in these cases, one of the intentions is handling outdated information regarding the counter and, therefore, the result is incorrect.

The remainder of Section 3.1 describes the approach taken by the Java implementation of Jason to solve this kind of problem as well as the alternative solution (equivalent, whereas more elegant) contributed by this thesis work.

3.1.1 Jason: Atomic Plans

The Java implementation of Jason enables the labelling of plans as a way of including meta-level information that alters the agent's reasoning cycle. The label *atomic* is one of such labels. A plan labelled as atomic, also referred to as *atomic plan*, is such that, once this plan is selected for execution during an iteration of the reasoning cycle, all subsequent iterations will also select this intention until the atomic plan is fully executed. More informally, the atomic label represents a way of temporarily disabling the multiplicity of foci of attention, keeping the attention of the agent in the intention until the atomic plan is executed.

In order to avoid the data dependency explained at the beginning of this section, an atomic plan can be used. For instance, by replacing the plan in Figure 3.1 for the two (semantically dependent) plans provided in Figure 3.3.

Figure 3.2: Possible interleavings for the counter update example.

```

+!load(File) <-
  load(File);           //a
  !update_counter.

[@atomic]
+!update_counter <-
  ?files_loaded(Num);    //b
  -+files_loaded(Num+1). //c

```

Figure 3.3: File counter with an atomic plan

This way, the formulas b and c are always executed consecutively. Therefore, the aforementioned faulty execution traces are not allowed, as depicted in Figure 3.5. This solution can be used not only for belief updates, but also to implement behaviours that require the agent to maintain its focus of attention on the same intention for several iterations of the reasoning cycle (i.e. executing the formulas from the same intention consecutively). For instance, when a programmer tries to

minimise the time that an agent requires exclusive access to a shared resource like, e.g. to write the result of certain computation into a file, the plans in Figure 3.4 can be used.

```

+!task(A,B, File) <-
  compute(A,B,Result)
  !write_result(Result, File);
  [...] // some other operations
  clean_up.

[@atomic]
+!write_result(Result,File) <-
  open_file(File);
  write(Result,File);
  close(File).

```

Figure 3.4: Sample use of the Jason atomic plans

Figure 3.5: Possible interleavings for the counter update example using synchronisation mechanisms.

Note that this approach requires the introduction of additional plans in order to determine the sets of actions that must be executed consecutively without a change

in the focus of attention, even for relatively simple plans. This thesis work considers that this approach is not ideal from a programmer’s perspective. It obscures the code, as it increases the number of plans in the agent’s plan base, consequently increasing the complexity to maintain the agent’s program.

It is a premise of this work that providing several plans for the same goal shall be used to provide several alternatives for its accomplishment, not to implement continuations of the same plan. Therefore, to avoid the aforementioned synchronisation problem, this thesis work proposes an alternative solution which is described in the following section.

3.1.2 eJason: Critical Sections

For the sake of clarity, an analogy to a well-known and widespread synchronisation mechanism is used. Consider a mapping such that the different intentions of an agent represent the different processes that run concurrently in a computer. These processes compete for the use of the processor (in a similar way as the intentions compete for the agent’s attention) and share a common memory space (as the intentions share access to the agent’s belief base). A scheduler (typically provided by the operative system) decides which process must be executed by the processor at anytime. However, the programmer sometimes requires certain degree of control over the scheduling procedure in order to, e.g., prevent the potential hazards derived from the data dependencies existing among the processes. Requiring the programmer to provide its own implementation of the scheduler is not optimal, as it is a complex and error-prone task. Similarly, requesting the Jason programmer to provide an alternative implementation of the intention selection function (i.e. a scheduler establishing an execution order for the intentions of the agent) may be considered suboptimal. One of the most popular approaches to satisfy the synchronisation needs of the programmer is the implementation of *critical sections*, i.e. sections of a program such that, whenever a process p is executing the code within a critical section, no other process p' can enter such section until p has left it. The critical sections typically delimit code regions that include instructions that access and update shared resources (e.g. a program variable shared by different execution threads or a file stored in the disk).

Inspired by this classical solution, this thesis proposes the introduction of symbols that enable the specification of critical sections within the body of Jason plans. The symbols proposed are “{” and “}” to delimit the beginning and end of a critical section, respectively. This solution is similar to the critical

section synchronisation mechanism implemented by several classical programming languages like, e.g. the mutex (mutual exclusion) construct, which is described in the POSIX Threads execution model [Butenhof, 1997] and is implemented by the C programming language [Jtc, 2011] in order to control the synchronisation of (concurrent) multiprocessed software applications. The semantics of the Jason critical sections proposed slightly vary from those typically implemented in other programming languages. When an agent executes a formula within a critical section, there can be no change in the focus of attention as long as the critical section is not left (as happens during the execution of an atomic plan).

Using critical sections as synchronisation mechanism, the program for the agent that updates a counter, introduced earlier in this section, would be as shown in Figure 3.6.

```
+!load(File) <-
  load(File);
  {{?files_loaded(Num);
  -+files_loaded(Num+1)}}.
```

Figure 3.6: File counter with eJason critical sections

This plan provides the same functionality as the combination of the two plans above (Figure 3.3), i.e., the possible execution traces are the ones depicted in Figure 3.5), whereas in a more concise and elegant way. In the case of atomic plans, the parametrisation (via labelling) of the intention selection function requires the programmer to consider the different intentions that may conflict and establish a priority order for their execution. In contrast, using critical sections, the programmer only identifies the regions of the agent’s program that should be executed without interference from other intentions of the agent. It is the task of the programmer to implement minimal critical sections (i.e. ones that are as short as possible).

In a nutshell, Jason atomic plans and eJason critical sections are equivalent in order to tackle data dependencies, being their difference purely syntactic. However, as shown above, the latter enables a simpler and more concise code, which adheres to the spirit of the declarative programming paradigm.

3.2 Intention selection function

Consider a simplified multi-agent system showing a classical client-server architecture. The client agents must write information into different files. In order to avoid conflicts generated by simultaneous attempts of writing different information into the same file by different agents, the access to the file is managed by a server agent. A client agent sends a message to the server agent to request the exclusive rights to access a file before it can write into such file. When the exclusive access to the file is no longer necessary, the client agent asks the server (again, via message passing) to “unlock” the resource. The server agent handles different requests from the agents.

The Jason code for the client agents is included in Figure 3.7. It shows that, when a client agent has the goal of writing some text, **Text**, into a file, **FName**, it must first send an achieve message to the server requesting the lock over the file (i.e. delegating a goal of the shape **!lock(FName)** to the server). Then, it waits for the notification about the acquisition of exclusive access to the file. This notification is represented by a belief update event **+granted(FName)**. After the reception of this notification, the client agent writes the text into the file and requests the server to unlock the file (again, delegating this task as an achievement goal).

```

+!write(FName, Text) : true <-
  .send(server, achieve, lock(FName));
  .wait("+granted(FName)");
  write(Text, FName);
  .send(server, achieve, unlock(FName)).

```

Figure 3.7: Jason code for the client agents

The Jason code for the server agent is given in Figure 3.8. The plan, referred to as **PSrv1**, handling the achievement goal to lock some file, **FName**, delegated from some client, **Client**, requires such file to exist and not to be blocked by other agent. If these conditions hold, the first plan can be applied, which amounts to adding a mental note, **+blocked(Client, FName)**, recording that **Client** has exclusive access to the file **FName**. Then, it notifies the client by sending a tell message with the belief **granted(FName)**. The plan handling the achievement goal to unlock a file, referred to as **PSrv2**, checks whether the file exists and whether it is locked by the same agent that attempts to unlock it. The recipe provided by this plan implies erasing the aforementioned mental note that records the exclusive access granted to the agent **Client** over the file **FName** and, finally, notifying this client agent about the successful unlock of the file.

```

+!lock(FName)[source(Client)] :    //PSrv1
  file(FName) & not blocked(_,FName)<-
    +blocked(Client,FName);
    .print("Agent ",Client," locks ",FName);
    .send(Client, tell, granted(FName)).

+!unlock(FName)[source(Client)] : //PSrv2
  file(FName) & blocked(Client,FName) <-
    -blocked(Client,FName);
    .print("Agent ",Client," unlocks ",FName);
    .send(Client, tell, unlocked(FName)).

```

Figure 3.8: Jason code for the file server agent

The reiterated execution of this system (to explore different interleavings) shows that, sometimes, the locks are not properly granted, as several agents acquire access to the same file simultaneously. The analysis of the different execution traces indicates that the problem derives from the program semantics implemented by the Jason interpreter, described in Section 2.1.2. Consider the following excerpt of an execution trace for the server agent (recall that the behaviour of an agent is determined by the computations carried out within the different steps that compose an iteration of the agent's reasoning cycle):

Iteration #1:

The belief base contains one belief: `file(f1)`.

The set of events includes the goal additions corresponding to lock requests for the same file `f1` from client agents `c11` and `c12`: `{+!lock(f1)[source(c11)], +!lock(f1)[source(c12)]}`.

The selected event is `+!lock(f1)[source(c11)]`

The plan `PSrv1` is applicable, as its context conditions `file(f1)` and `not blocked(_,f1)` hold, and is therefore added to the set of intentions (labelled `PSrv1-f1c11`).

The set of intentions contains an intention `other` apart from `PSrv1-f1c11`. The specific details of the intention `other` are irrelevant for the example. It is an abstraction utilised to represent the case when, during an iteration, the intention selected for execution (see next step) is not the same as the last intention added to the set (see previous step).

The intention selected for execution is **other**. Its execution does not generate new events. This intention gets removed from the agent's set of intentions.

Iteration #2:

The belief base contains one belief: `file(f1)`. Note that the belief `+blocked(c11,f1)` has not been added yet, as the intention `PSrv1-f1cl1` was not selected for execution in the previous iteration.

The set of events includes the goal addition not selected in the previous iteration: `{+!lock(f1)[source(c2)]}`.

The selected event is `+!lock(f1)[source(c2)]`

The plan `PSrv1` is applicable, as its context conditions `file(f1)` and `not blocked(_,f1)` hold and is therefore added to the set of intentions (labelled `PSrv1-f1cl2`).

The set of intentions contains the intentions `PSrv1-f1cl1` and `PSrv1-f1cl2`.

The intention selected for execution is `PSrv1-f1cl1`. Therefore, the first formula in its plan body is executed, which adds the belief `+blocked(c11,f1)`. As the plan is not fully executed, the intention containing the remaining formulas is added to the set of intentions (labelled `PSrv1-f1cl1'`).

Iteration #3:

The belief base contains two beliefs: `{file(f1), blocked(c11,f1)}`.

[...]

The set of intentions contains the intentions `PSrv1-f1cl1'` and `PSrv1-f1cl2`.

The intention selected for execution is `PSrv1-f1cl2`. Therefore, the first formula in its plan body is executed, which adds the belief `+blocked(c12,f1)`. As the plan is not fully executed, the intention containing the remaining formulas is added to the set of intentions (labelled `PSrv1-f1cl2'`).

Iteration #4:

The belief base contains three beliefs: `{file(f1), blocked(c11,f1), blocked(c12,f1)}`. Note that both clients `c11` and `c12` obtain the lock over the file `f1` simultaneously.

[...]

The problem here derives from the fact that the server does not effectively register a lock over a file until the first formula in an instance of the corresponding plan **PSrv1** is executed. Therefore, as shown by the execution trace above, there can be several instances of this plan for the same file in the set of intentions, leading to write conflicts.

Note that this problem stems from the fact that the evaluation of a plan context and the execution of its plan body are decoupled. The Jason semantics allow several iterations to take place between the one in which the plan context is evaluated and deemed applicable; and the iteration in which the plan is chosen for execution. This may be a problem in systems similar to the one described above, which require the plan context evaluation and the subsequent execution of the plan body to be as close in time as possible. The Java implementation of Jason addresses this problem in a very different way from the one proposed in this thesis work. Both approaches are described next.

3.2.1 Jason: customisation of the intention selection function

The solution to the problems derived from the delay between the evaluation of the plan context and its actual execution, requires the customisation of the agent's intention selection function. This customisation shall ensure that whenever a new instance of the plan **PSrv1** is added to the set of intentions, the corresponding intention (the one that contains this instance on top of the stack) is selected for execution during that same iteration. This way, a belief **blocked(Client,FName)** is immediately added, preventing further inclusions of instances of **PSrv1** for the same file before this file is unlocked. Note also that this problem cannot be solved just with the use of atomic plans, as an applicable atomic plan is not always chosen for execution during the same iteration in which it is added.

As described in [Bordini et al., 2007], the customisation of the intention selection function must be done in the Java programming language. This solution is equivalent to implementing a custom process scheduler, analogous to the solution described in 3.1.2. This solution requires the programmer not only to be fluent in the Jason programming language, but also on its implementation, which is arguably not an optimal approach.

3.2.2 eJason: consecutive evaluation and execution of a plan

The alternative solution proposed in this thesis work implies the provision of a means to ensure that, whenever necessary, the evaluation of the plan context (i.e. the conditions that enable the execution of the plan) and the execution of its plan body can be carried out consecutively.

This alternative is implemented as a modification to the Jason interpreter such that, when the intended means for some event starts with a critical section (i.e. when the first formula in the body of the plan on top of the last added intention belongs within a critical section), that intention is selected for execution. Therefore, if the programmer needs to implement a plan that gets the agent's attention as soon as it is deemed applicable, he/she must just include a critical section at the beginning of the body of such plan. Note that it is not mandatory for such critical section to span the whole plan body, only to be located at its beginning, whatever its length.

The implementation of the server agent using the code given in Figure 3.9 would then suffice, as the files are immediately locked.

```

+!lock(FName)[source(Client)] : //PSrv1
    file(FName)&not blocked(_,FName)<-
    {{+blocked(Client,FName)}};
.print("Agent ",Client," locks ",FName);
.send(Client, tell, granted(FName)).

+!unlock(FName)[source(Client)] : //PSrv2
    file(FName) & blocked(Client,FName) <-
    -blocked(Client,FName);
.print("Agent ",Client," unlocks ",FName);
.send(Client, tell, unlocked(FName)).

```

Figure 3.9: Jason code for the file server agent using critical sections

Note that, exploiting this feature of eJason, the programmer can easily identify the plans that, when deemed applicable, should immediately get the agent's attention. Note also that this implementation of the Jason interpreter does not represent a new agent behaviour, as the consecutive execution of the evaluation of a plan context and its plan body was allowed by the original Jason semantics. However, there did not exist any means to narrow the possible execution paths to such case, when necessary.

3.3 Achievement Goals

Consider again the multi-agent system described in the previous section. The code in Figure 3.9 removes the problems derived from having several agents simultaneously blocking the same file. However, the system does not yet provide the desired functionality. An analysis of the resulting execution traces shows that, whenever a server agent gets the lock over a file, the requests from different client agents to lock the same file are disregarded (just dropped) by the server agent, as the context of the relevant plan `PSrv1` cannot be satisfied. For instance, consider the following excerpt of an execution trace of the server agent showing this behaviour:

Iteration #1:

The belief base contains one belief: `file(f1)`.

The set of events includes the goal additions corresponding to lock requests for the same file `f1` from client agents `c11` and `c12`: `{+!lock(f1)[source(c11)], +!lock(f1)[source(c12)]}`.

The selected event is `+!lock(f1)[source(c11)]`

The plan `PSrv1` is applicable, as its context conditions `file(f1)` and `not blocked(_,f1)` hold, and is therefore added to the set of intentions (labelled `PSrv1-f1c11`).

The set of intentions contains the intentions `PSrv1-f1c11` and `other` (irrelevant to the example).

The intention selected for execution is `PSrv1-f1c11`, because it starts with a critical section. The belief `blocked(c11,f1)` is added to the belief base.

Iteration #2:

The belief base contains the beliefs: `file(f1)` and `blocked(c11,f1)`.

The set of events is: `{+!lock(f1)[source(c12)], +blocked(c11,f1)}`.

The selected event is `+!lock(f1)[source(c12)]`

The plan `PSrv1` is not applicable as the condition `not blocked(_,f1)` does not hold. The related event is dropped. Therefore, the request from the client agent `c12` is, by all means, as good as ignored.

[...]

Note that the Jason interpreter does not provide the programmer with any means to declare that the applicability of an achievement goal shall be retried at a later moment. Instead, the programmer must design and implement some means to provide this behaviour. In contrast, the approach proposed in this thesis work, considers the necessity of such retry as default. Both approaches are described next.

3.3.1 Jason: maintaining a record of intentions to retry

The simplest solution to avoid dropping all the achievement goals that cannot be immediately handled (due to the lack of applicable plans) implies recording them within the agent's mental state in order to posteriorly pursue them. This solution can be achieved, e.g., adding a new plan, `PSrv3`, whose context matches whenever the file is already blocked by a different client agent. Following this plan, the server agent records the requests that cannot be immediately served by, e.g. returning the achievement goal addition event to the set of events:

```
+!lock(FName)[source(Client)] :    //PSrv3
    file(FName) & blocked(_,FName)<-
!lock(FName)[source(Client)]. // requeue
```

This solution, and similar ones, require the programmer to introduce a plan whose context is satisfied whenever the contexts of other (preferred) plans are not. As stated in the previous section, this approach obscures the code, and is very demanding, as increases the complexity of the code design with each plan whose preferred execution order must be implemented. Besides, note that by just enqueueing again the selected achievement goal addition event, the mental state of the agent is reinstated after a complete iteration of the reasoning cycle, hence consuming computational resources without changing the aforementioned mental state. Moreover, given the non-determinism of the selection functions, this event may be selected in consecutive iterations of the reasoning cycle, possibly enqueueing it in all such iterations (this behaviour is guaranteed if the belief base of the agent has not been updated, e.g. by other intentions, in-between), degrading the performance of the agent.

The Jason interpreter (as an interpreter of AgentSpeak), by considering a failure the lack of applicable plans for a given goal addition event, requires the programmer to consider time constraints (i.e. the moment at which the agent's reasoning cycle must attempt the computation of the sets of relevant and applicable plans for such events) that are highly demanding, given the weak guarantees inherent to distributed

systems. This requirement translates into strengthening the vulnerability of multi-agent systems to race conditions. As a matter of example of the potential impact of race conditions, consider the following Jason code for an agent:

```
at(office). // Initial belief

!go(home). // Initial goals
!read(book).

+!read(Item): at(home) <-
    read(Item).

+!go(home): at(office) <-
    drive(home).
```

This agent initially believes to be at the office and possesses, simultaneously, the desires of going home and reading a book. There are two possible outcomes for the execution of this agent. In both of them, the agent goes home (as the plan to accomplish such desire is always applicable in the initial state). However, the agent does not always satisfy its desire of reading a book. If it tries to find an applicable plan for this desire before going home, the desire is dropped (there is no relevant plan for the failure event generated). Therefore, not that this agent program provides two different outcomes: the agent going home and reading a book; or the agent just going home. The actual outcome being determined by the moment in which the intention selection function selects a desired to be fulfilled (recall that, by default, it is implemented as a random selector).

A programmer intending the agent to achieve both goals has to ensure that the goals are selected in the desired order. This requires considering all the possible interleavings and implementing some suitable synchronisation mechanism. The complexity of this synchronisation increases exponentially with respect to the number of goals to synchronise.

3.3.2 eJason: requeuing not applicable achievement goals

The alternative approach proposed in this thesis implies not generating a failure event for achievement goal addition events (i.e. events of the shape $\{+!g, \iota\}$). Instead, when one of these events is selected and there are no relevant or applicable plans for it, the event is returned to the agent's set of events (i.e. requeued). This way, the programmer does not need to be concerned about the timing of the

selection of these events as they can be selected again later. For instance, the agent in the simple example above would always go home and read a book (i.e. avoiding possible race conditions). Hence having one and only possible outcome for the agent program.

In contrast to the Jason alternative above, the achievement goal addition events queued are not selected by the selection functions if the belief base is not update. Therefore, the performance of the agent is potentially better.

Note that it could be the case that the intention of the programmer were to provide a means for the agent to desist on its desire of reading a book were it not at home. This thesis work claims that such behaviour shall not be relied upon the randomness of the intention selection function. Instead, the constructs already provided by the language, like the internal action *.drop_intention*, should be used. The following example shows how this construct can be used in this case:

```
at(office). // Initial belief

!go(home). // Initial goals
!read(book).

+!read(Item): at(home) <-
    read(Item).

+!read(Item): not at(home)<-
    .drop_intention(read(Item)).

+!go(home): at(office) <-
    drive(home).
```

The authors of the Java implementation of Jason already noticed that need and instrumented their Jason interpreter to admit a special configuration parameter, *requeue*, at startup. By using this special configuration parameter, the alternative semantics described above are used, as described in [Bordini et al., 2007]). However, the proposal in this work implies the selection of this semantics as the default one.

3.4 Test Goals

With regards to test goals, the impact of race conditions is as considerable as that for achievement goals described above. Whenever a programmer introduces a test goal, $?g$, into the body of a plan, p , this programmer must consider the possibility that such test goal may fail (along with the whole plan). This failure occurs if the test, g , cannot be satisfied when the corresponding test goal addition event, $\{+?g, \iota\}$, is selected by the agent's event selection function. When the programmer does not want such failure to happen, he/she must either introduce some synchronisation mechanism such that the event is only selected when it is satisfiable or ensure that the test, g , can always be satisfied. The first alternative requires, again, the customisation of the event selection function, which involves a re-implementation of the interpreter. The second alternative cannot always be applied, as described in the next section.

3.4.1 Jason: implementing busy waiting

Consider an agent whose program is the following:

```
gather_results(Res1,Res2) :-
    result(t1, Res1) & result(t2, Res2).

+!task3(Result) <-
    .send(alice, achieve, t1);    // 1
    .send(bob, achieve, t2);
    .wait(10000);                // 2
    ?gather_results(Res1,Res2);   // 3
    operation(Res1,Res2,Result).  // 4
```

In order to provide its functionality, the agent cooperates (interacts) with other agents. This cooperation amounts to 1) delegating some tasks, $t1$ and $t2$, to agents *alice* and *bob*, respectively; 2) waiting for ten seconds (in order to let the agents provide their reply); 3) gathering the results, *Res1* and *Res2*, obtained from *alice* and *bob* and 4) using *Res1* and *Res2* in order to compute some final result *Result*. In this case, the programmer decided to explicitly suspend the execution of the plan for ten seconds before executing the test goal that gathers the results received from the agents it cooperates with. This time span should be long enough for those results to be received, otherwise the whole plan fails. An alternative approach requires

splitting the functionality of the plan in two different plans such that the second one is only applicable when the replies are available, otherwise iterates:

```
gather_results(Res1,Res2) :-
    result(t1, Res1) & result(t2, Res2).

+!task3(Result) <-
    .send(alice, achieve, t1);
    .send(bob, achieve, t2);
    !continue_task3(Result).

+!continue_task3(Result) : ?gather_results(Res1,Res2);
    operation(Res1,Res2,Result).

+!continue_task3(Result) : not ?gather_results(Res1,Res2);
    !continue_task3(Result).
```

This, and similar approaches, decrease the readability of the code and increase its complexity, as stated already in Section 3.1.1. Besides, the busy wait implemented consumes computational resources.

3.4.2 eJason: providing different operators

This work proposes the introduction of a new operator for the addition of test goals, “?”. The semantics of a goal $??g$ is similar to the semantics of a $?g$ test goal. The difference lies in the treatment that the corresponding goal addition event, respectively $\{+??g, \iota\}$ and $\{+?g, \iota\}$, receives from the eJason interpreter. When the test g cannot be satisfied using the information in the agent’s belief base, for some test goal $\{+??g, \iota\}$, this event is returned to the agent’s set of events (note the similarity to the proposed semantics for achievement goal addition events). Therefore, this event will be selected again at a later iteration of the reasoning cycle.

The code for the agent above would then be:

```
gather_results(Res1,Res2) :-
    result(t1, Res1) & result(t2, Res2).
```

```

+!write(FName, Text) :true <-
  .send(server, achieve, lock(FName));
  ??granted(FName);
  write(Text, FName);
  .send(server, achieve, unlock(FName)).

```

Figure 3.10: Final Jason code for the client agents

```

+!task3(Result) <-
  .send(alice, achieve, t1);
  .send(bob, achieve, t2);
  ??gather_results(Res1, Res2);
  operation(Res1, Res2, Result).

```

Note that the new operator “??” is used to introduce a mechanism to suspend the execution of an intention until some conditions are met. This mechanism provides a functionality similar to that of the internal action `.wait(Event)`. However, while `.wait` relies on the occurrence of a single event as condition for the reactivation of an intention, the operator “??” establishes a goal g that must be matched in order to reactivate the intention. The condition imposed by a goal may be as complex as a Jason rule. This capability is useful to provide simpler code, as shown in Figure 3.10, where the behaviour of the client agent introduced in Section 3.2 does no longer need the inclusion of two separate, though semantically dependent, plans.

3.5 The Reasoning Cycle of eJason agents

The introduction of the new language constructs and semantics described above alters the agent’s interpreter. This Section describes the changes to the agent’s reasoning cycle. The modifications to the semantics are described in Section 5.1.4.

The possible state transitions depicted in Figure 2.2 have increased. These new transitions, depicted in Figure 3.11 as dashed transitions, are the following:

- When there are no relevant plans for either an achievement goal or a test goal introduced using the operator “??”, the transition k is taken.
- During the execution of a critical section no new events are selected. After executing an action within a critical section, the transition l is taken.

- During the execution of a critical section, the focus of attention does not change. After the addition of the intended means for a goal within a critical section, the transition m is taken.
- During the execution of critical sections, the presence of failures introduces new transitions. The absence of relevant plans for an event causes transition n . The emptiness of the set of applicable plans causes transition o .

Figure 3.11: Possible state transitions within a reasoning cycle in eJason.

Finally, note that the inclusion of the different proposals described in this section into the Jason interpreter allows the implementation of the behaviour of the client and server agents, introduced in Section 3.2, using the code included in Figure 3.10 and Figure 3.9, respectively.

Chapter 4

DISTRIBUTION AND FAULT TOLERANCE

This chapter describes the second contribution of this thesis work. It describes the distribution model and fault-tolerance capabilities introduced into Jason via its extension, eJason. These extensions accommodate the relevant strengths of the Erlang programming language and expand their functionality. The aim of these extensions to Jason is bringing the efficiency of Erlang not only by implementing the Jason interpreter in Erlang, but also by introducing constructs that, in a declarative way, leverage the Erlang capabilities. While the distribution schema and the fault-tolerance mechanisms introduced by eJason are discussed in this chapter, more details on the implementation are given in Chapter 6. The contents of this chapter are organised as follows.

Section 4.1 describes the distribution model implemented in eJason. It addresses the distribution schema, the necessary conditions for any two agents to be considered connected (i.e. able to communicate) and the dynamics of the model. Then, follows an analysis of the improvements that this extension represents to Jason. The section concludes with the API for distribution related activities.

Section 4.2 describes the fault-tolerance mechanisms introduced by eJason. This section begins with an overview of the techniques introduced. Then it focuses on each of these techniques, i.e. the monitoring mechanism and the supervision mechanism. In both cases, a comparison with the corresponding Erlang mechanisms is provided in order to specify how the Erlang mechanism expand the functionality of the Erlang ones. Finally, the API for the monitoring and supervision mechanisms is included.

4.1 Distribution

The Jason programming language does not provide language constructs to implement distributed systems. Instead, it allows the development of distributed multi-agent systems by interfacing with JADE [Bellifemine et al., 2007, Bellifemine et al., 2005]. In contrast, eJason introduces language constructs that enable an agent distribution. The agent distribution model extension to Jason is inspired by the distribution model of Erlang. It complies with the three design goals of eJason: distribution transparency (i.e., ensuring that any two agents can communicate, even if oblivious to the physical location of each other), efficiency (via decentralised control), and minimising the impact on the syntax of the Jason programming language.

4.1.1 Distribution Schema

The terminology used to describe the distribution model is described below:

- A **multi-agent system** is composed by one or more agent containers. Unlike in other agent development environments, such as JADE, there is no hierarchy between agent containers.
- Each **(agent) container**¹ belongs to one and only one multi-agent system, is comprised of a set of agents and is located on a computing host. It is given a *name* that is unique in the whole system. Concretely each container name is of the form `NickName@HostName`, where `NickName` is an arbitrary short name given to the container when it is started and `HostName` is the full name of the host in which the container runs. For instance, the name `home@avalor-laptop.fi.upm.es` corresponds to a container that runs in a host whose name in the system is `avalor-laptop.fi.upm.es`.
- Each **agent** is present in a single container and will stay within it during its lifetime. At the moment of agent creation, it is given a symbolic name that is unique within the system. This unique name can be provided statically (specified by its source code) or dynamically (given by the runtime system upon its creation).

Using the unique name of an agent, agents can communicate with each other irrespectively of the containers they reside in.

¹the name container is used to emphasise the similarities with a JADE container

As a matter of example, consider a distributed a multi-agent system that provides the functionality of a simplified e-commerce service. This system is physically distributed in four machines, namely **homePC**, **salesPC**, **madridPC** and **barcelonaPC**, as depicted in Figure 4.1. These machines are spread across different locations, respectively: the buyer's home, the company's headquarters and two storehouses in Madrid and Barcelona. Each machine hosts one of the agent containers named **onlineShopping**, **orderDepartment**, **storeHouse1** and **storeHouse2**. Within each **storeHouseN** container, run three agents named: **managementN**, **accountingN** and **deliveryN**, while the **orderDepartment** container hosts a single agent named **orderProcessing** and the **onlineShopping** container hosts a single **buyer** agent.

Figure 4.1: eJason multi-agent system for a simplified e-commerce service

4.1.2 Connectivity

Given the distribution schema presented above, the following restrictions apply:

- r1) Any two different agents can interact directly (communicate, monitor, supervise, ...) if and only if their respective containers are connected.

- r2) All the agent containers belonging to the same multi-agent system are interconnected.
- r3) If two agent containers are interconnected, they belong to the same multi-agent system.

In a nutshell, these restrictions imply that an agent can interact with (and only with) all other agents that belong to the same multi-agent system.

For the sake of formality, the definition of the connection between agent containers can be specified as follows:

- Given AC_m the set of all the agent containers in some multi-agent system m , the connection between agent containers is defined as a binary relation *Connection* over AC_m such that *Connection* is:
 - Total
 - Reflexive
 - Symmetric
 - Transitive

4.1.3 Multi-Agent System Dynamics

A multi-agent system is not a static entity, hence its composition varies over time. New agent containers can be created and added to the system or, conversely, the containers belonging to certain system can get detached from it. The following cases illustrate the possible dynamics of a multi-agent system (the agent actions that enable these dynamics are described later in the document):

- **Merge:** Consider two different multi-agent systems, m_1 and m_2 , comprised of the sets of agent containers AC_1 and AC_2 , respectively. Recall that $|AC_1| \geq 1 \leq |AC_2|$ and $AC_1 \cap AC_2 = \emptyset$, as $m_1 \neq m_2$. By restrictions **r1-r3** above, the agents belonging to different systems cannot interact (e.g. send a message) with each other. Nevertheless, any two different multi-agent systems, m_1 and m_2 , can be merged to compose a new multi-agent system m_3 if all the following conditions hold:

- $AC_3 = AC_1 \cup AC_2$

- Given $Name_i$ the set containing the names of all agents in the system m_i (recall that an agent name is unique within its system), then $Name_3 = Name_1 \cup Name_2$ and $Name_1 \cap Name_2 = \emptyset$, i.e. there are no two agents with the same name and that belong to the different systems m_1 and m_2 . Therefore, the merging of m_1 and m_2 does provoke a clash between agent names.
- **Split:** Consider a multi-agent system m_1 with a set AC_1 of agent containers. At any moment in time, this system can evolve into a different system m'_1 such that $AC'_1 \subset AC_1$ and $1 \leq |AC'_1|$. There exist two possibilities for a system split:
 - A non-empty subset, $AC_2 = AC_1 \setminus AC'_1$, of agent containers detaches from the system m_1 to form a new system m_2 (e.g. due to the execution of the disconnect action below or due to the isolation of this subset caused by a network problem).
 - A non-empty subset, $AC_2 = AC_1 \setminus AC'_1$, of agent containers is stopped (e.g. due to problems in the host(s) in which they run).

4.1.4 Improvements over Jason

Note that *all agent names are unique* within the system, which brings in some improvements:

- It enables the *transparent distribution* of the system, i.e. the agents can interact with each other completely oblivious to their respective containers. Agent names are unique within the whole system, hence any given agent only needs to know the name of the agent with which it wants to interact in order to address it. From a programmer's perspective, the implementation of distributed systems is facilitated. The logic dominating the behaviour of an agent can be made independent to the actual system orchestration at runtime. Furthermore, it facilitates a future implementation of the transparent migration of agents between containers.
- Another improvement introduced is the possibility to *delegate the naming of an agent to the runtime system* of eJason. This delegation avoids name clashes between agents (ensured by the runtime system). The drawback is that an agent created in this way needs to make itself visible to other agents (e.g. by starting a communication) in order to allow them to interact with it. The use of this feature is recommended in order to create agents that will themselves

start the interaction with other agents, instead of just waiting for others to start it. Conversely, agents that are meant to be addressed by others without prior interaction, e.g. those that play a major role in the infrastructure or that provide some shared service, should be given a name statically. For instance, consider a classical system composed by a server (an agent that provides some kind of service) and an arbitrary number of clients (other agents). The recommendation for this kind of system is providing a name, e.g. **server**, for the server agent statically and delegating the naming of the client agents to the runtime system. The logic for the clients just needs to use the name **server** to interact with the server agent. In the case of service providers, this name can be used as a contract of their services. For instance, it may be natural for an agent that provides a service to buy books to be named **bookSeller**.

- The guarantees of the internal action whose execution creates a new agent (see [Bordini et al., 2007] or the distribution API below) have changed. In the case that the name of a new agent is provided by the source code, it is either created with that name or its creation is aborted (due to a name clash). Besides, the plan executing an aborted agent creation fails. Hence, it is now possible to implement agent behaviours that react to name clashes. As a matter of example, consider a system where some agent **alice** executes a plan that requires the creation of an agent named **bob** and, later, it sends a message to it:

```
+!plan <-
  .create_agent(bob,"./bob.asl");
  .send(bob,tell,"hello!");
```

If some agent with name **bob** exists in the system prior to the creation of the new agent, a plan failure is generated. A failure plan can be provided to deal with it, e.g. by delegating the name creation to the runtime system:

```
-!plan <-
  .create_agent(Bob,"./bob.asl");
  .send(Bob,tell,"hello!");
```

Instead, following the strategy of the Java implementation of Jason the agent would be created with another name, e.g. **bob_a**. Neither the agent invoking the creation nor the new agent would be aware of this change of name. Moreover, the plan would be executed successfully and the message “hello!” would be sent to the agent with name **bob** that existed before the plan execution, which may have unpredictable consequences.

- The lack of a hierarchy between agent containers facilitates the execution of split and merge operations, hence enhancing the flexibility of the agent systems. There is no need to select a new main container after a split operation (as would happen in other platforms like JADE). Conversely, there is no need to demote a main container after a merge of two systems. Furthermore, after a split operation, all the resulting subsystems are fully operational multi-agent systems themselves.

The implementation of eJason's distribution schema requires the development of (non-trivial) mechanisms, briefly described in Section 6.1 that maintain the consistency of the system and allow its proper functioning:

- The need to keep track of the containers in which the agents run is relieved from the programmers. Hence, a *white pages service* becomes necessary in order to resolve the addressing of agents. Given some atom `AgentName`, this service will determine whether it is a valid reference to an agent in the system or not. This service must also be aware of the connections between containers, i.e. the set of all containers belonging to the system.
- Every time that the creation of a new agent is attempted, it must be checked whether a name clash would occur or not in order to allow such agent creation. Besides, if the naming is delegated to the runtime system, a means to generate a unique name at any point in time is needed.
- During a merge operation, all the agent names in the two systems that are about to be merged must be compared, whereas it is not necessary for any kind of split operation.

The different internal actions that enable the use of the operations above are described in the following Section.

4.1.5 Distribution API

The different internal actions that enable the utilisation of the capabilities of eJason's distribution can be grouped according to their functionality. Unless otherwise specified, the internal actions described below are introduced by eJason.

Container connection

The execution of the following internal action adds a container to the system:

- `.connect_to(Container)`

When an agent in some system m_1 executes this action, a merge operation is attempted. This operation merges the system m_1 and the system, m_2 , to which the container **Container** belongs. Therefore, a new system m_3 may arise. In the case that $m_1 = m_2$, i.e., when a connection to an already connected container is attempted, the execution of the internal action succeeds. The execution of the internal action fails if the merge operation cannot be completed (see the conditions for a merge operation in Section 4.1.3) or if **Container** cannot be found (which includes the case when **Container** is an unbound variable).

Conversely, an agent container AC in a system m_1 can get detached from it, or ensure that it does not belong to the same system m_2 that some agent container belongs to by the execution of the internal action:

- `.disconnect_from(Container)`

This execution fails if **Container** is a free variable. Otherwise, it succeeds and the possibilities are the following:

- If **Container** is the name of the container in which the agent executing the action runs (i.e. $AC = Container$) or if **Container** and AC are interconnected (i.e. $m_1 = m_2$), AC gets disconnected from every other container and, therefore, becomes part of a new system with only one container (itself).
- If **Container** is not connected to AC , AC does not get disconnected from any other container and the execution of the action succeeds, even if no agent container with name **Container** can be found.

As a matter of example, consider two Jason multi-agent system in Figure 4.2, inspired by a similar example in [Bordini et al., 2007]. These systems are composed as follows: the agents with the names **owner**, **robot** and **fridge** reside in a container named `home@avalor-laptop.fi.upm.es`. The agent **supermarket** runs in the container `shopping_mall@babel.ls.fi.upm.es`, located on a different host. These two systems can be merged into one if, e.g., the robot agent executes the internal action:

```
.connect_to(shopping_mall@babel.ls.fi.upm.es')
```

Figure 4.2: Connection of Multi-Agent Systems

Similarly, these containers may split if, e.g. the agent `supermarket` executes the internal action:

```
.disconnect_from('home@avalor-laptop.fi.upm.es')
```

An agent can discover which containers (apart from its own) belong to the system using the internal action:

- `.containers_list(ContainerList)`

The variable `ContainerList` is bound to a list with the names of the containers belonging to the system, excluding the one corresponding to the agent that executes the action. Note that, if the system is composed by only one container, such list is empty. As a matter of example, the following plan can be used to print (in the standard output) the name of the containers connected to that where an agent runs:

```
.print_connected_containers <-
  .containers_list(ContList); //binds the names to ContList
  .print("The connected containers are: ", ContList).
```

Agent Communication

The communication between agents in eJason requires the use of the internal action (already existing in Jason):

a) `.send(AgentName, Performative, Message [Reply, Timeout])`,

where the parameters in brackets are optional and **AgentName** is the symbolic name of the agent receiving the message. The parameters **Reply** and **Timeout** can only be used when the performative is of type **ask**. For the sake of concision, the exact description of their meaning is not reproduced in this work. They remain unchanged from [Bordini et al., 2007].

The symbolic name must uniquely identify one agent within the system. Thus the above internal action can be used only to communicate agents whose containers are already interconnected.

Therefore, to permit communication between agents belonging to different systems (i.e. whose agent containers are not interconnected) the distributed extension implemented in eJason provides a new override for this internal action:

b) `.send(Address, Performative, Message, [Reply, Timeout])`

where the parameter **Address** is an annotated atom with the structure **AgentName[container(Container)]** and **Container** is the name of the container in which the agent with symbolic name **AgentName** may run.

The guarantees² provided by the aforementioned internal actions differ:

- If the execution of **a)** succeeds, the membership of the receiver to the system is guaranteed, i.e. the address of the receiver can be resolved. However, it does not ensure that the receiving agent actually receives the message or that it considers the message socially acceptable (i.e. considered suitable for processing, cf. not socially acceptable messages are automatically discarded and do not generate any event, see [Bordini et al., 2007]) nor that it has a plan which can be triggered by the reception of the message. The execution fails if there is no agent in the system whose symbolic name is **AgentName**.

²The reader is encouraged to read the content at <http://www.erlang.org/faq/academic.html#id54296> to get a feel of how useless and inefficient the imposition of strong guarantees on message delivery can be for distributed systems.

- The execution of **b)** depends on the value of the variable **Container** of the structure **Address**. The possible cases are:
 - If **Container** is an agent container that already belongs to the system, the guarantees are the same of **a)**. Note that, in this case, the execution is independent to whether the agent **AgentName** runs in the container with name **Container** or not.
 - If **Container** is an agent container that does not belong to the system, a merge operation is attempted (as if `.connect_to(Container)` had been invoked). If the merge operation is not successfully completed, the execution fails. Otherwise, the guarantees are those of **a)**. Note that even if the execution of the internal action fails (if **AgentName** does not correspond to any agent in the system resulting from the merge), a merge operation successfully completed is not rolled back.

Examples of the use of this internal action can be found in Section 6.2, e.g. in Figures 6.6 and 6.7.

Agent Creation and Destruction

Agents can *create* agents in a named container using the internal action:

- `.create_agent(AgentName|Address,Source,[InitialBeliefs])`

where the parameter in brackets is optional. This internal action is already provided by the Java implementation of Jason. It only allows **AgentName** to be an atom and the new agent is created in the same container as the agent executing the action.

The interpreter of eJason extends the capabilities of this internal action. If a **Container** is provided, using an **Address** structure, the new agent is created in the container specified, which is not necessarily the same as the one that hosts the agent executing the internal action. If **AgentName** is a free variable, the runtime system provides a (unique) name for the agent and binds this variable to it. As in [Bordini et al., 2007], the parameter **Source** indicates the implementation of the agent (its plans, initial goals and initial beliefs). Finally, **InitialBeliefs** is a list of beliefs that should be added to the set of initial beliefs of the new agent upon creation. This internal action succeeds if (1) the named container exists in the system, and (2) there is no other agent with symbolic name **AgentName** already in the system, and (3) **Source** correctly identifies an agent implementation.

An agent can also explicitly *kill* other agents or even itself. The following internal action allows this:

- `.kill_agent(AgentName).`

This internal action already exists in Jason to kill agents within the same container. However, eJason allows an agent to kill another agent in any container. Note that the name of the agent to kill suffices, as distribution is transparent. The execution of this internal action does not fail, even if the agent to terminate does not exist.

Examples of the use of these internal actions can be found in Section 6.2.2 in Figure 6.6.

Agent Discovery

Any agent can check whether some other agent with name `AgentName` belongs to its system executing the internal action:

- `.find_agent(AgentName).`

The execution of this action fails if `AgentName` does not identify an agent in the system or if `AgentName` is unbound.

As a matter of example, the following plan can be used by an agent `alice` to check that some agent `bob` exists before trying to send a message to it:

```
.send_if_exists: find_agent(bob) <-
    .send(bob, tell, greetings("Hi there!")).
```

Container Name Discovery

An agent can *discover* the name of its own container by executing the internal action

- `.my_container(Var)`

If the variable `Var` is unbound, the execution of the internal action does not fail and, as a result, `Var` is bound to the name of the container of the agent. If `Var` is bound to any value different from the name of the container of the agent executing it, the internal action fails, otherwise it succeeds.

As a matter of example, the following plan can be used to print (in the standard output) the name of the container where an agent runs:

```
.print_container <-
  .my_container(ContName); //binds the name to var ContName
  .print("My container is: ", ContName).
```

An agent can *discover* the container in which some other agent is running using the internal action:

- `.container_of_agent(AgentName, Container)`

This internal action binds the variable `Container` to the name of the container of the agent `AgentName`. This action fails if `AgentName` does not belong to the system or if `Container` is already bound to some value different from the name of the container. As a matter of example, the following plan can be used by an agent `alice` to print (in the standard output) the name of the container where the agent `bob` runs:

```
.print_container_another_agent <-
  .container_of_agent(bob, ContName);
  //binds the name to var ContName
  .print("Bob runs in container: ", ContName).
```

4.2 Fault Tolerance

Multi-agent systems are susceptible to the same kind of faults than any distributed system. The Jason programming language provides low level fault-tolerant mechanisms that enable an agent to retry a procedure that fails at some step but lacks mechanisms to recover from the failure of the agent itself. In contrast, eJason introduces constructs that enable the programmer to implement mechanisms to detect and recover from failures on the agents that compose the system, offering a wide variety of alternatives on how to react to such failures.

This work describes a fault as an event that arises at some component of the system and which threatens the proper functioning of such system. Faults can propagate among the components of a system and, in the worst case, provoke a system breakdown. Some instances of faults are the unexpected death of an agent, the isolation (preventing interaction) of a set of agents and the lack of response to incoming stimuli by an agent.

These errors may be originated by different causes, e.g. a broken physical link in the network connection, the shutdown of a host where some agent containers run or bugs in the source code that defines the behaviour of an agent. A classification

of the different faults, attending to the causes that provoke them, can be found in [Potiron et al., 2008].

The fault detection and fault recovery techniques that are present in this work disregard the underlying causes of failure. Instead, these techniques focus on the observable effects of faults:

- The death of an agent.
- The attempt of interaction with an agent that does not belong to the system.
- The (undesirable and unexpected) unresponsiveness of an agent.
- The termination of the connection between two agents.

It can be argued whether some of the observable events above are actually faults. For instance, the death of an agent *a* may be programmed to occur at some point as part of its behaviour, which is not a fault itself. However, it can generate a deadlock in some other agent *b* that has started some interaction protocol and is expecting a response from it. This deadlock can be avoided, e.g. allowing *b* to be aware of the death of *a*. The fault tolerance techniques discussed next not only address the recovery from faults by the components in the system (e.g. restarting *a*), but prevent the propagation of these faults between different components (e.g. deadlocking *b*).

In order to cope with the different observable faults, eJason provides two different mechanisms, namely *monitoring* and *supervision*. They follow citizen approach, described in Section 1.2, and are inspired by the homonym fault tolerance mechanisms provided by Erlang.

4.2.1 Monitoring Mechanism

The agents in a multi-agent system, despite being autonomous, are not isolated from each other. The interaction between agents is a recurrent phenomenon. Agents interact (e.g. cooperate, negotiate, provide a service) as part of the execution of their plans (intentions) to achieve their goals. Therefore, a fault in one of the agents involved in some interaction can spread among all other agents interacting with it. In order to avoid that, eJason introduces a monitoring mechanism that allows an agent *a* to be aware of the execution state of any other agent *b* in the system, following a citizen approach. Ideally, *a* interacts or expects some future interaction with *b*.

The monitoring mechanism proposed does not inform all the agents about faults in each other by default. Instead, a (monitoring) agent interested in the state of another (monitored) agent, with name **AgentName**, must explicitly request to be notified when the status of the monitored agent worsens. This phenomenon is referred to as *monitoring relation*. Note that a monitoring relation just requires the monitoring agent to know the unique identifier of the monitored agent, i.e. its name.

Once a monitoring relation is established, the monitoring agent will be informed about the following changes in the execution state of the monitored agent:

- 1) The monitored agent with the symbolic name **AgentName** has been restarted, i.e. its supervisor agent has stopped and started it anew (see Section 4.2.2 for details on agent supervision). This information can be useful as the restarted agent may have lost part of its belief base and/or goals.
- 2) The monitored agent with the symbolic name **AgentName** has been revived, i.e. this agent has died but its supervisor agent has started it anew (see Section 4.2.2 for details on agent supervision). In a similar way to restarts, this information can be useful as the revived agent may have lost part of its belief base and/or goals.
- 3) The monitored agent with the symbolic name **AgentName** has stopped running, i.e. has died and it is not revived (i.e. it has no supervisor agent or the supervision policy maintained for it does not allow its revival).
- 4) The containers of the monitored and monitoring agent are no longer connected, e.g. caused by a network problem, or by a problem with the container host, or by a failure in the container itself, or due to some split operation. Therefore, the direct interaction between these agents is not possible: they are “unreachable” to each other.
- 5) The monitored agent does not belong to the system. There is no agent whose symbolic name is **AgentName** in the system at the moment of the request.
- 6) An agent with name **AgentName** has been added to the system (either newly created or as a result of a merge operation). This information can be useful, e.g. to allow an agent in need of a service from **AgentName** to be aware of the moment when it is available or even to resume an interaction interrupted by a split operation after a network failure. Note that the only guarantee is that an agent with **AgentName** has been added to the system. It may be the case that it were not the agent expected. Therefore, the programmer is responsible of ensuring that agent names are not reused.

The characteristics that define the monitoring mechanism implemented in eJason are:

- A monitoring agent can monitor several agents, while a monitored agent can be monitored by several monitors. Besides, there can only be one instance of each monitoring relation.
- A monitoring relation exists until the monitoring agent disables it, executing the proper action, or until the monitoring agent dies (restarts and revivals excluded).
- It is possible for an agent to receive more than one notification for the same monitoring relation, but never two consecutive ones of the same type (excepting restarts and revivals). Clearly the errors above are not mutually exclusive. For instance, an unreachable agent (disconnected) may be dead as well. However, after a disconnection the only observable change of state is a reconnection.

Note how these features of the monitors in eJason contrast with the monitors of Erlang. Erlang allows many instances of the same monitor, with a maximum of one notification per instance, hence not allowing their persistence after the death or disconnection of the agents involved.

Improvements over Erlang

The monitoring mechanism implemented in eJason provides the following improvements over that of Erlang:

- As there can only be *one instance of each eJason monitor*, the agents do not need to keep track of the identification (reference) of each monitor in case they need to disable them.
- The *monitors persist over agent deaths and disconnections* and, moreover, notify the (re)apparition of the monitored agent. They can then be used to implement complex interaction protocols that survive agent deaths and temporal disconnections. Recall that the main limitation is that the monitoring relation depends on the name of the monitored agent. Given the death or disconnection of a monitored agent with name **AgentName** and the later apparition of an agent using that same name, all the monitoring relations depending on

AgentName that have not been disabled will send a notification to the monitoring agents. It may be the case that this new **AgentName** agent is a completely different agent (i.e. has a different behaviour or is a different instance than the one existing when the monitoring relation was established). In a system where the reuse of names is possible, it is recommended to automatically disable the monitoring relations after the death or the disconnection of the monitored agent. Section 6.2.3 shows an example of usage of this feature.

- The *observational power is augmented* as not only agent deaths and disconnections are observable. A monitoring agent can detect if a monitored agent has been restarted or revived. This implies that it can rest assured that the restarted/revived agent maintains, at least, its plan base (i.e. its control logic).

Note that the monitoring mechanism can also be used to implement fault recovery. For instance, if an agent receives the notification about the death of one of its monitored agents, it can try to respawn it. However, such an approach is prone to errors. Instead, the supervision mechanism described in Section 4.2.2 should be used, as it provides high-level constructs for fault recovery.

Monitoring API

An agent can request the set up of a monitoring relation executing the internal action:

- `.monitor_agent(AgentName [, Configuration])` where the parameter **AgentName** is the symbolic name of the monitored agent. The parameter in brackets is optional. The variable **Configuration** is either the structure `demonitor(Options)` or `persist(Options)`. **Options** must be a list whose elements are a subset of `{ unknown_agent, dead_agent, restarted_agent, revived_agent and unreachable_agent }` or the atom `any`, equivalent to a list of all these five values. If the structure `demonitor(Options)` is used, the monitoring relation is automatically disabled after the notification of any of the values in **Options** (see below). If the structure `persist(Options)` is used, the monitoring relation persist *only* after the notification of any of the values in **Options**, being disabled otherwise. If **Configuration** is not given, its default value is `persist(any)`.

This action is executed by the monitoring agent. The execution of this internal action never fails.

Similarly, a monitoring relation can be disabled by the monitoring agent executing the action:

- `.demonitor_agent(AgentName)`. The execution of this action never fails.

The fault notifications related to a monitoring relation are sent by the runtime system to the monitoring agent. They are represented by a new belief of the form:

- `+agent_down(AgentName) [reason(RType)]` along with the corresponding belief addition event. The possible values of `RType` are `unknown_agent`, `dead_agent`, `restarted_agent`, `revived_agent` and `unreachable_agent`. Their meaning corresponds to the notifications 1) to 5) in Section 4.2.1, respectively.

Besides, the notification of the (re)apparition (notification 6) in Section 4.2.1) of a monitored agent is represented by the new belief:

- `+agent_up(AgentName)`

Figure 4.3: Sample eJason supervision tree

4.2.2 Supervision Mechanism

The implementation of strategies that enable fault-recovery is not a trivial task. There exists a wide variety of faults that must be identified and different corrective actions that can be taken to tackle them. The supervision mechanism described in this section facilitates the implementation of such strategies for fault-recovery.

It provides runtime system support to (easily) execute operations like reviving a dead agent, breaking a possibly infinite loop or implementing complex supervision behaviours à la Erlang [Erlang, 2013].

The use of the supervision mechanism enables the construction of supervision trees. Figure 4.3 depicts a sample supervision tree. The branch nodes of a supervision tree correspond to supervisor agents. An edge of a supervision tree is the counterpart of a *supervision relation* between a supervisor agent and a non-empty set of agents (namely supervised set). Establishing a supervision relation, the supervisor requests the runtime system to execute actions aimed at recovering the proper execution state of its supervised agents (according to some supervision policy).

The restrictions imposed on a supervision relation are the following:

- a) Each agent can have at most one supervisor agent at any given moment in time.
- b) A supervisor agent can supervise several sets of agents. These supervised sets must be disjoint.

Restrictions a) and b) impose that any supervised agent belongs to at most one supervised set. It removes the possibility of maintaining conflicting supervision policies for the same agent.

- c) The supervision relation is automatically disabled when the supervisor agent dies (revivals and restarts excluded) and when it is disconnected from all the agents in the supervised set. The agents that are disconnected from the supervisor are removed from the supervised set.
- d) An agent cannot be supervised by itself. It would be pointless, as the death of a supervisor agent terminates a supervision relation.

Apart from the faults identified by the monitoring mechanism, the supervision mechanism enables the identification of the *divergence* of an agent. The concept of divergence is borrowed from the classical literature of process algebras, like e.g. CSP [Hoare, 1978], where a divergent process executes an infinite set of hidden actions and, consequently, does not terminate. In eJason, it is considered that an agent diverges if it is unable to complete any further iteration of its reasoning cycle. This situation can be caused, e.g. by the execution of an infinite loop during the resolution of a rule in the context of a plan. The observable properties of an agent that diverges is that it is alive but totally unresponsive, i.e. it no longer

processes new incoming messages and does not answer to any queries. However, an agent divergent shall not be confused with neither an overload (i.e. when an agent must process many incoming requests and cannot answer them as quickly as expected) nor a situation where the agent intentionally avoids the communication with some other agent. Strictly, the divergence of an agent occurs when it is no longer completing any iteration of its reasoning cycle.

Supervision Actions

The actions that the runtime system of eJason carries out in order to enforce the supervision policy requested by the supervisor are the following:

- **Ping:** this action (similar to the traditional ping protocol used in computer networks to test the reachability of hosts) is used to identify whether a supervised agent diverges or is overloaded. As the beginning of each agent's reasoning cycle, all ping requests are answered in order to inform the runtime system that the agent does not diverge.
- **Unblock:** this action makes a supervised agent cancel the resolution of a test goal or plan context in place. It enables the supervised agent to avoid divergence when such resolution does not terminate in the time interval defined by the programmer.
- **Restart:** this action stops a running (alive) supervised agent and starts it anew i.e. the execution of this action causes the death of an agent and its subsequent creation. The agent restarted maintains its name, the initial beliefs that it was given during its creation and its monitoring/supervision relations.
- **Revive:** this action starts anew a dead agent. A revived agent maintains its name, the initial beliefs that it was given during its creation and its monitoring/supervision relations. In contrast to a restart, the execution of this action only causes the creation of the supervised agent as a reaction to its death, which has necessarily occurred before such execution. The revival of an agent can only occur as part of the application of a supervision policy. The existence of such policy disables the possible interference from other agents with regards to a name clash as long as the corresponding supervision relation exists.

All the actions above take place to comply with a **Supervision Policy**. The supervisor agent enforces a complex supervision policy for its supervised agents, following then a citizen approach as noted earlier in the document. This supervision policy defines the course of action of the eJason runtime system with regards to

both fault identification and recovery. Note that in the remainder of the chapter it is described that the supervisor agent executes these actions, while it is the eJason runtime (on behalf of the supervisor) who actually executes them. An example of a valid supervision policy is one that requires pinging all the supervised agents every 15 seconds and restarting any of them that fails to answer to a ping two consecutive times (as it is probably diverging). The language constructs that enable this behaviour are described at the end of this chapter.

High-level description of the Supervision Actions

This section provides a high-level description of how the runtime system of eJason provides the functionality described in Section 4.2.2. This description utilises the concept of signal as a convenience. A signal is an internal message sent between different components of the runtime system of eJason. This instrument helps describe and understand actions like “a supervisor agent pings a supervised agent”. A signal does not generate a new event or belief, but remains within the scope of the runtime system of eJason.

The execution of the different actions related to supervision requires the emission and processing of signals. These signals are processed by the different distributed components of the runtime system.

Considering a supervisor agent that enables a supervision policy over a supervised agent with name **AgentName**, their execution works as follows:

- **Ping:** when the supervisor executes this action, a **ping signal** is sent to the supervised agent **AgentName**. Ping signals are answered at the beginning of each iteration of the reasoning cycle of the supervised agent. Answering a ping signal amounts to sending a **pong signal** to the supervisor agent. The belief base of the supervised agent is not modified. Ping signals have a timeout of **PingTime** milliseconds associated to each of them. If a **ping_signal** is not responded by a **pong_signal** before **PingTime** milliseconds, then a **pang_signal** is sent to the supervisor agent.

The emission of ping signals is used to test the divergence of an agent. Recall that the ping signals are answered at the beginning of the iterations of the reasoning cycle. Therefore, if no pong signal is emitted (for a **PingTime** big enough), the supervised agent has not completed an iteration in at least **PingTime** milliseconds, i.e. it is suspected of divergence. Note that a diverging agent will *never* respond to a ping signal. Besides, an overloaded agent (e.g.

an agent with many incoming messages) answers to ping signals, as they are processed with preference with respect to the incoming messages.

- **Unblock:** when a supervisor agent invokes this action, the agent with name **AgentName** receives an **unblock signal**. This signal causes the runtime system to abort the current resolution of a plan context or test goal being executed by the agent **AgentName**. Unblocking the resolution of a plan context amounts to stop identifying applicable plans for the event processed in the current reasoning cycle. The standard execution of the reasoning cycle is resumed then, i.e. the option selection function is applied over the applicable plans already identified, if any. Unblocking a test goal amounts to making it fail as if no matching could be found.

If the supervised agent is not resolving a test goal or plan context, the unblock signal is ignored. Note that an unblock action modifies the default behaviour for goal resolution (i.e. it can cause the failure of a resolution that, provided enough time, could have been successfully executed) and should be used carefully (Section 6.2.5 shows a use case example).

The purpose of this action is enabling the system to overcome the divergence of an agent without restarting it (hence avoiding some possible loss of the agent's state like e.g. its belief base).

- **Restart:** when a supervisor agent invokes this action, the supervised agent receives a **restart signal**. This signal causes the agent to die. Then, the agent is started again using *the same Jason source code* (hence maintaining its initial plan base and beliefs) and keeping the initial beliefs provided to its last creation, i.e. maintaining the parameters used to create it.

Note that the agent does not preserve the information about its state (i.e. its belief, goal and intention bases) at the moment it died. Notice also that the supervision and monitoring relations are maintained.

The use of this action enables the system to recover from a situation where some agent stubbornly falls into divergence, even after using unblock signals. Such an agent is then *freshly started*.

- **Revive:** when a supervised agent dies for a reason other than a restart, a signal **death signal** is sent to the supervisor agent. It then computes whether the agent must be restarted. In such case, the supervised agent is

started again with the same conditions as a restart above. Note that the only observable difference with a restart is that the supervised agent has not died as a consequence of the reception of a restart signal.

This action enables the system to maintain the agents alive after their death. As shown in the example in Section 6.2.4, it also avoids possible race conditions derived from a name clash.

The use of supervision policies enables the definition of complex supervision behaviours in a declarative way. A supervision policy defines the behaviour of a supervisor agent with respect to a certain supervised set. A supervision policy is completely defined by four subpolicies along with a restart strategy:

- *Ping policy*: it defines the frequency, **Frequency**, with which the possible divergence of the supervised agent must be checked using ping signals and a maximum threshold, **MaxPings**, of unanswered pings. The supervisor sends a ping signal every **Frequency** milliseconds to its supervised agents. In the case that one of them fails to respond to more than **MaxPings** consecutive pings (i.e. the ping threshold is exceeded), the periodical ping stops and the supervisor agent executes its unblock policy. The periodical ping is resumed afterwards.
- *Unblock policy*: it specifies the maximum number, **MaxUnblocks**, of unblock operations that can be executed over an agent in the supervised set in a temporal threshold **UMaxTime** (namely the unblock threshold). If a ping policy is specified, an unblock operation is executed when the ping threshold is exceeded (over the agent that exceeded it), unless the unblock threshold has been exceeded as well. If the unblock threshold is exceeded, the restart policy is applied.
- *Restart policy*: it specifies the maximum number of restarts **MaxRestarts** that can be executed on an agent in the set of supervised agents in a time span **RestartMaxTime**. When an agent exceeds the unblock threshold, this agent is restarted, unless the restart threshold is exceeded as well. If this threshold is exceeded, all the supervised agents in the set are killed and the supervision relation is terminated, hence disabling any revival policy.
- *Revival policy*: it specifies the maximum number of revivals **MaxRevive** that can occur for an agent in the supervised set in a time span **ReviveMaxTime**. This policy can be used to maintain the agents alive when they are killed or

die unexpectedly. If the revival threshold is exceeded, all the supervised agents in the set are killed and the supervision relation is disabled.

- *Restart strategy*: it defines a dependency, `RestartStrategy`, between the agents belonging to the same supervised set. It is applied anytime that an agent `Agent` in the supervised set is restarted or revived. Its possible values are similar to those of the Erlang supervisor [Erlang, 2013]. The strategy `one_for_one` states that only the agent `Agent` is started/revived. The strategy `rest_for_one` implies that the agents that appear after `Agent` in the list of supervised agents (the supervised set is represented by a list of agents) are restarted as well. Finally, the strategy `one_for_all` states that all the agents in the supervised set are restarted. Note that these restarts may exceed the corresponding restart threshold.

As a matter of example, consider a system where a supervision relation exists such that the supervisor agent `s` and the supervised set is composed by the agents `ag1`, `ag2` and `ag3` (in this order). The supervisor agent must check every fifteen seconds whether any of the supervised agents diverges or not. Anytime that an agent is suspected of divergence an unblock operation must be sent to it, with a maximum of 3 such operations in ten minutes. An agent can only be restarted once per minute when it exceeds this unblock threshold. Besides, anytime that an agent dies, it must be revived up to a maximum of 10 revivals every minute. In this example the functioning of the supervised agents depends on each other, whenever one is restarted or revived, all others must be restarted as well. In order to implement this kind of policy it suffices to specify a supervision policy such that:

- The ping policy has values: `Frequency` = 15000 (15 seconds) and `MaxPings` = 0 (any failed ping requires an unblock).
- The unblock policy has values: `MaxUnblocks` = 3 and `UMaxTime` = 600000 (10 minutes).
- The restart policy has values: `MaxRestarts` = 1 and `RMaxTime` = 60000 (1 minute).
- The revival policy has values: `MaxRevive` = 10 and `ReviveMaxTime` = 60000 (1 minute).
- The restart strategy is `one_for_all`.

The API that enables the use of the different features of the supervision mechanism can be found in 4.2.2. Some practical uses of this supervision mechanism are given in Section 6.2.

Improvements over Erlang

Although the supervision mechanism proposed is inspired by that of Erlang [Erlang, 2013], its capabilities are superior in several ways. These capabilities correspond to an agent-oriented way of programming in contrast to the process-oriented perspective of Erlang. The improvements introduced by the supervision mechanism of eJason are:

- A supervisor not only detects the death of its supervised agents. If an agent is alive but underperforming (overloaded) or blocked (diverging) the supervision detects (using pings) and corrects this situation.
- Agent termination and restart are not the only possible corrective actions. The supervisor unblocks diverging agents. Unblocking an agent is far less drastic than restarting it, as the agent maintains its state (e.g. its beliefs and goals).
- The introduction of the concept of revival of an agent. This operation is supported by the runtime system and provides the strong guarantee that, if the agent is revived, it maintains its name. It avoids race conditions derived from name clashes. An example of the potential use of this feature is shown in Section 6.2.4.
- It is not mandatory for the supervisor agent to create all the supervised agents. This way, the supervised agents do not need to be specifically designed to belong to a supervision tree, i.e. any agent can be supervised without any impositions to its behaviour (cf. supervised processes in an Erlang supervision tree must adhere to the requirements of the supervisor behaviour).
- The eJason supervision trees are more dynamic than the Erlang ones:
 - A supervisor agent can maintain several supervision policies for different sets of supervised agents. Besides, the supervisor of an agent and/or its supervision policy may vary over time.
 - The supervisor can maintain a supervision policy chosen from a variety of predefined supervision behaviours or provide its own behaviour.
 - The supervisor can supervise agents not created by itself and does not need to possess a description of their behaviour (a so-called child specification).
 - The supervised agents do not die if the supervisor dies. This strong dependence between the supervisor and its supervised agents is not introduced.

Supervision API

A supervision relation is always set up by the supervisor agent. This agent executes the internal action:

```
.supervise_agents(SupervisedSet,[supervision_policy(Options)]).
```

The possible values for `SupervisedSet` are:

- If it is an atom `AgentName`, a supervision relation between the agent executing the action and the agent with name `AgentName` is established. The execution of this action will fail if `AgentName` is a free variable, if this name does not correspond to any agent in the system or if the agent `AgentName` is already supervised by any agent.
- If it is a list `List` of agent names, the same supervision policy is applied to all the agents in the list as they belong to the same supervised set. The execution of this internal action never fails.

The second parameter is optional and must be a structure that contains a list of options. These options specify the supervision policy that will be executed. This policy is defined by the elements of the list:

- `ping(Frequency,PingTime,MaxPings)` defines the *ping policy*, i.e. the frequency with which the possible divergence or overload of the supervised agents will be checked using ping signals. The possible values of the structure are: a non-negative integer `Frequency`, a non-negative integer `PingTime` and a non-negative integer `MaxPings`.
- `unblock(MaxUnblocks,UMaxTime)` specifies the *unblock policy*. The arguments in the structure are: a non-negative integer `MaxUnblocks` and a non-negative integer `UMaxTime`. `UMaxTime` can also be the atom *infinity*. This policy allows the execution of a maximum of `MaxUnblocks` unblock operations in `UMaxTime` milliseconds (for the same agent).
- `restart(MaxRestarts,RMaxTime)` specifies the *restart policy*. The elements in the structure are: a non-negative integer `MaxRestarts` and a non-negative integer `RMaxTime`. `RMaxTime` can also be the atom *infinity*. This policy allows the execution of a maximum of `MaxRestarts` restart operations (due to a possible divergence) in `RMaxTime` milliseconds. If this threshold is exceeded, the agent is killed and the supervision relation is terminated, hence disabling any revival policy.

- `revive(MaxRevive,ReviveMaxTime)` specifies the *revival policy*. The elements in the structure are: a non-negative integer `MaxRevive` and a non-negative integer `ReviveMaxTime`. `ReviveMaxTime` can also be the atom *infinity*. This policy allows a maximum of `MaxRevive` restarts (due to the death of the agent) in `ReviveMaxTime` milliseconds. If this limit is exceeded, the supervision relation is terminated and the agent is not revived.
- `no_ping` specifies the absence of all three ping, unblock and restart policies. Therefore, the divergence of `AgentName` will not be periodically checked. Notice that, without a ping policy, its threshold is never exceeded, hence the other two can never be executed.
- `unblock(never)` is equivalent to `unblock(0,infinity)`. It specifies that no unblock signal shall be sent automatically. Conversely, `unblock(always)`, which is equivalent to `unblock(1,0)`, specifies that unblock signals shall always be sent when the threshold of the ping policy has been exceeded. Note that, the latter disables any the restart policy, as the unblock threshold cannot be exceeded.
- `restart(never)` is equivalent to `restart(0,infinity)`. It specifies that no restart signal shall be sent, i.e. the supervised agent is never restarted. Conversely, `restart(always)`, which is equivalent to `restart(1,0)`, specifies that restart signals shall always be sent when the threshold of the unblock policy has been exceeded i.e. the supervised agent is always restarted when the unblock policy does not resolve its divergence.
- `revive(never)` is equivalent to `revive(0,infinity)`. It specifies that the supervised agent shall never be revived when it dies. Conversely, `revive(always)`, equivalent to `revive(1,0)`, specifies that the supervised agent is always restarted when it dies. Note that this death does not include the case where the supervisor terminates an agent because it has exceeded the limit imposed by its restart policy.
- `strategy(Strategy)` specifies the *restart strategy*. The possible values for the variable `Strategy` are the atoms `one_for_one`, `one_for_all` and `rest_for_one`.
- The default values for the different policies as well as for the restart strategy are the following:
 - If no ping, unblock or restart policy is specified, the default option is `no_ping`.

- If at least one ping, unblock or restart policy is specified, the default values for the policies not specified are: `ping(15000,2000,1)`, `unblock(3,infinity)` and `restart(always)`.
- The default revival policy is `revive(never)`.
- The default restart strategy is `strategy(one_for_one)`.

Finally, note that if more than one policy of the same type is included, it is only considered the one closer to the head of the list, ignoring the rest.

Examples of the usage of this internal action are described in Section 6.2.

A supervision relation can be terminated by the supervisor agent by executing the internal action:

- `.unsupervise_agents(Agents)`

The parameter **Agents** corresponds to either the name of an agent or a list of agents. If these agents belong to some supervised set, they are removed from the set. The supervision relation is terminated if and only if there are no more agents left in the supervised set. The execution of this action always succeeds.

A supervision relation also terminates when the supervisor agent dies, when the supervised agent dies due to any reason other than a restart or revival, or when the supervisor and supervised agents get disconnected (i.e. supervision relations do not persist after system splits).

Chapter 5

FORMAL SEMANTICS

This chapter provides the formal semantics for the interpretation of eJason programs. This formal semantics of eJason builds on top of the formal semantics for Jason provided in [Bordini et al., 2007], which, in turn, is built on top of the formal semantics for AgentSpeak given in [Vieira et al., 2007]. The operational semantics of Jason agents in [Bordini et al., 2007], hence likewise in this thesis document, is given by the specification of a set of transition rules between agent configurations (see Section 5.1.2), i.e. for the evolution of the state of each single agent within the system. The operational semantics defined for eJason is also on a higher level, i.e., it provides a set of transition rules for the evolution of the whole multi-agent system. The formalisation of an eJason multi-agent system includes the formal definition of (1) a distributed multi-agent system, and (2) the fault-tolerant mechanisms considered: monitoring and supervision, and the system events observable by the fault-tolerant mechanisms. The contents of this chapters are summarised next.

Section 5.1 provides the semantics for the reasoning cycle of eJason agents. This semantics specifies the expected behaviour of each individual eJason agent.

Section 5.2 formally describes the distribution and fault-tolerant mechanisms of eJason. This formalisation includes the actions that enable the use of such mechanisms, the dynamics of the multi-agent system and the aforementioned mechanisms themselves. The semantics of these mechanisms determines how the multi-agent system evolves.

Section 5.3 enumerates a series of relevant properties of the eJason multi-agent systems whose validity is shown as proof for the corresponding theorems.

5.1 Agent Evolution: Reasoning Cycle

This section provides a formal semantics to the reasoning cycle that determines the behaviour of the agents running under the runtime environment of eJason. The semantics of the reasoning cycle of eJason subsumes that of Jason, hence being equivalent for the most part (i.e. excepting the semantics of the new operators and the critical sections introduced in Chapter 3). In order to help identify the contributions of this thesis work, these are addressed explicitly throughout this section (note that the contents of the subsequent sections is completely novel). First, it introduces the notation utilised throughout the present chapter. This notation includes a description of the grammar that defines the eJason programming language. Then, it presents the different tuples that represent the state (namely agent configuration) of each eJason agent. Finally, it enumerates the different transition rules that specify all the possible transitions between agent states.

5.1.1 Accepted Programs

Any eJason program is implemented by the language defined by a grammar that slightly differs from that of Jason (see Section 2.1.3) in order to introduce the new operators and critical sections. The aforementioned eJason grammar is the following (a simplification, for the sake of readability):

$$\begin{aligned}
ag &::= bs \ ps \\
bs &::= b_1 \ \dots \ b_n & (n \geq 0) \\
ps &::= p_1 \ \dots \ p_n & (n \geq 1) \\
p &::= te : ct \leftarrow h \\
te &::= +at \mid -at \mid +g \mid -g \\
ct &::= ct_1 \mid \top \\
ct_1 &::= at \mid \neg at \mid ct_1 \wedge ct_1 \\
h &::= h_1; \top \mid \top \\
h_1 &::= a \mid g \mid u \mid h_1; h_1 \mid [[h_1]] \\
at &::= P(t_1, \dots, t_n)[s_1, \dots, s_m] & (n \geq 0, m \geq 0) \\
s &::= at \\
a &::= A(t_1, \dots, t_n) & (n \geq 0) \\
g &::= !at \mid !!at \mid ?at \mid ??at \\
u &::= +b \mid -at
\end{aligned}$$

This grammar represents an agent as ag , composed by a set of beliefs bs and plans ps . Each belief, b , is a ground atomic formula, i.e. a possibly annotated

predicate which does not contain any variable. Conversely, an atomic formula, at , represents a possibly annotated predicate which may either be a variable itself or contain variables in some of its arguments, t_i , and/or annotations, s_i , but not only in its predicate functor P (i.e. $V(t_1, \dots, t_n)$, where V is an unbound variable, is not a valid atomic formula). Each individual plan, p , is composed by a triggering event, te , a context, ct , and a plan body, h , that may be empty ($h = \top$). The context of a plan is represented by a conjunction of literals (where a literal is either at or $\neg at$). Each instruction h_1 in a plan body may either be an action to execute, a (with functor A and arguments t_i), an addition or deletion of a goal, g and g' , or a belief update, u . Besides, a set of consecutive instructions, $h_1; \dots; h_n$, in a plan body may be enclosed in a critical section, $[[h_1; \dots; h_n]]$. Finally, the annotations s of the different atomic formulas are themselves possibly annotated predicates (i.e. annotations can be nested).

5.1.2 Agent Configurations

An agent configuration represents the state of one agent and is defined by the tuple $(ag, name, C, M, T, s)$. The element ag is the agent program composed by the set of beliefs bs and the set of plans ps . The initial state of a program, denoted by ag_0 , is the initial state of the agent, i.e. its initial beliefs bs_0 and plans ps_0 at the moment the agent was created. The element $name$ is the name of the agent. Note that the element $name$ is not part of the definition for agent configurations provided in [Bordini et al., 2007]. The element $name$ has been added to enable the addressing of agents by their unique name.

Let C be the agent's circumstance, a tuple (I, E, A) . The set of intentions $I = \{i, i', \dots\}$, where each intention i is a stack of partially instantiated plans (i.e. $i = [te_1 : ct_1 \leftarrow h_1, \dots, te_n : ct_n \leftarrow h_n]$). The set of events $E = \{(te, i), (te', i'), \dots\}$ where each event pair (te, i) is composed by the triggering event te and the related intention i (which may be the empty intention $\top$, in the case of external events or the internal events from non-suspending goal additions). The set of actions A includes the actions to be performed in the environment by the corresponding architectural component.

Let M be a tuple (In, Out, SI) that represents several aspects of the communication. The message mailbox $In = \{(mid, sender, ilf, cnt)\}$ contains all the messages addressed to the agent but not yet processed by it. The element mid is a message identifier, $sender$ corresponds to the name of the agent that originally sent the message, ilf is the illocutionary force and cnt is the content of the message.

Analogously, the message outbox $Out = \{(mid, receiver, ilf, cnt)\}$ contains the messages pending to be sent (by the runtime system) to some agent named *receiver*. The set of suspended intentions $SI = \{(message, i, timeout)\}$ contains the intentions i whose execution is suspended until some reply is received. This reply corresponds to some message $message = (mid, receiver, ilf, cnt)$ and has a related timeout $timeout$ (which can be infinite).

Let T be a tuple $(R, Ap, \iota, \varepsilon, \rho)$ that contains the temporary information generated and processed during the different phases of each iteration of the agent's reasoning cycle: R and Ap are, respectively, the sets of relevant plans and applicable plans for the selected event ε , ι is the selected intention and ρ is the selected applicable plan.

Let $s \in \{ProcMsg, SelEv, RelPl, ApplPl, SelAppl, AddIM, SelInt, ExecInt, ClrInt\}$ be the current step of the agent's reasoning cycle (see Section 2.1.2). Its possible values correspond, respectively, to: processing a message from the mailbox, selecting one event from the set of events, computing the set of relevant plans for the selected event, computing the set of applicable plans, selecting one applicable plan (the intended means), adding the intended means to the set of intentions, selecting one intention to execute, executing the selected intention, and clearing an intention that has been completely executed (if applicable). These steps are written as **s*** as a convenience to indicate that a critical section is being executed. Therefore, s can also take values **RelPl***, **AppPl***, **SelAppl***, **AddIM***, **ExecInt*** and **ClrInt***.

5.1.3 Notational Conventions and Auxiliary Functions

In the remainder of the chapter, a series of notational conventions and auxiliary functions, aimed at enhancing the readability and concision of the semantics, have been adopted. These elements are the following:

- An expression T_{comp} is written in order to allude to certain component *comp* of some tuple T . For instance, to refer to the set of events E in an agent circumstance C , the expression C_E is used.
- To abbreviate long tuples where the values of some of its elements are unimportant the ellipsis is used. For instance, an agent configuration $(ag, name, C, M, T, s)$ is abbreviated as $(ag, name, \dots)$ if the values of the elements C, M, T and s are not necessary for the transition rule in which they appear.

- The expression $i=k[te : ct \leftarrow h]$ is used to distinguish the partially instantiated plan, $p = te : ct \leftarrow h$, on top of some intention i from the remaining stack of partially instantiated plans, k , below it.
- As it is described in [Bordini et al., 2007], a notion of logical consequence, $\models$, can be defined between the different atomic formulae in eJason. This logical consequence relies on the assumption of the existence of some procedure to compute the most general unifier, θ , of two literals. The logical consequence in eJason is then defined as follows:

Definition 1 An atomic formula $at_1 = P_1(t_1, \dots, t_n)[s_1, \dots, s_m]$, where $n \geq 0 \leq m$, is a logical consequence of a set of atomic formulae fs , written $fs \models at_1$, if and only if there exists $at_2 = P_2(u_1, \dots, u_n)[v_1, \dots, v_l] \in fs$ such that (i) $(P_1(t_1, \dots, t_n))\theta = P_2(u_1, \dots, u_n)$, and (ii) $\forall s_i \in \{s_1, \dots, s_m\}, \exists v_j \in \{v_1, \dots, v_l\}. s_i\theta = v_j$ for some most general unifier θ .

- The signature of the different agent-specific selection functions introduced in Section 2.1.2 is:
 - $S_\varepsilon :: \{(te, i)\} \rightarrow (te, i)$ for the *event selection function*
 - $S_o :: \{(p, \theta)\} \rightarrow (p, \theta)$ for the *option selection function*
 - $S_I :: \{i\} \rightarrow i$ for the *intention selection function*.

These functions remain immutable during an agent's lifetime, hence they are not included as elements of each agent configuration in order to prevent obscuring the readability of the different transition rules below.

- Two operations are defined on belief bases. The addition of a belief (a ground atomic formula) to a belief base, written $bs' = bs + b$, implies that bs' is as bs except that $bs' \models b$. The deletion of an atomic formula $at = P(t_1, \dots, t_n)[s_1, \dots, s_m]$ (which may not be ground), written $bs' = bs - at$, is somewhat more complex. This operation deletes all the beliefs matching the pattern represented by at in such a way that bs' is as bs except that $\forall \theta, bs \models at\theta. bs' \not\models at\theta$.
- A Boolean auxiliary function is used in order to test whether any given atomic formula, $at = P(t_1, \dots, t_n)[s_1, \dots, s_m]$, is annotated with a source construct (as it is mandatory for all the beliefs in the belief base). This function is defined as follows:

$$\text{has_source}(at) = \begin{cases} \text{true} & \text{if } \exists \theta, s_i, i \in [1, m]. s_i \models \text{source}(X)\theta \\ \text{false} & \text{otherwise} \end{cases}$$

- An auxiliary function is used to represent the substitution of each of the arguments in an atomic formula at by the anonymous variable “ $_$ ” (maintaining its arity). This function is used as a convenience to formalise the semantics of the execution of the operator “ $-+$ ”. This function is the following:

$\text{pattern}(P(t_1, \dots, t_n[s_1, \dots, s_m]) = P(_, \dots, _)(s_1, \dots, s_m)$ for any $n, m \geq 0$

- The function to compute the set $\{(p, \theta)\}$ of relevant plans, for some set of plans ps and a triggering event te , (i.e. those plans whose triggering event can be matched to te) is defined as follows:

$$\text{RelPlans}(ps, te) = \{(p, \theta) \mid p \in ps \text{ and } \theta \text{ is s.t. } \{te\} \models p_{te}\theta\}.$$

The application of this function relies on the definition of logical consequence for the triggering events of plans and events (which are atomic formulas prefixed by operators):

Definition 2 *The triggering event of a plan, p_{te} , is a logical consequence of the triggering event of some event (te, i) , written $te \models p_{te}$, if, and only if, (i) one of the following cases applies:*

$$\left. \begin{array}{ll} p_{te} = +at_1, & te = +at_2 \\ p_{te} = -at_1, & te = -at_2 \\ p_{te} = +!at_1, & te = +!at_2 \\ p_{te} = +?at_1, & te = +?at_2 \\ p_{te} = +??at_1, & te = +??at_2 \\ p_{te} = -!at_1, & te = -!at_2 \\ p_{te} = -?at_1, & te = -?at_2 \end{array} \right\} \text{ and (ii) } at_2 \models at_1$$

- The function that computes the set of applicable plans for some set R of relevant plans and an agent’s belief base bs (i.e. those relevant plans whose context is a logical consequence of the beliefs in bs) requires a further generalisation of the definition of logical consequence given above. This generalisation provides the notion of logical consequence for the logical connectives allowed in plan contexts from a set of beliefs (ground atomic formulas):

Definition 3 *The conjunction of literals that represents the context of a plan $p_{ct} = l_i \wedge \dots \wedge l_n$ (where l_i is either at or $\neg at$), is a logical consequence of a set of beliefs bs , written $bs \models p_{ct}$ if, and only if, for each literal l_i one of the following applies:*

- (i) $l_i = at$ and $bs \models at$
- (ii) $l_i = \neg at$ and $bs \not\models at$

The aforementioned function is then defined as follows:

$$\mathbf{App1Plans}(bs, R) = \{(p, \theta' \circ \theta) \mid (p, \theta) \in R \text{ and } \theta' \text{ is s.t. } bs \models p_{ct}\theta\theta'\}$$

The computation of all possible valuations θ above may be impossible in real-time (e.g. because that set is infinite). Therefore, an underapproximation that computes a, possibly empty, subset is utilised instead. Whether the whole set of valuations is computed or not is determined by the use of the unblocking mechanism. This underapproximation is defined as:

$$\mathbf{app1Plans}(bs, R) \subseteq (\mathbf{App1Plans}(bs, R) \cup \emptyset)$$

- In order to compute the set of substitutions, $\{\theta\}$, that determines whether some atomic formula at can be derived (i.e. is a logical consequence) from the agent's beliefs bs (as it is necessary when, e.g., resolving test goals), the following function is used:

$$\mathbf{Test}(bs, at) = \{\theta \mid bs \models at\theta\}$$

Similarly to the function **App1Plans** above, the computation of this set may be intractable. The following underapproximation is then used instead:

$$\mathbf{test}(bs, R) \subseteq (\mathbf{Test}(bs, R) \cup \emptyset)$$

- To represent the generation of a failure event for some event (te, i) , the auxiliary function **fail_event** (te, i) is used. This function is defined as follows:

$$\left. \begin{array}{l} \mathbf{fail_event}(+b, i) \\ \mathbf{fail_event}(-at, i) \\ \mathbf{fail_event}(+!at, i) \\ \mathbf{fail_event}(+?at, i) \\ \mathbf{fail_event}(+??at, i) \\ \mathbf{fail_event}(te, \top) \end{array} \right\} = \emptyset$$

$$\left. \begin{array}{l} \mathbf{fail_event}(-?at, k[te' : ct \leftarrow h]) \\ \mathbf{fail_event}(-!at, k[te' : ct \leftarrow h]) \end{array} \right\} = \mathbf{fail_event}(te', k)$$

5.1.4 Transitions of the Reasoning Cycle

The reasoning cycle that determines the behaviour of an eJason agent is determined by the transition rules described next. These transitions are graphically represented in Figure 5.1. Notice that the transition rules whose name is underlined do not belong to the semantics of Jason, but are introduced by eJason.

Figure 5.1: Labelled state transitions within a reasoning cycle. Labels omitted for readability: a) achvG11, achvG12, testG12, testG13; b) action, testG11, addBel, delBel, critExecInt2; c) critAddG12, critAction, critAddBel, critDelBel; d) critAddG11; e) clrInt1, clrInt3; f) critExecInt1; g) critAddIM; h) critRel3

Initial state

The initial configuration (at creation time) of an agent is determined by the program in its source code, *agCode*, and the name *agName* given to the agent. The creation of an agent is represented as the execution of a function `create(agName, agCode)` that returns an agent's initial configuration $(ag_0, agName, C, M, T, ProcMsg)$. Note that the reasoning cycle of an agent starts in the step **ProcMsg** and evolves from that state.

Process Message (ProcMsg)

The first transition of a reasoning cycle implies the processing of a message from the mailbox. The different transition rules triggered in order to process that message have been described in [Bordini et al., 2007, Vieira et al., 2007] and correspond to the different speech-act performatives. Their semantics remains unchanged and, due to their relatively high length, are not reproduced here. It suffices to consider that the processing of these messages may introduce or remove plans, goals and beliefs, along with their corresponding events, depending on the illocutionary force *ilf* of the message. This message processing is abstracted by the formula **processMsg** used in the rule *procMsg*.

$$\begin{array}{c}
 \text{(procMsg)} \frac{(mid, name, ilf, cnt) \in M_{In}}{(ag, name, C, M, T, ProcMsg) \Rightarrow (ag', name, C', M', T', SelEv)} \\
 \text{where:} \quad (ag', name, C', M', T') = \text{processMsg}((ag, name, C, M, T), \\
 (mid, ilf, cnt)) \\
 M'_{In} = M_{In} \setminus (mid, name, ilf, cnt)
 \end{array}$$

Select Event (SelEv)

The next transition implies the selection of an event from the set of events (rule *selEv1*). It amounts to the application of the event selection function S_ε to the set of events C_E . If that set is empty (rule *selEv2*), the next step of the reasoning cycle corresponds to the selection of an intention to execute.

$$\begin{array}{c}
 \text{(selEv1)} \frac{S_\varepsilon(C_E) = (te, i)}{(ag, name, C, M, T, SelEv) \Rightarrow (ag, name, C', M, T', RelPl)} \\
 \text{where:} \quad C'_E = C_E \setminus (te, i) \\
 T'_\varepsilon = (te, i)
 \end{array}$$

$$\text{(selEv2)} \frac{C_E = \emptyset}{(ag, name, C, M, T, SelEv) \Rightarrow (ag, name, C, M, T, SelInt)}$$

Relevant Plans (RelPl)

Given a selected event, T_ε , the set of relevant plans is computed and assigned to T_R (rule *rel1*). If there are no relevant plans, depending on the type of the triggering event te , different alternatives can be taken. If te corresponds to the addition of an achievement goal ($te = +at$) or the addition of a test goal using the operator “??” ($te = +??at$), the event is returned to the agent’s set of events and will be retried later (rule *rel2*). Otherwise, the corresponding failure event is generated and the reasoning cycle returns to its event selection phase (rule *rel3*). The transition rule *rel2* is introduced by eJason.

$$\text{(rel1)} \frac{T_\varepsilon = (te, i) \quad \text{RelPlans}(ag_{ps}, te) \neq \emptyset}{(ag, name, C, M, T, RelPl) \Rightarrow (ag, name, C, M, T', ApplPl)}$$

$$\text{where:} \quad T'_R = \text{RelPlans}(ag_{ps}, te)$$

$$\text{(rel2)} \frac{T_\varepsilon = (te, i) \quad te = \begin{cases} +!at \\ +??at \end{cases} \quad \text{RelPlans}(ag_{ps}, te) = \emptyset}{(ag, name, C, M, T, RelPl) \Rightarrow (ag, name, C', M, T, SelInt)}$$

$$\text{where:} \quad C'_E = C_E \cup T_\varepsilon$$

$$\text{(rel3)} \frac{T_\varepsilon = (te, i) \quad te \neq \begin{cases} +!at \\ +??at \end{cases} \quad \text{RelPlans}(ag_{ps}, te) = \emptyset}{(ag, name, C, M, T, RelPl) \Rightarrow (ag, name, C', M, T, SelEv)}$$

$$\text{where:} \quad C'_E = C_E \cup \text{fail_event}(te, i)$$

Applicable Plans (ApplPl)

Given a set of relevant plans, a set of applicable plans can be determined using the agent's belief base (rule *appl1*). If no applicable plan can be found, the alternatives are similar to those for the transition rules *rel2* (retry later) and *rel3* (drop the event and generate a failure event) above, here represented by transition rules *appl2* and *appl3* respectively.

$$(\text{appl1}) \frac{\text{applPlans}(ag_{bs}, T_R) \neq \emptyset}{(ag, name, C, M, T, ApplPl) \Rightarrow (ag, name, C, M, T', SelAppl)}$$

$$\text{where: } T'_{Ap} = \text{applPlans}(ag_{bs}, T_R)$$

$$(\text{appl2}) \frac{T_\varepsilon = (te, i) \quad te = \begin{cases} +!at \\ +??at \end{cases} \quad \text{applPlans}(ag_{bs}, T_R) = \emptyset}{(ag, name, C, M, T, ApplPl) \Rightarrow (ag, name, C', M, T, SelInt)}$$

$$\text{where: } C'_E = C_E \cup T_\varepsilon$$

$$(\text{appl3}) \frac{T_\varepsilon = (te, i) \quad te \neq \begin{cases} +!at \\ +??at \end{cases} \quad \text{applPlans}(ag_{bs}, T_R) = \emptyset}{(ag, name, C, M, T, ApplPl) \Rightarrow (ag, name, C', M, T, SelInt)}$$

$$\text{where: } C'_E = C_E \cup \text{fail_event}(te, i)$$

Select Applicable Plan (SelAppl)

The application of the option selection function S_o to the set of applicable plans T_{Ap} determines the selected plan T_ρ (rule *selAppl*).

$$(\text{selAppl}) \frac{S_o(T_{Ap}) = (p, \theta)}{(ag, name, C, M, T, SelAppl) \Rightarrow (ag, name, C, M, T', AddIM)}$$

$$\text{where: } T'_\rho = (p, \theta)$$

Add the Intended Means (AddIM)

The selected applicable plan, (p, θ) , defines a partially instantiated plan, $p\theta$, that, along with the selected event (te, i) , determines a new intention to be executed. Recall that this new intention (a stack of partially instantiated plans) contains only one element if the selected event did not have a related intention, i.e. $i = \top$ (rule *addIM1*). Otherwise, the new intention results by putting the selected applicable plan on top of the related intention (rule *addIM2*). Note that the transition rules *addIM1* and *addIM2* are equivalent to the rules *ExtEv* and *IntEv* in [Bordini et al., 2007], respectively. They are, however, renamed as it is considered in this work that their previous names can be somewhat misleading. An internal event may have an empty related intention (e.g. a belief addition/deletion or an achievement goal addition using the operator “!”), hence triggering rule *addIM1* (previously named *ExtEv*) in such case. An external event does always have an empty related intention.

$$\text{(addIM1)} \frac{T_\varepsilon = (te, \top) \quad T_\rho = (p, \theta)}{(ag, name, C, M, T, AddIM) \Rightarrow (ag, name, C', M, T, SelInt)}$$

$$\text{where:} \quad C'_I = C_I \cup [p\theta]$$

$$\text{(addIM2)} \frac{T_\varepsilon = (te, i) \quad T_\rho = (p, \theta)}{(ag, name, C, M, T, AddIM) \Rightarrow (ag, name, C', M, T, SelInt)}$$

$$\text{where:} \quad C'_I = C_I \cup i[p\theta]$$

Select Intention (SelInt)

The selection of the next intention to be executed, T_i , requires the application of the intention selection function S_I to the agent's set of intentions C_I (rule *selInt1*). If this set is empty, the current iteration of the reasoning cycle is considered completed and a new one is started (rule *selInt2*).

$$\text{(selInt1)} \frac{C_I \neq \emptyset \quad S_I(C_I) = i}{(ag, name, C, M, T, SelInt) \Rightarrow (ag, name, C, M, T', ExecInt)}$$

where: $T'_l = i$

$$\text{(selInt2)} \frac{C_I = \emptyset}{(ag, name, C, M, T, SelInt) \Rightarrow (ag, name, C, M, T, ProcMsg)}$$

Execute Intention (ExecInt)

The execution of the intended means in the selected intention, T_l , triggers a different transition rule depending on the next formula, f , in the partially instantiated plan on top of the stack ($T_l = i[te : ct \leftarrow f; h]$), to be executed:

- *Action*: the successful execution of an (internal or external) action act is symbolically represented by its inclusion into the set C_A (rule *action*). Recall that the actions in C_A are executed by the runtime system and may alter the environment (in the case of external actions). Besides, the execution of an action may vary the valuation, θ' , of the variables in the partially instantiated plan (symbolically represented as $\theta' = \text{execute}(act)$).

$$\text{(action)} \frac{T_l = i[te : ct \leftarrow act; h]}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag, name, C', M, T', ClrInt)}$$

where:

$$\begin{aligned} T'_l &= i \\ C'_A &= C_A \cup act \\ T'_l &= i[(te : ct \leftarrow h)\theta'] \quad \text{s.t.} \quad \theta' = \text{execute}(act) \\ C'_I &= C_I \setminus T_l \cup T'_l \end{aligned}$$

- *Achievement goal*: the addition of a new achievement goal, at , requires the generation of a new internal event. If the operator “!” is used, the new internal event contains the selected intention, T_l , as related intention (rule *achvGl1*). Note that the execution of T_l is suspended until an applicable plan for the goal at is fully executed (in subsequent iterations of the reasoning cycle). If the operator used is “!”, the new internal event has an empty related intention and the execution of T_l is not suspended (rule *achvGl2*).

$$\text{(achvGl1)} \frac{T_l = i[te : ct \leftarrow !at; h]}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag, name, C', M, T, ProcMsg)}$$

$$\text{where:} \quad \begin{aligned} C'_E &= C_E \cup (+!at, T_l) \\ C'_I &= C_I \setminus T_l \end{aligned}$$

$$\text{(achvGl2)} \frac{T_l = i[te : ct \leftarrow !at; h]}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag, name, C', M, T, ClrInt)}$$

$$\text{where:} \quad C'_E = C_E \cup (+!at, \top)$$

- *Test goal*: the execution of a test goal, at , amounts to attempting to find a substitution, θ , for the variables in the structure at such that it is a logical consequence of the beliefs in the agent's belief base, ag_{bs} . If a substitution can be found (represented as $\theta \in \mathbf{test}(ag_{bs}, at)$), it is applied to the variables in the selected intention (rule *testGl1*). Otherwise, a new internal event, similar to the one in rule *achvGl1* for achievement goals, is added to the agent's set of events. This event includes the selected intention T_l as related intention. The triggering event, te , varies depending on the operator, “?” or “??”, used to add the test goal (rules *testGl2* and *testGl3* respectively). Transition rules *testGl2* and *testGl3* are introduced by eJason.

$$\text{(testGl1)} \frac{T_l = \begin{cases} i[te : ct \leftarrow ?at; h] & \theta \in \mathbf{test}(ag_{bs}, at) \\ i[te : ct \leftarrow ??at; h] \end{cases}}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag, name, C', M, T', ClrInt)}$$

$$\text{where:} \quad \begin{aligned} T'_l &= i[(te : ct \leftarrow h)\theta] \\ C'_I &= C_I \setminus T_l \cup T'_l \end{aligned}$$

$$\text{(testGl2)} \frac{T_l = i[te : ct \leftarrow ?at; h] \quad \mathbf{test}(ag_{bs}, at) = \emptyset}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag, name, C', M, T, ProcMsg)}$$

$$\text{where:} \quad \begin{aligned} C'_E &= C_E \cup (+?at, T_l) \\ C'_I &= C_I \setminus T_l \end{aligned}$$

$$\text{(testGl3)} \frac{T_i = i[te : ct \leftarrow ??at; h] \quad \mathbf{test}(ag_{bs}, at) = \emptyset}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag, name, C', M, T, ProcMsg)}$$

$$\begin{aligned} \text{where:} \quad C'_E &= C_E \cup (+??at, T_i) \\ C'_I &= C_I \setminus T_i \end{aligned}$$

- *Belief*: the addition of a belief b and the deletion of the set of beliefs that match the pattern at modify the agent's belief base and, in some cases, generate a new internal event. If b or at are not annotated with any source, the annotation $source(self)$ is added to the new/removed belief (rules *addBel* and *delBel* respectively). The semantics of the operator “-+” are specified by rule *delAddBel*. Note that b and at are used to emphasise that any belief added must be ground, while a belief deletion may contain free variables.

$$\text{(addBel)} \frac{T_i = i[te : ct \leftarrow +b; h]}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag', name, C', M, T', ClrInt)}$$

where:

$$\begin{aligned} belief &= \begin{cases} b & \text{if } \mathbf{has_source}(b) \\ b[source(self)] & \text{otherwise} \end{cases} \\ ag'_{bs} &= ag_{bs} + belief \\ C'_E &= C_E \cup \begin{cases} (+belief, \top) & \text{if } ag_{bs} \not\models belief \\ \emptyset & \text{otherwise} \end{cases} \\ T'_i &= i[te : ct \leftarrow h] \\ C'_I &= C_I \setminus T_i \cup T'_i \end{aligned}$$

$$\text{(delBel)} \frac{T_i = i[te : ct \leftarrow -at; h]}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag', name, C', M, T', ClrInt)}$$

where:

$$\begin{aligned} pattern &= \begin{cases} at & \text{if } \mathbf{has_source}(at) \\ at[source(self)] & \text{otherwise} \end{cases} \\ ag'_{bs} &= ag_{bs} - pattern \\ C'_E &= C_E \cup \{ (-pattern\theta, \top) \mid \theta \in \mathbf{test}(ag_{bs}, at) \} \\ T'_i &= i[te : ct \leftarrow h] \end{aligned}$$

$$C'_I = C_I \setminus T_i \cup T'_i$$

$$\text{(delAddBel)} \frac{T_i = i[te : ct \leftarrow - + b; h]}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag', name, C', M, T', ClrInt)}$$

where:

$$\begin{aligned} pattern &= \begin{cases} pattern(b) & \text{if } has_source(b) \\ pattern(b)[source(self)] & \text{otherwise} \end{cases} \\ belief &= \begin{cases} b & \text{if } has_source(b) \\ b[source(self)] & \text{otherwise} \end{cases} \\ ag'_{bs} &= (ag_{bs} - pattern) + belief \\ C'_E &= C_E \cup \{ (-pattern\theta, \top) \mid \theta \in test(ag_{bs}, at) \} \cup \\ (+belief, \top) \\ T'_i &= i[te : ct \leftarrow h] \\ C'_I &= C_I \setminus T_i \cup T'_i \end{aligned}$$

Clear Intention (ClrInt)

Finally, after the execution of a formula from the selected intention, it must be checked whether the partially instantiated plan on top of it has been fully executed (i.e. $T_i = i[te : ct \leftarrow \top]$). In that case, if that partially instantiated plan was the only one remaining in the selected intention ($i = []$) the whole intention is removed (rule *clrInt1*). If the plan is fully executed but there are more partially instantiated plans ($i = k[te' : ct' \leftarrow g; h]$) the next one ($[te' : ct' \leftarrow g; h]$) is updated with the variable substitutions, θ , for the variables in the triggering event te (of the plan just fully executed) and it is further instantiated (rule *clrInt2*). Otherwise, i.e., if the partially instantiated plan on top of the selected intention is not fully executed, the reasoning cycle starts a new iteration (rule *clrInt3*).

$$\text{(clrInt1)} \frac{T_i = [te : ct \leftarrow \top]}{(ag, name, C, M, T, ClrInt) \Rightarrow (ag, name, C', M, T, ProcMsg)}$$

where: $C'_I = C_I \setminus T_i$

$$\text{(clrInt2)} \frac{T_i = i[te : ct \leftarrow \top]}{(ag, name, C, M, T, ClrInt) \Rightarrow (ag, name, C', M, T, ClrInt)}$$

where:

$$\begin{aligned} C'_I &= (C_I \setminus T_i) \cup k[(te' : ct' \leftarrow h)\theta] \\ i &= k[te' : ct' \leftarrow g; h] \\ \theta &\text{ is such that } g\theta = te \end{aligned}$$

$$\text{(clrInt3)} \frac{T_i = i[te : ct \leftarrow h] \quad h \neq \top}{(ag, name, C, M, T, ClrInt) \Rightarrow (ag, name, C, M, T, ProcMsg)}$$

Critical Sections

As described in Section 3.1.2, the formulas in a plan body enclosed within a critical section are executed consecutively. During the execution of a critical section, the agent does not process incoming messages from other agents and its belief base is not modified by changes in the environment (i.e. perception is disabled). The following transition rules represent the steps in the agent's reasoning cycle necessary to execute a critical section. Recall that critical sections are introduced by eJason, hence all these transition rules are novel contributions of this thesis.

A critical section is detected during the intention execution phase (**ExecInt**) by the apparition of the delimiter "[[" on top of the plan body of the selected intention T_i . This detection implies a different execution of the reasoning cycle, here represented as the transition to the critical intention execution step (**ExecInt***, as specified by the rule *critExecInt1*). This critical section will only be exited either when it is completely executed (rule *critExecInt2*) or when the plan, to which the critical section belongs, fails (rules *critRel3* and *critAppl3*).

$$\text{(critExecInt1)} \frac{T_i = i[te : ct \leftarrow [[: h]]]}{(ag, name, C, M, T, ExecInt) \Rightarrow (ag, name, C, M, T, ExecInt*)}$$

$$\text{(critExecInt2)} \frac{T_i = i[te : ct \leftarrow]:] h]}{(ag, name, C, M, T, ExecInt*) \Rightarrow (ag, name, C, M, T, ClrInt)}$$

Then, the different formulas are executed consecutively. In order to guarantee this consecutive execution, a special set of transition rules are triggered which guarantee that the currently selected intention, T_i , remains as such until the critical

section is exited. The different transitions depend on the kind of next formula to execute:

If a **goal addition**, $+g$, occurs within a critical section, the corresponding event, $(+g, i)$, is immediately processed (except in the case of a non-suspending achievement goal addition, $g = !at$, see rule *critAddGl2* below), i.e., it becomes the new selected event T_ε (rule *critAddGl1*). If both the set of relevant plans and applicable plans for such event are not empty (rules *critRel1* and *critAppl1* triggered), one of the applicable plans is selected (rule *critSelAppl*), and the corresponding intended means added for execution (rule *critAddIM*). Note that the new selected intention, T'_i , corresponds to the intended means for the goal addition event which, in turn, contains the former selected intention T_i that contains the critical section, hence not leaving this critical section. In the presence of plan failures, i.e., if either the set of relevant plans or the set of applicable plans are empty (either rule *critRel2* or rule *critAppl2* is triggered), the corresponding failure event is generated. Note that these failure events may lead to exiting the critical section if the triggering event corresponds to the failure plan of a goal addition, $te = -g$, and the related intention $i = k[te' : ct' \leftarrow h]$ is such that $]] \in h$ (rules *critRel3* and *critAppl3*).

$$\text{(critAddGl1)} \frac{T_i = i[te : ct \leftarrow g; h] \quad g = \begin{cases} !at \\ ?at \\ ??at \end{cases}}{(ag, name, C, M, T, ExecInt*) \Rightarrow (ag, name, C', M, T', RelPl*)}$$

$$\text{where:} \quad \begin{aligned} T'_\varepsilon &= (+g, T_i) \\ C'_I &= C_I \setminus T_i \end{aligned}$$

$$\text{(critAddGl2)} \frac{T_i = i[te : ct \leftarrow !at; h]}{(ag, name, C, M, T, ExecInt*) \Rightarrow (ag, name, C', M, T, ClrInt*)}$$

$$\text{where:} \quad C'_E = C_E \cup (+!at, \top)$$

$$\text{(critRel1)} \frac{T_\varepsilon = (te, i) \quad \text{RelPlans}(ag_{ps}, te) \neq \emptyset}{(ag, name, C, M, T, RelPl*) \Rightarrow (ag, name, C, M, T', ApplPl*)}$$

where: $T'_R = \text{RelPlans}(ag_{ps}, te)$

$$\text{(critRel2)} \frac{T_\varepsilon = (te, i) \quad \text{RelPlans}(ag_{ps}, te) = \emptyset \quad (te \neq -g \vee i = k[te' : ct' \leftarrow h] \text{ s.t. } \llbracket \rrbracket \notin h)}{(ag, name, C, M, T, RelPl*) \Rightarrow (ag, name, C, M, T', RelPl*)}$$

where: $T'_\varepsilon = \text{fail_event}(te, i)$

$$\text{(critRel3)} \frac{T_\varepsilon = (te, i) \quad \text{RelPlans}(ag_{ps}, te) = \emptyset \quad te = -g \quad i = k[te' : ct' \leftarrow h] \text{ s.t. } \llbracket \rrbracket \in h}{(ag, name, C, M, T, RelPl*) \Rightarrow (ag, name, C', M, T, ProcMsg)}$$

where: $C'_E = C_E \cup \text{fail_event}(te, i)$

$$\text{(critAppl1)} \frac{\text{applPlans}(ag_{bs}, T_R) \neq \emptyset}{(ag, name, C, M, T, ApplPl*) \Rightarrow (ag, name, C, M, T', SelAppl*)}$$

where: $T'_{Ap} = \text{applPlans}(ag_{bs}, T_R)$

$$\text{(critAppl2)} \frac{T_\varepsilon = (te, i) \quad \text{applPlans}(ag_{bs}, T_R) = \emptyset \quad (te \neq -g \vee i = k[te' : ct' \leftarrow h] \text{ s.t. } \llbracket \rrbracket \notin h)}{(ag, name, C, M, T, ApplPl*) \Rightarrow (ag, name, C, M, T', RelPl*)}$$

where: $T'_\varepsilon = \text{fail_event}(te, i)$

$$\text{(critAppl3)} \frac{T_\varepsilon = (te, i) \quad \text{applPlans}(ag_{bs}, T_R) = \emptyset \quad te = -g \quad i = k[te' : ct' \leftarrow h] \text{ s.t. } \llbracket \rrbracket \in h}{(ag, name, C, M, T, ApplPl*) \Rightarrow (ag, name, C', M, T, ProcMsg)}$$

where: $C'_E = C_E \cup \text{fail_event}(te, i)$

$$\text{(critSelAppl)} \frac{S_o(T_{Ap}) = (p, \theta)}{(ag, name, C, M, T, SelAppl*) \Rightarrow (ag, name, C, M, T', AddIM*)}$$

$$\text{where:} \quad T'_\rho = (p, \theta)$$

$$\text{(critAddIM)} \frac{T_\varepsilon = (te, i) \quad T_\rho = (p, \theta)}{(ag, name, C, M, T, AddIM*) \Rightarrow (ag, name, C, M, T', ExecInt*)}$$

$$\text{where:} \quad T'_l = i[p\theta]$$

If a **belief update** or an **action** are included in a critical section, they are executed almost as usual. The only difference being that the next step of the reasoning cycle is **ClrInt*** instead of **ClrInt**.

$$\text{(critAction)} \frac{T_l = i[te : ct \leftarrow act; h]}{(ag, name, C, M, T, ExecInt*) \Rightarrow (ag, name, C', M, T', ClrInt*)}$$

$$\begin{aligned} \text{where:} \quad & C'_A = C_A \cup act \\ & T'_l = i[(te : ct \leftarrow h)\theta'] \quad \text{s.t.} \quad \theta' = execute(act) \\ & C'_I = C_I \setminus T_l \cup T'_l \end{aligned}$$

$$\text{(critAddBel)} \frac{T_l = i[te : ct \leftarrow +b; h]}{(ag, name, C, M, T, ExecInt*) \Rightarrow (ag', name, C', M, T', ClrInt*)}$$

where:

$$\begin{aligned} belief &= \begin{cases} b & \text{if } has_source(b) \\ b[source(self)] & \text{otherwise} \end{cases} \\ ag'_{bs} &= ag_{bs} + belief \\ C'_E &= C_E \cup \begin{cases} (+belief, \top) & \text{if } ag_{bs} \not\models belief \\ \emptyset & \text{otherwise} \end{cases} \\ T'_l &= i[te : ct \leftarrow h] \\ C'_I &= C_I \setminus T_l \cup T'_l \end{aligned}$$

$$\text{(critDelBel)} \frac{T_i = i[te : ct \leftarrow -at; h]}{(ag, name, C, M, T, ExecInt*) \Rightarrow (ag', name, C', M, T', ClrInt*)}$$

where:

$$\begin{aligned} pattern &= \begin{cases} at & \text{if } \text{has_source}(at) \\ at[source(self)] & \text{otherwise} \end{cases} \\ ag'_{bs} &= ag_{bs} - pattern \\ C'_E &= C_E \cup \{ (-pattern\theta, \top) \mid \theta \in \text{test}(ag_{bs}, at) \} \\ T'_i &= i[te : ct \leftarrow h] \\ C'_I &= C_I \setminus T_i \cup T'_i \end{aligned}$$

$$\text{(critDelAddBel)} \frac{T_i = i[te : ct \leftarrow -+b; h]}{(ag, name, C, M, T, ExecInt*) \Rightarrow (ag', name, C', M, T', ClrInt*)}$$

where:

$$\begin{aligned} pattern &= \begin{cases} pattern(b) & \text{if } \text{has_source}(b) \\ pattern(b)[source(self)] & \text{otherwise} \end{cases} \\ belief &= \begin{cases} b & \text{if } \text{has_source}(b) \\ b[source(self)] & \text{otherwise} \end{cases} \\ ag'_{bs} &= (ag_{bs} - pattern) + belief \\ C'_E &= C_E \cup \{ (-pattern\theta, \top) \mid \theta \in \text{test}(ag_{bs}, at) \} \cup \\ (+belief, \top) & \\ T'_i &= i[te : ct \leftarrow h] \\ C'_I &= C_I \setminus T_i \cup T'_i \end{aligned}$$

Finally, each time that a formula within the critical section is executed, a transition rule similar to the one of rule *ClrInt* is executed. The main difference is that the next step of the reasoning cycle is **ExecInt***, i.e. the same selected intention (possibly updated) is chosen for execution (rules *critClrInt1* and *critClrInt2*), hence not changing the focus of attention.

$$\text{(critClrInt1)} \frac{T_i = i[te : ct \leftarrow \top] \quad i \neq \top}{(ag, name, C, M, T, ClrInt*) \Rightarrow (ag, name, C, M, T', ClrInt*)}$$

$$\begin{aligned} \text{where:} \quad T'_i &= k[(te' : ct' \leftarrow h)\theta] \\ i &= k[te' : ct' \leftarrow g; h] \end{aligned}$$

θ is such that $g\theta = te$

$$\text{(critClrInt2)} \frac{T_i = i[te : ct \leftarrow h] \quad h \neq \top}{(ag, name, C, M, T, ClrInt*) \Rightarrow (ag, name, C, M, T, ExecInt*)}$$

5.2 Distribution and Fault Tolerance

This section provides a formal specification for the distribution and fault-tolerance mechanisms of eJason. It describes the notation utilised to represent a multi-agent system. The formalisation of the internal actions, that affect the distribution of the agents and that enable the use of the mechanisms for fault detection and recovery, is then provided. The evolution of the eJason multi-agent systems as different agent containers enter or leave the system (i.e. its dynamics, as well as a description of the different notifications corresponding to the monitoring mechanism, follow. Finally, the formalisation of the behaviour of the supervision mechanism is included.

5.2.1 eJason Multi-Agent Systems

Let a multi-agent system in eJason be a tuple $M = (Di, Mo, Su, SE, t)$ where Di represents agent distribution, Mon represents monitoring relations, Su represents supervision relations, SE represents system events and t represents the time of the system. Each of these sets are described in the remainder of this section.

Agent distribution (Di)

Let Di be a set of tuples $\{(container, Agents)\}$ that represents the distribution of the agents in different containers and hosts. Let $container$ be the name of an agent container (unique within the system). Let $Agents$ be the set of agent configurations (see Section 5.1.2) of all the agents running in the container named $container$. Recall that every agent belongs to one and only one container and has a name that is unique within the system. Thus, consider the two tuples $(container_1, Agents_1)$ and $(container_2, Agents_2)$ in Di with agents $(ag_1, ag_{01}, name_1, C_1, M_1, T_1, s_1) \in Agents_1$ and $(ag_2, ag_{02}, name_2, C_2, M_2, T_2, s_2) \in Agents_2$. Then $container_1 = container_2 \Rightarrow Agents_1 = Agents_2$ and $name_1 = name_2 \Rightarrow container_1 = container_2$.

Monitoring relations (Mo)

Let $Mo = \{(monitoring, monitored, Persist)\}$ be the set of monitoring relations in the MAS where each tuple represents a different monitoring relation. The elements *monitoring* and *monitored* correspond to the names of the monitoring and monitored agents in the relation, respectively. The element $Persist \subseteq \{\text{unknown, dead, restarted, revived, unreachable}\}$ specifies the notifications after which the monitoring relation is not automatically terminated. There can be at most one tuple for each $(monitoring, monitored)$ pair. Therefore, the following condition holds for any monitoring relation: $(alice, bob, Persist) \in Mo \Rightarrow \nexists Persist'. (alice, bob, Persist') \in Mo \setminus (alice, bob, Persist)$.

Supervision relations (Su)

Let Su be a set of tuples $\{(supervisor, SupSet, SupPolicy)\}$ that represents the different supervision relations in the system. In each tuple, *supervisor* is the name of the supervisor agent, *SupSet* is the supervised set, i. e., the set of agents supervised by *supervisor*, and *SupPolicy* is the supervision policy followed by *supervisor*.

Let *SupSet* be a list of tuples $(supervised, Ping, Unblock, Restart, Revival)$ where *supervised* is the name of an agent in the supervised set, and elements *Ping*, *Unblock*, *Restart* and *Revival* are lists of timestamps. These timestamps correspond to the moments at which the supervised agent has failed to answer to a ping request or its unblock policy (resp. restart and revival policies) has been triggered.

Let *SupPolicy* be a tuple $(PingPolicy, UnblockPolicy, RestartPolicy, RevivalPolicy, revivalStrategy)$ that represents the supervision policy. Its elements correspond to the different policies described in Section 4.2.2. *PingPolicy* is a tuple $(frequency, maxPings)$. *UnblockPolicy* is a tuple $(maxUnblocks, uMaxTime)$. *RestartPolicy* is a tuple $(maxRestarts, restartMaxTime)$. *RevivalPolicy* is a tuple $(maxRevive, reviveMaxTime)$. Finally, *revivalStrategy* is one of $\{\text{one_for_one, rest_for_one, rest_for_all}\}$.

Any agent in the system belongs to at most one supervised set and, consequently, has at most one supervisor agent. Consider the tuple $(bob, Ping, Unblock, Restart, Revival) \in SupSet$ such that $(alice, SupSet, SupPolicy) \in Su$ (i.e. *alice* supervises *bob*). Then, for any other existing tuple $(bob, Ping', Unblock', Restart', Revival') \in SupSet' \Rightarrow SupSet = SupSet'$, which also implies $(alice, SupSet', SupPolicy) \in Su$.

Finally, for any given set Su of supervision relations, an underlying digraph $G_{Su}=(V, E)$ can be constructed. The set V of vertex corresponds to the set of all different agent names involved in a supervision relation, while the set E of edges is determined by the supervision relations in the following way:

$$V = \bigcup_{(aSup, SupSet, \dots) \in Su} (\{aSup\} \cup \{supervised \mid (supervised, \dots) \in SupSet\})$$

$$E = \{(aSup, supervised) \mid (aSup, SupSet, \dots) \in Su, (supervised, \dots) \in SupSet\}$$

In any eJason system, this digraph must be acyclic, which implies that the different supervision relations always present a tree structure. This absence of cycles is always tested as a precondition before the establishment of a new supervision relation using the following auxiliary function:

$$\text{has_cycle}(Su) = \begin{cases} false & \text{if } G_{Su} \text{ is acyclic} \\ true & \text{otherwise} \end{cases}$$

Note that checking whether G_{Su} is acyclic can be easily achieved using different algorithms from the classical literature of graph theory (e.g. checking whether a depth-first-search would visit any vertex more than once).

System events (SE)

Let SE be a set of tuples $\{(type, agent)\}$ that represents the different system events observable by the fault-tolerance mechanisms of eJason. The element $type \in \{\text{created, dead, restarted, revived, unreachable, stopped, ping_timeout}\}$ corresponds to the type of the event, while $agent$ corresponds to the name of the agent related to that event. The first five types of events are observable only by the monitoring mechanism, while the remaining ones are observable only by the supervision mechanism. As a matter of example, consider a system event $(\text{created}, \text{alice})$, which is generated after a new agent, named *alice*, is added to the system. This event will trigger the corresponding notification to the agents monitoring *alice* (see rule *created* in Section 5.2.3). An event $(\text{stopped}, \text{alice})$, occurs when the agent *alice* has stopped functioning (e.g. because its corresponding computational process has died). Depending on the applicable revival policy, this event will generate either $(\text{revived}, \text{alice})$ if *alice* is revived or $(\text{dead}, \text{alice})$ otherwise. Similarly, an event $(\text{ping_timeout}, \text{alice})$ represents the fact that *alice* has failed to respond to a ping signal on time.

Timestamps (t)

Let t be the time of the system. This element is included as a convenience to allow the association of a timestamp to the supervision actions whose execution time is relevant (i.e. a timed out ping, the triggering of an unblock policy, etc.). Note that t is omitted in all the rules that do not require its use as a timestamp. It is assumed to increase by some arbitrary amount ϵ after each transition (i.e. $t' = t + \epsilon$).

Auxiliary functions

For the sake of readability of the transition rules provided in the remainder of this section, the following auxiliary functions are considered:

- In order to obtain the set of all agent names in a multi-agent system $M = (Di, \dots)$, the following function is utilised:

$$\text{names}(Di) = \{Ag_{name} | Ag \in Agents, (cont, Agents) \in Di\}$$

- To represent the elimination of an agent from the system by the execution environment (e.g. when its revival threshold is exceeded), the following function is used:

$$\begin{aligned} \text{kill_agent}(name, Di) &= (Di \setminus Di_{name}) \cup Di'_{name} \text{ s.t.} \\ Di_{name} &= (cont, Agents) \in Di \text{ where } (ag, ag_0, name, \dots) \in Agents \\ Di'_{name} &= (cont, Agents \setminus (ag, ag_0, name, \dots)) \end{aligned}$$

- The function that represents the effects of returning an agent to its initial state (as it is necessary during restarts and revivals) is the following:

$$\begin{aligned} \text{initial_state}(name, Di) &= (Di \setminus Di_{name}) \cup Di'_{name} \text{ s.t.} \\ Di_{name} &= (cont, Agents) \in Di \text{ where} \\ (ag, ag_0, name, C, M, T, s) &\in Agents \\ Agents' &= (Agents \setminus (ag, ag_0, name, \dots)) \cup \\ (ag_0, ag_0, name, C, M, T, \text{ProcMsg}) & \\ Di'_{name} &= (cont, Agents') \end{aligned}$$

- The task of computing the set of agents to be restarted when a restart strategy of type **rest_for_one** is triggered (i.e. all agents that appear after an agent a_r in the list *SupSet* that represents some supervised set) is represented as the execution of an action **rest**(a_r , *SupSet*).
- The interruption of the computations that take place during the resolution of the conditions in a plan context or a test goal, for some agent a_{ub} , is represented by the execution the function **unblock**(a_{ub} , *Di*).

5.2.2 Relevant Internal Actions

This section provides the semantics of the most important language constructs (internal actions) available to the agent programmers with regards to agent distribution and fault-tolerance. These operations include the internal actions to create a new agent, to kill some existing agent and to establish new supervision or monitoring relations. As these transition rules correspond to the execution of internal actions, it can be considered that their execution occurs within the execution of the function `execute(act)` in the transition rule *action* (in page 99).

Agent creation with a given name

If an agent with name *name* attempts the creation of a new agent with the specific name *newName*, a name clash may occur if some other agent with that name already exists in the system (i.e. $newName \in \mathbf{names}(Di)$). If there is no name clash, the initial state of the new agent is determined by its source, represented as *agCode*, and the name given to the agent (rule *create_agent1*), as represented by the execution of the function `create` introduced in Section 5.1.4. Otherwise, the creation fails and the corresponding failure event is generated within the agent that attempted the creation (rule *create_agent2*).

$$\begin{array}{c}
 (container, Agents) \in Di \\
 (ag, ag_0, name, C, M, T, s) \in Agents \\
 .create_agent(newName, agCode) \in C_A \\
 newName \notin \mathbf{names}(Di) \\
 \hline
 (create_agent1) \frac{}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo, Su, SE')}
 \end{array}$$

$$\begin{array}{l}
 \text{where:} \quad Agents' = Agents \cup \text{NewAgent} \\
 \quad \quad \text{NewAgent} = \text{create}(newName, agCode) \\
 \quad \quad C'_A = C_A \setminus (.create_agent(newName, agCode)) \\
 \\
 SE' = SE \cup (\text{created}, newName)
 \end{array}$$

$$\begin{array}{c}
 (container, Agents) \in Di \\
 (ag, ag_0, name, C, M, T, s) \in Agents \\
 .create_agent(newName, agCode) \in C_A \\
 newName \in \mathbf{names}(Di) \\
 \hline
 (create_agent2) \frac{}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo, Su, SE)}
 \end{array}$$

$$\begin{aligned}
\text{where: } & Agent' = (ag, ag_0, name, C', M, T, ProcMsg) \\
& C'_E = C_E \cup \mathbf{fail_event}(te, i) \quad s.t. \quad T_i = i[te : ct \leftarrow h] \\
& C'_A = C_A \setminus (.create_agent(newName, agCode))
\end{aligned}$$

Anonymous agent creation

The anonymous creation of an agent cannot generate a name clash. The new agent created is given a name, *newName*, that does not generate a name clash, i.e. $newName \notin \mathbf{Names}(Di)$ (rule *create_agent3*).

$$\text{(create_agent3)} \quad \frac{
\begin{array}{l}
(container, Agents) \in Di \\
(ag, ag_0, name, C, M, T, s) \in Agents \\
.create_agent(UnboundVar, agCode) \in C_A
\end{array}
}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo, Su, SE')}$$

$$\begin{aligned}
\text{where: } & Agents' = Agents \cup \mathbf{NewAgent} \\
& \mathbf{NewAgent} = \mathbf{create}(newName, agCode) \quad s.t. \\
& \quad \quad \quad newName \notin \mathbf{names}(Di) \\
& \quad \quad \quad UnboundVar \text{ is bound to } newName \\
& C'_A = C_A \setminus (.create_agent(UnboundVar, agCode)) \\
& SE' = SE \cup (\mathbf{created}, newName)
\end{aligned}$$

Agent kill

When an agent named *name* kills another agent *agName*, this latter agent is stopped, generating the corresponding system event (rule *kill_agent*). In the same way, if the eJason runtime system detects that some agent *agName* has stopped functioning the same system event is generated (rule *agent_stop*). Note that, in both cases, the execution of the transition rules amounts to adding a new system event. This event will eventually trigger a new transition (see rules *stop1* and *stop2* in Section 5.2.4). Consequently, a stopped/killed agent does not leave the system until it is declared *dead* by the runtime system.

$$\text{(kill_agent)} \quad \frac{
\begin{array}{l}
(container, Agents) \in Di \\
(ag, ag_0, name, C, M, T, s) \in Agents \\
.kill_agent(agName) \in C_A
\end{array}
}{(Di, Mo, Su, SE) \Rightarrow (Di, Mo, Su, SE')}$$

$$\begin{aligned} \text{where:} \quad SE' &= SE \cup (\text{stopped}, agName) \\ C'_A &= C_A \setminus (.kill_agent(agName)) \end{aligned}$$

$$(\text{agent_stop}) \frac{}{(Di, Mo, Su, SE) \Rightarrow (Di, Mo, Su, SE')}$$

$$\text{where:} \quad SE' = SE \cup (\text{stopped}, agName)$$

Monitoring relation

The execution of the internal action that establishes a monitoring relation between the agents mon_{ing} and mon_{ed} (monitoring and monitored agents resp.) has three different outcomes. If a monitoring relation involving the same agents with the same roles already exists (i.e. $(mon_{ing}, mon_{ed}, \dots) \in Mo$), this relation is updated with $Persist'$, the new persistence information (rule $monitor_agent1$). Otherwise, a new monitoring relation is registered within the system. In this case, if the monitored agent belongs to the system (i.e. $mon_{ed} \in \mathbf{names}(Di)$), no other changes apply (rule $monitor_agent2$). If the monitored agent does not belong to the system (i.e. $mon_{ed} \notin \mathbf{names}(Di)$), a notification of type unknown is sent to the monitoring agent (rule $monitor_agent3$).

$$(\text{monitor_agent1}) \frac{\begin{aligned} &(container, Agents) \in Di \\ &(ag, ag_0, mon_{ing}, C, M, T, s) \in Agents \\ &(mon_{ing}, mon_{ed}, Persist) \in Mo \\ &.monitor(mon_{ed}, Persist') \in C_A \end{aligned}}{(Di, Mo, Su, SE) \Rightarrow (Di, Mo', Su, SE)}$$

$$\begin{aligned} \text{where:} \quad Mo' &= (Mo \setminus (mon_{ing}, mon_{ed}, Persist)) \cup \\ &(mon_{ing}, mon_{ed}, Persist') \\ C'_A &= C_A \setminus (.monitor(mon_{ed}, Persist)) \end{aligned}$$

$$(\text{monitor_agent2}) \frac{\begin{aligned} &(container, Agents) \in Di \\ &(ag, ag_0, mon_{ing}, C, M, T, s) \in Agents \\ &(mon_{ing}, mon_{ed}, \dots) \notin Mo \\ &mon_{ed} \in \mathbf{names}(Di) \\ &.monitor(mon_{ed}, Persist) \in C_A \end{aligned}}{(Di, Mo, Su, SE) \Rightarrow (Di, Mo', Su, SE)}$$

$$\begin{aligned} \text{where: } Mo' &= Mo \cup (mon_{ing}, mon_{ed}, Persist) \\ C'_A &= C_A \setminus (.monitor(mon_{ed}, Persist)) \end{aligned}$$

$$\begin{array}{c} (container, Agents) \in Di \\ (ag, ag_0, mon_{ing}, C, M, T, s) \in Agents \\ (mon_{ing}, mon_{ed}, \dots) \notin Mo \\ mon_{ed} \notin \mathbf{names}(Di) \\ .monitor(mon_{ed}, Persist) \in C_A \\ \hline (monitor_agent3) \quad (Di, Mo, Su, SE) \Rightarrow (Di', Mo', Su, SE) \end{array}$$

$$\begin{aligned} \text{where: } Mo' &= Mo \cup (mon_{ing}, mon_{ed}, Persist) \\ ag'_{bs} &= ag_{bs} \cup agent_down(mon_{ed})[reason(unknown_agent)] \\ C'_A &= C_A \setminus (.monitor(mon_{ed}, Persist)) \end{aligned}$$

Supervision relation

To successfully establish a new supervision relation between a supervisor agent $aSup$ and a set of agents $SupAgents$ (i.e. the supervised set) none of the agents in the supervised must be already supervised, i.e. none of them belongs to another supervised set, and the addition of the new relation should not generate loops in the supervision structure (rule *supervise1*). If the different resulting supervision sets were not disjoint or a loop in the supervision structure would be introduced, the corresponding failure event is added to the set of events for agent $aSup$, i.e. the agent that tries to establish the supervision relation, (rules *supervise2* and *supervise3*, respectively).

$$\begin{array}{c} (container, Agents) \in Di \\ (ag, ag_0, aSup, C, M, T, s) \in Agents \\ .supervise(SupAgents, SupPolicy) \in C_A \\ \neg has_loop(Su \cup (aSup, SupAgents, SupPolicy)) \\ \forall (superv, SupSet, \dots) \in Su, (supervised, \dots) \in SupSet. \\ supervised \notin SupAgents \\ \hline (supervise1) \quad (Di, Mo, Su, SE) \Rightarrow (Di', Mo, Su', SE) \end{array}$$

$$\begin{aligned} \text{where: } Su' &= Su \cup (aSup, SupAgents, SupPolicy) \\ C'_A &= C_A \setminus (.supervise(SupAgents, SupPolicy)) \end{aligned}$$

$$\begin{array}{c}
\text{(supervise2)} \quad \frac{
\begin{array}{l}
(container, Agents) \in Di \\
(ag, ag_0, aSup, C, M, T, s) \in Agents \\
.supervise(SupAgents, SupPolicy) \in C_A \\
\exists(monitoring, SupSet, \dots) \in Su, (supervised, \dots) \in SupSet. \\
supervised \in SupAgents
\end{array}
}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo, Su, SE)}
\end{array}$$

$$\begin{array}{l}
\text{where:} \quad C'_E = C_E \cup \text{fail_event}(te, i) \quad s.t. \quad T_i = i[te : ct \leftarrow h] \\
\quad \quad C'_A = C_A \setminus (.supervise(SupAgents, SupPolicy))
\end{array}$$

$$\begin{array}{c}
\text{(supervise3)} \quad \frac{
\begin{array}{l}
(container, Agents) \in Di \\
(ag, ag_0, aSup, C, M, T, s) \in Agents \\
.supervise(SupAgents, SupPolicy) \in C_A \\
has_loop(Su \cup (aSup, SupAgents, SupPolicy))
\end{array}
}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo, Su, SE)}
\end{array}$$

$$\begin{array}{l}
\text{where:} \quad C'_E = C_E \cup \text{fail_event}(te, i) \quad s.t. \quad T_i = i[te : ct \leftarrow h] \\
\quad \quad C'_A = C_A \setminus (.supervise(SupAgents, SupPolicy))
\end{array}$$

System split

Any eJason multi-agent system, (Di, Mo, Su, SE) that undergoes a split process diminishes its number of agent containers. There are two types of system splits. On the one hand a non-empty set of agent containers $Co_1 \subset Di$ may leave (disconnect from) the system to compose a different, independent multi-agent system. Hence obtaining two different multi-agent systems (Co_1, Mo_1, Su_1, SE_1) and (Co_2, Mo_2, Su_2, SE_2) (rule *split1*). On the other hand, a non-empty set of agent containers Co may stop working (e.g. because the hosts in which they run have been stopped), hence not creating a new multi-agent system but modifying the original one to be $(Di \setminus Co, Mo', Su', SE')$ (rule *split2*).

$$\text{(split1)} \quad \frac{
\begin{array}{l}
\emptyset \neq Co_1 \subset Di \quad \emptyset \neq Co_2 = Di \setminus Co_1
\end{array}
}{(Di, Mo, Su, SE) \Rightarrow \left\{ \begin{array}{l} (Co_1, Mo_1, Su_1, SE_1) \\ (Co_2, Mo_2, Su_2, SE_2) \end{array} \right\}}$$

where:

$$\begin{aligned}
Mo_1 &= \{(monitoring, monitored, Persist)\} \subseteq Mo \text{ s.t.} \\
&\quad monitoring \in \mathbf{names}(Co_1) \\
Mo_2 &= \{(monitoring, monitored, Persist)\} \subseteq Mo \text{ s.t.} \\
&\quad monitoring \in \mathbf{names}(Co_2) \\
\\
Su_1 &= \{(supervisor, SupSet_1, SupPolicy)\} \text{ s.t.} \\
&\quad (supervisor, SupSet, SupPolicy) \in Su \wedge \\
&\quad supervisor \in \mathbf{names}(Co_1) \wedge \\
&\quad \emptyset \neq SupSet_1 = \{(supervised, \dots)\} \subseteq SupSet \text{ s.t.} \\
&\quad supervised \in \mathbf{names}(Co_1) \\
\\
Su_2 &= \{(supervisor, SupSet_2, SupPolicy)\} \text{ s.t.} \\
&\quad (supervisor, SupSet, SupPolicy) \in Su \wedge \\
&\quad supervisor \in \mathbf{names}(Co_2) \wedge \\
&\quad \emptyset \neq SupSet_2 = \{(supervised, \dots)\} \subseteq SupSet \text{ s.t.} \\
&\quad supervised \in \mathbf{names}(Co_2) \\
\\
SE_1 &= SE \cup \left\{ \bigcup_{agent \in \mathbf{names}(Co_2)} (unreachable, agent) \right\} \\
SE_2 &= SE \cup \left\{ \bigcup_{agent \in \mathbf{names}(Co_1)} (unreachable, agent) \right\}
\end{aligned}$$

Note how, as a result of the split operation, two different MAS appear. The sets Mo_1 and Mo_2 retain only the monitoring relations, from Mo , whose monitoring agent belongs to their system. The resulting sets Su_1 and Su_2 contain the supervision relations from Su such that the supervisor agent belongs to the system. Besides, these supervision relations are modified to remove from the supervised set the agents that get disconnected from the supervisor (i.e. which belong to a different MAS). If the resulting supervised set is empty, the supervision relation is removed. Finally, new system events are added in order to notify the disconnection of agents in each system.

$$(\text{split2}) \frac{\emptyset \neq Co \subset Di \quad \emptyset \neq Co' = Di \setminus Co}{(Di, Mo, Su, SE) \Rightarrow (Co', Mo', Su', SE')}$$

$$\text{where:} \quad Mo' = \{(monitoring, monitored, Persist)\} \subseteq Mo \text{ s.t.} \\
\quad \quad \quad monitoring \in \mathbf{names}(Co')$$

$$Su' = \{(supervisor, SupSet', SupPolicy)\} \text{ s.t.}$$

$$\begin{aligned}
& (supervisor, SupSet, SupPolicy) \in Su \wedge \\
& supervisor \in \mathbf{names}(Co') \wedge \\
& \emptyset \neq SupSet' = \{(supervised, \dots)\} \subseteq SupSet \text{ s.t.} \\
& supervised \in \mathbf{names}(Co')
\end{aligned}$$

$$SE' = SE \cup \left\{ \bigcup_{agent \in \mathbf{names}(Co_1)} (dead, agent) \right\}$$

In this case, only one MAS remains after the split. The main difference with respect to the rule *split1* is that the system events included notify the death of agents (as they have not been just disconnected, but killed).

System merge

Two different eJason multi-agent systems $M_1 = (Di_1, Mo_1, Su_1, SE_1)$ and $M_2 = (Di_2, Mo_2, Su_2, SE_2)$ can be merged into a new one $M' = (Di', Mo', Su', SE')$, provided that it does not cause a name clash. Otherwise, this operation cannot take place. Moreover, the execution of this operation is delayed until all the system events in SE_1 (resp. SE_2) involving agents in Di_2 (resp. Di_1) have been processed. This restriction is added for consistency, as the concatenation of SE_1 and SE_2 into SE' can potentially interfere with the delivery order of the different notifications related to the fault-tolerance mechanisms. The successful execution of a merge operation is formalised by the transition rule *merge1*. The failed attempt to execute a merge operation (due to either the nonexistence of M_2 or the occurrence of a name clash) is formalised by the transition rule *merge2*.

$$\begin{array}{c}
(container, Agents) \in Di \\
(ag, ag_0, agName, C, M, T, s) \in Agents \\
.merge(container_2) \in C_A \text{ s.t. } (container_2, \dots) \in Di_2 \\
\mathbf{names}(Di_1) \cap \mathbf{names}(Di_2) = \emptyset \\
\forall (type_1, name_1) \in SE_1. name_1 \notin \mathbf{names}(Di_2) \\
\forall (type_2, name_2) \in SE_2. name_2 \notin \mathbf{names}(Di_1) \\
\hline
(merge1) \frac{\left\{ \begin{array}{l} (Di_1, Mo_1, Su_1, SE_1) \\ (Di_2, Mo_2, Su_2, SE_2) \end{array} \right\}}{(Di', Mo', Su', SE')}
\end{array}$$

$$\begin{aligned}
\text{where: } Di' &= Di_1 \cup Di_2 \\
Mo' &= Mo_1 \cup Mo_2 \\
Su' &= Su_1 \cup Su_2 \\
SE' &= SE_1 \cup SE_2 \cup \left\{ \bigcup_{(cont, Agents) \in Di'} \bigcup_{Ag \in Agents} (created, Ag_{name}) \right\}
\end{aligned}$$

$$\begin{array}{c}
\frac{
\begin{array}{l}
(container, Agents) \in Di \\
(ag, ag_0, agName, C, M, T, s) \in Agents \\
.merge(container_2) \in C_A \text{ s.t.} \\
(container_2, \dots) \notin Di_2 \vee (\mathbf{names}(Di_1) \cap \mathbf{names}(Di_2) \neq \emptyset)
\end{array}
}{
\text{(merge2)} \quad \left\{ \begin{array}{l} (Di_1, Mo_1, Su_1, SE_1) \\ (Di_2, Mo_2, Su_2, SE_2) \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} (Di'_1, Mo_1, Su_1, SE_1) \\ (Di_2, Mo_2, Su_2, SE_2) \end{array} \right\}
}
\end{array}$$

$$\begin{array}{l}
\text{where:} \quad Agent' = (ag, ag_0, name, C', M, T, ProcMsg) \\
C'_E = C_E \cup \mathbf{fail_event}(te, i) \text{ s.t. } T_i = i[te : ct \leftarrow h] \\
C'_A = C_A \setminus (.merge(container_2))
\end{array}$$

5.2.3 Monitoring Notifications

The system events that correspond to the changes (creation, disconnection, death, restart, revival) of the execution state of an agent and that are observable by the monitoring mechanism are notified to the proper monitoring agents using their homonym transition rules.

$$\begin{array}{c}
\frac{
\text{(created)} \quad (created, agent_{cr}) \in SE
}{
(Di, Mo, Su, SE) \Rightarrow (Di', Mo', Su, SE')
} \\
\text{where:} \quad \begin{array}{l}
Di' = \{(container, Agents')\} \text{ s.t.} \\
(container, Agents) \in Di \wedge \\
\forall (monitoring, agent_{cr}, Persist) \in Mo, \\
(ag, monitoring, \dots) \in Agents. \\
(ag \cup +agent_up(agent_{cr}), monitoring, \dots) \in Agents' \wedge \\
\forall (ag, name, \dots) \in Agents, (name, agent_{cr}, Persist) \notin Mo. \\
(ag, monitoring, \dots) \in Agents' \\
Mo' = \{(monitoring, monitored, Persist)\} \in Mo \text{ s.t.} \\
monitored \neq agent_{cr} \vee \\
(monitored = agent_{cr} \wedge \mathbf{created} \in Persist) \\
SE' = SE \setminus (created, agent_{cr})
\end{array}
\end{array}$$

The execution of this transition rule adds a new belief of the form $+agent_up(agent)$ to the belief base of all the agents monitoring the agent $agent_{cr}$ (i.e. to all agents $monitoring$ s.t. $(monitoring, agent_{cr}, Persist) \in Mo$). Besides, the monitoring relations that do not persist after the notification of an agent creation are terminated, hence not included in the set Mo' .

$$\text{(unreachable)} \frac{(\text{unreachable}, \text{agent}_{\text{unr}}) \in SE}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo', Su, SE')}$$

where:

$$\begin{aligned}
 & Di' = \{(container, Agents')\} \text{ s.t.} \\
 & (container, Agents) \in Di \wedge \\
 & \forall (monitoring, \text{agent}_{\text{unr}}, Persist) \in Mo, \\
 & (ag, monitoring, \dots) \in Agents. \\
 & (ag', monitoring, \dots) \in Agents' \text{ s.t.} \\
 & ag' = (ag \cup +\text{agent_down}(\text{agent}_{\text{unr}}) \\
 & \quad [\text{reason}(\text{unreachable_agent})]) \wedge \\
 & \forall (ag, name, \dots) \in Agents, \\
 & (name, \text{agent}_{\text{unr}}, Persist) \notin Mo. \\
 & (ag, monitoring, \dots) \in Agents' \\
 & Mo' = \{(monitoring, monitored, Persist)\} \in Mo \text{ s.t.} \\
 & \quad monitored \neq \text{agent}_{\text{unr}} \vee \\
 & \quad (monitored = \text{agent}_{\text{unr}} \wedge \text{unreachable} \in Persist) \\
 & SE' = SE \setminus (\text{unreachable}, \text{agent}_{\text{unr}})
 \end{aligned}$$

The rules involved in the notification of agent deaths, restarts and revivals are constructed in the same way as the rule for agent disconnection above.

$$\text{(dead)} \frac{(\text{dead}, \text{agent}_{\text{dead}}) \in SE}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo', Su, SE')}$$

where:

$$\begin{aligned}
 & Di' = \{(container, Agents')\} \text{ s.t.} \\
 & (container, Agents) \in Di \wedge \\
 & \forall (monitoring, \text{agent}_{\text{dead}}, Persist) \in Mo, \\
 & (ag, monitoring, \dots) \in Agents. \\
 & (ag', monitoring, \dots) \in Agents' \text{ s.t.} \\
 & ag' = (ag \cup +\text{agent_down}(\text{agent}_{\text{dead}}) \\
 & \quad [\text{reason}(\text{dead_agent})]) \wedge \\
 & \forall (ag, name, \dots) \in Agents, \\
 & (name, \text{agent}_{\text{dead}}, Persist) \notin Mo. \\
 & (ag, monitoring, \dots) \in Agents' \\
 & Mo' = \{(monitoring, monitored, Persist)\} \in Mo \text{ s.t.} \\
 & \quad monitored \neq \text{agent}_{\text{dead}} \vee \\
 & \quad (monitored = \text{agent}_{\text{dead}} \wedge \text{dead} \in Persist) \\
 & SE' = SE \setminus (\text{dead}, \text{agent}_{\text{dead}})
 \end{aligned}$$

$$\text{(restarted)} \frac{(\text{restarted}, \text{agent}_{rst}) \in SE}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo', Su, SE')}$$

where:

$$\begin{aligned} Di' &= \{(container, Agents')\} \text{ s.t.} \\ (container, Agents) &\in Di \wedge \\ \forall (monitoring, \text{agent}_{rst}, Persist) &\in Mo, \\ (ag, monitoring, \dots) &\in Agents. \\ (ag', monitoring, \dots) &\in Agents' \text{ s.t.} \\ ag' &= (ag \cup +\text{agent_down}(\text{agent}_{rst}) \\ &\quad [\text{reason}(\text{restarted_agent})]) \wedge \\ \forall (ag, name, \dots) &\in Agents, (name, \text{agent}_{rst}, Persist) \notin Mo. \\ (ag, monitoring, \dots) &\in Agents' \\ Mo' &= \{(monitoring, monitored, Persist)\} \in Mo \text{ s.t.} \\ monitored &\neq \text{agent}_{rst} \vee \\ (monitored = \text{agent}_{rst} \wedge \text{restarted} &\in Persist) \\ SE' &= SE \setminus (\text{restarted}, \text{agent}_{rst}) \end{aligned}$$

$$\text{(revived)} \frac{(\text{revived}, \text{agent}_{rvd}) \in SE}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo', Su, SE')}$$

where:

$$\begin{aligned} Di' &= \{(container, Agents')\} \text{ s.t.} \\ (container, Agents) &\in Di \wedge \\ \forall (monitoring, \text{agent}_{rvd}, Persist) &\in Mo, \\ (ag, monitoring, \dots) &\in Agents. \\ (ag', monitoring, \dots) &\in Agents' \text{ s.t.} \\ ag' &= (ag \cup +\text{agent_down}(\text{agent}_{rvd}) \\ &\quad [\text{reason}(\text{revived_agent})]) \wedge \\ \forall (ag, name, \dots) &\in Agents, \\ (name, \text{agent}_{rvd}, Persist) &\notin Mo. \\ (ag, monitoring, \dots) &\in Agents' \\ Mo' &= \{(monitoring, monitored, Persist)\} \in Mo \text{ s.t.} \\ monitored &\neq \text{agent}_{rvd} \vee \\ (monitored = \text{agent}_{rvd} \wedge \text{revived} &\in Persist) \\ SE' &= SE \setminus (\text{revived}, \text{agent}_{rvd}) \end{aligned}$$

5.2.4 Supervision

As described in Section 4.2.2, the supervision mechanism of eJason implements supervision policies that not only react when an agent stops functioning, but also periodically test the possible divergence of the agents supervised.

Agent stop

The event that represents the stop of the functioning of an agent is formalised as the tuple $(\text{stopped}, \text{agent})$. When this tuple belongs to the set SE of system events, one of the two rules below, stop1 or stop2 , is triggered. Note that, whether an agent dies or not is determined by the supervision policy (or the lack of it) applied over the agent.

The transition rule stop1 is triggered when the stopped agent does not belong to any supervised set. In this case, the agent is removed from its container (Di_{agent}). The supervision relations where the stopped agent acted as supervisor agent are also terminated. Finally, a new system event $(\text{dead}, \text{agent})$ is generated.

$$\text{(stop1)} \frac{\begin{array}{l} (\text{stopped}, \text{agent}_{st}) \in SE \\ \forall (\text{supervisor}, \text{SupSet}, \text{SupPolicy}) \in Su. \\ (\text{agent}_{st}, \dots) \notin \text{SupSet} \end{array}}{(Di, Mo, Su, SE) \Rightarrow (Di', Mo, Su', SE')}$$

where:

$$Di' = Di'_{\text{agent}} \cup Di_{\text{Rest}}$$

$$Di_{\text{agent}} = (\text{container}, \text{Agents}) \in Di \text{ s.t.}$$

$$(\text{ag}, \text{ag}_0, \text{agent}_{st}, \dots) \in \text{Agents}$$

$$Di'_{\text{agent}} = (\text{container}, \text{Agents} \setminus (\text{ag}, \text{ag}_0, \text{agent}_{st}, \dots))$$

$$Di_{\text{Rest}} = Di \setminus Di_{\text{agent}}$$

$$Su' = \begin{cases} Su & \text{if } (\text{agent}, \dots) \notin Su \\ Su \setminus (\text{agent}_{st}, \text{SupSet}, \text{SupPolicy}) & \text{otherwise} \end{cases}$$

$$SE' = (SE \setminus (\text{stopped}, \text{agent}_{st})) \cup (\text{dead}, \text{agent}_{st})$$

The transition rule stop2 is triggered when the stopped agent belongs to some supervised set. In that case, the revival policy is triggered in order to determine whether the agent is revived or not. The function **Supervision**_{stopped}, whose pseudocode is given in Algorithm 1, represents the procedure followed by the revival policy. This procedure checks whether the revival limit has been exceeded

or not. If it has been exceeded, the supervision relation is terminated and all the agents in the supervised set are killed ($SupSet'=\emptyset$). Otherwise, the agent stopped is revived (returned to its initial state), the timestamp t for that revival is recorded and the corresponding restart strategy is applied over the rest of the agents in the supervised set Di_{SupSet} . The function **restart_strategy**, whose pseudocode is given in Algorithm 2, represents the procedure followed in order to apply each restart strategy.

$$\text{(stop2)} \frac{\begin{array}{l} (\text{stopped}, agent_{st}) \in SE \\ (supervisor, SupSet, SupPolicy) \in Su \\ (agent_{st}, \dots) \in SupSet \end{array}}{(Di, Mo, Su, SE, t) \Rightarrow (Di', Mo, Su', SE', t')}$$

where:

$$\begin{aligned} Di' &= Di'_{SupSet} \cup Di_{Rest} \\ Di_{SupSet} &= \{(cont, Agents)\} \subseteq Di \text{ s.t.} \\ &\quad (ag, ag_0, name, \dots) \in Agents \wedge (name, \dots) \in SupSet \\ Di_{Rest} &= Di \setminus Di_{SupSet} \end{aligned}$$

$$\begin{aligned} Su' &= (Su \setminus SupRel) \cup SupRel' \\ SupRel &= (supervisor, SupSet, SupPolicy), \\ SupRel' &= \begin{cases} (supervisor, SupSet', \\ \quad SupPolicy) & \text{if } SupSet' \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

$$SE' = (SE \setminus (\text{stopped}, agent_{st})) \cup NewSE$$

$$\begin{aligned} (Di'_{SupSet}, SupSet', NewSE) &= \\ &= \text{Supervision}_{\text{stopped}}(agent_{st}, t, Di_{SupSet}, SupSet, SupPolicy) \end{aligned}$$

$$t' = t + \epsilon$$

Algorithm 1 Supervision process triggered when some agent a_{st} stops its execution

procedure
 $\text{SUPERVISION}_{\text{stopped}}(a_{st}, t, Di_{\text{SupSet}}, \text{SupSet}, \text{SupPolicy})$
 $\text{Info} \leftarrow (a_{st}, \text{Ping}, \text{Unblock}, \text{Restart}, \text{Revival}) \in \text{SupSet}$
 $\text{Info}' \leftarrow (a_{st}, \text{Ping}, \text{Unblock}, \text{Restart}, \text{Revival} \cup t)$
if $\text{revival_limit_exceeded}(\text{Info}', \text{SupPolicy})$ **then**
for each agent $(\text{other}, \dots) \in \text{SupSet}$ **do**
 $Di_{\text{SupSet}} \leftarrow \text{kill_agent}(\text{other}, Di_{\text{SupSet}})$
 $\text{NewSE} \leftarrow \text{NewSE} \cup (\text{dead}, \text{other})$
end for each
 $\text{SupSet}' \leftarrow \emptyset$
else
 $Di_{\text{SupSet}} \leftarrow \text{initial_state}(a_{st}, Di_{\text{SupSet}})$
 $\text{SupSet} \leftarrow (\text{SupSet} \setminus \text{Info}) \cup \text{Info}'$
 $(Di_{\text{SupSet}}, \text{SupSet}', \text{NewSE}) \leftarrow \text{restart_strategy}(a_{st}, t, Di_{\text{SupSet}}, \text{SupSet}, \text{SupPolicy})$
 $\text{NewSE} \leftarrow \text{NewSE} \cup (\text{revived}, a_{st})$
end if
return $(Di_{\text{SupSet}}, \text{SupSet}', \text{NewSE})$
end procedure

Algorithm 2 Execution of a restart strategy triggered by the revival/restart of some agent a_r

```

procedure
  RESTART_STRATEGY( $a_r, t, Di_{SupSet}, SupSet, SupPolicy$ )
     $restart\_strategy \in SupPolicy$ 
    if  $restart\_strategy = one\_for\_one$  then
       $SupSet' \leftarrow SupSet$ 
       $NewSE \leftarrow \emptyset$ 
    else if  $restart\_strategy = rest\_for\_one$  then
       $Rest \leftarrow rest(a_r, SupSet)$ 
      for each agent ( $other, \dots$ )  $\in Rest$  do
         $Di_{SupSet} \leftarrow initial\_state(other, Di_{SupSet})$ 
         $Info \leftarrow (other, P, U, Restart, Rev) \in SupSet$ 
         $Info' \leftarrow (other, P, U, Restart \cup t, Rev)$ 
         $SupSet' \leftarrow (SupSet \setminus Info) \cup Info'$ 
         $NewSE \leftarrow NewSE \cup (restarted, other)$ 
      end for each
    else if  $restart\_strategy = one\_for\_all$  then
      for each agent ( $other \neq a_r, \dots$ )  $\in SupSet$  do
         $Di_{SupSet} \leftarrow initial\_state(other, Di_{SupSet})$ 
         $Info \leftarrow (other, P, U, Restart, Rev) \in SupSet$ 
         $Info' \leftarrow (other, P, U, Restart \cup t, Rev)$ 
         $SupSet' \leftarrow (SupSet \setminus Info) \cup Info'$ 
         $NewSE \leftarrow NewSE \cup (restarted, other)$ 
      end for each
    end if
    return ( $Di_{SupSet}, SupSet', NewSE$ )
end procedure

```

Ping timeout

Another observable event by the supervision mechanism is the timeout for the response corresponding to some ping signal, represented as the event (`ping_timeout`, agent). The transition rule *ping* formalises the behaviour of the supervision mechanism when such event occurs. This transition rule triggers the ping policy, i.e. executes the function `Supervisionping` whose pseudocode is given in Algorithm 3. This policy records the timestamp t that corresponds to the timeout event. Then, it checks whether the ping threshold has been exceeded or not. If it has been exceeded, the unblock policy is triggered, as described in Section 4.2.2. Similarly,

the functions `Supervisionunblock` and `Supervisionrestart`, whose pseudocode is given in Algorithms 4 and 5, correspond to the triggering of the unblock and restart policies, respectively.

$$\text{(ping)} \frac{\begin{array}{l} (\text{ping_timeout}, agent_{pn}) \in SE \\ (supervisor, SupSet, SupPolicy) \in Su \\ (agent_{pn}, \dots) \in SupSet \end{array}}{(Di, Mo, Su, SE, t) \Rightarrow (Di', Mo, Su', SE', t')}$$

where:

$$\begin{aligned} Di' &= Di'_{SupSet} \cup Di_{Rest} \\ Di_{SupSet} &= \{(cont, Agentss)\} \subseteq Di \text{ s.t.} \\ &\quad (agent_{pn}, \dots) \in SupSet \wedge (ag, ag_0, agent_{pn}, \dots) \in Agents \\ Di_{Rest} &= Di \setminus Di_{SupSet} \end{aligned}$$

$$\begin{aligned} Su' &= (Su \setminus SupRel) \cup SupRel' \\ SupRel &= (supervisor, SupSet, SupPolicy), \\ SupRel' &= \begin{cases} (supervisor, SupSet', \\ \quad SupPolicy) & \text{if } SupSet' \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

$$SE' = (SE \setminus (\text{stopped}, agent_{pn})) \cup NewSE$$

$$\begin{aligned} (Di'_{SupSet}, SupSet', NewSE) &= \\ &\quad \text{Supervision}_{\text{ping}}(agent_{pn}, t, Di_{SupSet}, SupSet, SupPolicy) \end{aligned}$$

$$t' = t + \epsilon$$

Algorithm 3 Supervision process triggered when some agent a_{pn} fails to respond to a ping signal

```

procedure
  SUPERVISIONping( $a_{pn}, t, Di_{SupSet}, SupSet, SupPolicy$ )
    Info  $\leftarrow (a_{pn}, Ping, Unblock, Restart, Revival) \in SupSet$ 
    Info'  $\leftarrow (a_{pn}, Ping \cup t, Unblock, Restart, Revival)$ 
    if ping_limit_exceeded(Info', SupPolicy) then
      Info'  $\leftarrow (a_{pn}, \emptyset, Unblock, Restart, Revival)$ 
      SupSet  $\leftarrow (SupSet \setminus Info) \cup Info'$ 
      ( $Di'_{SupSet}, SupSet', NewSE$ )  $\leftarrow$ 
        supervisionunblock( $a_{st}, t, Di_{SupSet}, SupSet, SupPolicy$ )
    else
       $Di'_{SupSet} \leftarrow Di_{SupSet}$ 
      SupSet'  $\leftarrow (SupSet \setminus Info) \cup Info'$ 
      NewSE  $\leftarrow \emptyset$ 
    end if
    return ( $Di'_{SupSet}, SupSet', NewSE$ )
end procedure

```

Algorithm 4 Supervision process triggered when some agent a_{ub} is unblocked

```

procedure
  SUPERVISIONunblock( $a_{ub}, t, Di_{SupSet}, SupSet, SupPolicy$ )
    Info  $\leftarrow (a_{ub}, Ping, Unblock, Restart, Revival) \in SupSet$ 
    Info'  $\leftarrow (a_{ub}, Ping, Unblock \cup t, Restart, Revival)$ 
    if unblock_limit_exceeded(Info', SupPolicy) then
      Info'  $\leftarrow (a_{pn}, \emptyset, \emptyset, Restart, Revival)$ 
      SupSet  $\leftarrow (SupSet \setminus Info) \cup Info'$ 
      ( $Di'_{SupSet}, SupSet', NewSE$ )  $\leftarrow$ 
        supervisionrestart( $a_{ub}, t, Di_{SupSet}, SupSet, SupPolicy$ )
    else
       $Di'_{SupSet} \leftarrow unblock(a_{ub}, Di_{SupSet})$ 
      SupSet'  $\leftarrow (SupSet \setminus Info) \cup Info'$ 
      NewSE  $\leftarrow \emptyset$ 
    end if
    return ( $Di'_{SupSet}, SupSet', NewSE$ )
end procedure

```

Algorithm 5 Supervision process triggered when some agent a_{rst} is restarted

procedureSUPERVISION_{restart}($a_{rst}, t, Di_{SupSet}, SupSet, SupPolicy$)Info $\leftarrow (a_{rst}, Ping, Unblock, Restart, Revival) \in SupSet$ Info' $\leftarrow (a_{rst}, Ping, Unblock \cup t, Restart, Revival)$ **if** restart_limit_exceeded(Info', SupPolicy) **then** **for each** agent (*other*, ...) $\in SupSet$ **do** $Di'_{SupSet} \leftarrow \text{kill_agent}(\textit{other}, Di_{SupSet})$ $NewSE \leftarrow NewSE \cup (\textit{dead}, \textit{other})$ **end for each** $SupSet' \leftarrow \emptyset$ **else** $Di_{SupSet} \leftarrow \text{initial_state}(a_{st}, Di_{SupSet})$ $SupSet' \leftarrow SupSet \setminus Info \cup Info'$ ($Di'_{SupSet}, SupSet', NewSE$) $\leftarrow$ restart_strategy($a_{rst}, t, Di_{SupSet}, SupSet', SupPolicy$) $NewSE \leftarrow NewSE \cup (\textit{restarted}, a_{rst})$ **end if** **return** ($Di'_{SupSet}, SupSet', NewSE$)**end procedure**

5.3 System Properties

The formal semantics provided before support the enunciation of a series of properties of the eJason systems that provide useful guarantees to the agent programmers. important ones.

Unicity of agent names

Theorem 1 *The name of an agent is a globally-unique identifier for that agent within the multi-agent system.*

Proof Let M be the multi-agent system (Di, Mo, Su, SE, t) . Then take any two different agents in the system, $agent_1$ and $agent_2$ with agent configurations $Ag_1 = (ag_1, ag_{01}, name_1, C_1, M_1, T_1, s_1)$ and $Ag_2 = (ag_2, ag_{02}, name_2, C_2, M_2, T_2, s_2)$, respectively. At some point t_0 in the past, $agent_2$ was added to M while $agent_1$ already existed within M at the immediately previous instant $t_{-1} < t_0$ (or, alternatively, $agent_2$ existed before $agent_1$; the resulting proof is analogous). We will show that the freshly created $agent_2$ always receives a fresh name (i.e., a name different from the names of any agent already existing in the system, as exemplified by the name of the arbitrary $agent_1$). The only possible situations leading to such an addition of the agent are that *i*) $agent_2$ is created using an internal action, i.e., executing one of the transition rules *create_agent1* or *create_agent3*, or *ii*) $agent_2$ comes from the system merge between M and a different multi-agent system $M' = (Di', Mo', Su', SE', t')$ that hosts $agent_2$, i.e., triggering the transition rule *merge*. Consequently:

- i*) At both times t_{-1} and t_0 the property $name_1 \in \mathbf{names}(Di)$ holds. For the transition to succeed the property $name_2 \notin \mathbf{names}(Di)$ holds, as implied by both the precondition of the transition *create_agent1* and the postcondition of the transition *create_agent3*. Thus, $name_2 \neq name_1$.
- ii*) At time t_{-1} the property $name_1 \in \mathbf{names}(Di)$ as well as the property $name_2 \in \mathbf{names}(Di')$ hold. The precondition of transition *merge* states that $\mathbf{names}(Di) \cap \mathbf{names}(Di') = \emptyset$ which implies $name_2 \neq name_1$.

Theorem 1 guarantees that the different transition rules whose execution adds new agents to the system cannot provoke a name clash, i.e., there cannot be two agents with the same name. This observation permits the use of an agent name within the different statements of the language in order to unambiguously address it, e.g., to specify the receiver of a message or to identify the monitored agent during the creation of a monitoring relation.

Non-conflicting Supervision Relations

Theorem 2 *At most one supervision policy can be maintained over an agent.*

Proof Let M be a multi-agent system (Di, Mo, Su, SE, t) that contains an agent sup_{ed} such that two different supervision policies are maintained over it. Therefore, Su contains at least two different supervision relations, $SupRel \neq SupRel'$, such that $SupRel = (superv, SupSet, SupPolicy) \neq SupRel' = (superv', SupSet', SupPolicy')$ where $(sup_{ed}, \dots) \in SupSet$ and $(sup_{ed}, \dots) \in SupSet'$. Assume that $SupRel$ was established before $SupRel'$ (since trivially there are no transition rules that establish multiple supervision rules). Then, at some moment t_1 , $SupRel'$ was established by the execution of the internal action $.supervise(SupSet', SupPolicy)$ by the agent $superv'$ at the immediately preceding time t_0 . As $SupRel$ already existed at the time t_0 , then the condition $sup_{ed} \in SupSet' \wedge (sup_{ed}, \dots) \in SupSet$ holds at t_0 , which falsifies the preconditions of the transition rule $supervision1$ for creating the new relation $SupRel'$ (there is only one transition rule that successfully establishes a new supervision relation). Thus, a multi-agent system such as M which contains multiple supervision policies over an agent cannot exist.

Theorem 2 guarantees that the different supervision policies maintained in the system cannot conflict. For instance, it is not possible for any supervision mechanism to decide that some agent, which exceeds its ping threshold, must be both restarted and unblocked at the same time.

Chapter 6

INTERPRETING eJASON PROGRAMS

This chapter provides useful insights on the interpretation of eJason programs. First, it describes several aspects of the eJason interpreter. The implementation of this interpreter represents one of the main contributions of this thesis. This activity has been the most time-consuming one due to its high level of complexity (the code comprises more than 20000 lines of Erlang code). Then, several examples that illustrate the potential use of the capabilities of eJason follow, as well as recommendations on the suitability of each of them.

Section 6.1 begins with a high-level overview of the design of the interpreter. This overview describes all the steps involved in the interpretation of eJason programs, as well as the components of the interpreter involved. Then follows a detailed enumeration of the scope of the implementation, i.e. a list of all the features that have been implemented into the interpreter and made fully available for their use by eJason programmers. Then, Subsection 6.1.3 describes how the most important parts of the interpreter have been implemented in Erlang. This description also addresses why the implementation does not simply reuse some of the Erlang built-in features like, e.g., process monitors. The audience of this subsection is a reader familiar with the Erlang programming language, as some of the Erlang features referred are beyond the scope of this thesis and would require an extensive explanation. The average reader can skip this section and still fully understand the extent of the contribution represented by the implementation of the eJason interpreter. Finally, the section describes how the eJason interpreter can be installed as well as some initial steps to run a simple eJason program.

Section 6.2 provides a series of use cases that expose the behaviour and performance of the eJason interpreter. These use cases address different features of eJason, ranging from computationally intensive benchmarks to recommendations on the use of the capabilities of eJason.

6.1 The Interpreter of eJason

This section provides some insights on the implementation of the eJason interpreter as well as instructions on how to install and run it.

6.1.1 High-Level Design

The interpreter of eJason is implemented in the Erlang programming language. It is composed by the following two main components:

- **eJason Parser:** The parser transforms eJason source code (i.e. a series of agent programs written into files with “.asl” extension) into Erlang source code (i.e. a set of module files with “.erl” extension). The implementation of this parser comprises more than 2000 lines of Erlang code.
- **eJason Runtime System:** The runtime system enables the compilation and execution of the Erlang source code corresponding to an eJason program. It is composed by a set of Erlang modules specialised on different parts of the language, e.g. variable handling, the implementation of the belief base of eJason agents or the implementation of the reasoning cycle of eJason, among others. These modules comprise more than 18000 lines of Erlang code.

As these components are implemented in Erlang, they run on top of BEAM, the **Erlang Virtual Machine**.

The main steps that are carried out in order to interpret an eJason program are depicted in Figure 6.1. These steps, which correspond to the labelled arrows in the figure, are:

- 1) The eJason Parser transforms the “.asl” files of an eJason program into the “.erl” files of its Erlang equivalent.
- 2) The corresponding functionality of the eJason Runtime System compiles these “.erl” files into “.beam” files (Erlang bytecode).
- 3) The Erlang Virtual Machine executes the instructions specified by the “.beam” files, using the corresponding modules of the eJason Runtime System.

Figure 6.1: High Level Design of the eJason interpreter

6.1.2 Scope of the implementation

The implementation of the eJason interpreter includes the following features of the Jason programming language described in [Bordini et al., 2007] and described in Section 2.1:

- All the terms of the language, including **atoms**, **strings**, **lists**, **variables (bound and unbound)** and **predicates**.
- All the classical **arithmetical operations** with operators: '+' (addition), '-' (subtraction), '*' (multiplication), '**' (power), '/' (division), 'div' (integer division) and 'mod' (remainder).
- All the classical **relation expressions** with operators: '<' (smaller than), '>' (greater than), '<=' (smaller than or equal to), '>=' (greater than or equal to), '==' (equality), '\==' (difference) and '=' (unification).
- All the classical **logical expressions** with operators: & (conjunction) and | (disjunction).
- The **decomposition** of a literal (atoms and predicates) into a list with the operator '...'. For instance: $p(x, y)[a, b, c] = ..[p, [x, y], [a, b, c]]$ and $atom = ..[atom., [], []]$.

- The **negation** of a predicate with the operator **not**. For example, the expression *not b* is true if it is not possible to infer that *b* is true using the information in the agent's belief base (i.e. *the agent does not believe b to be true*).
- The **strong negation** of a predicate with the operator $\sim$. The expression $\sim b$ meaning that using the information in the agent's belief base, *b* is false (i.e. *the agent does believe b to be false*)
- All the components of an **agent's mental state**:
 - The agent's **belief base**, along with all the operations necessary to update it by adding or removing **beliefs and rules** from it. The implementation of the belief base includes an **inference engine** that enables the resolution of inference rules (typically leading to the unification of variables).
 - The set of **agent goals**, including both achievement goals and test goals.
 - The **annotation of beliefs and goals**, including the automatic annotation that determines the **source of a belief or a goal**, is also implemented.
 - The set of **agent events**, including the events corresponding to the addition of beliefs and goals, the removal of beliefs and the failure events generated when either there are no applicable plans for a test goal addition or because the execution of a plan body fails (e.g. a forbidden unification attempt).
 - The set of **agent plans**. The triggering events allowed are the ones in the paragraph above. The context of these plans can be any logical expression that uses the operators listed above. The body of these plans can be goal additions, mental notes, variable assignments, conditions (relation or logical expressions) and internal actions.
 - The set of **agent intentions** as a stack of partially instantiated plans.
- The **communication performatives** *tell* (belief addition), *untell* (belief deletion) and *achieve* (achievement goal addition).
- The **agent's reasoning cycle** with the corresponding computations required to carry out its different steps, including the implementation of the default functions for event, option (plan) and intention selection.

Regarding the extensions introduced by eJason and described throughout the document, the following features are implemented by the eJason interpreter:

- The reasoning cycle implements all the **control mechanisms** introduced in Chapter 3:
 - The atomic execution of **critical sections**, i.e. the sequence of instructions enclosed within the symbols `{` and `}`, in a plan body.
 - The **consecutive evaluation and execution of a plan**, strongly coupling the evaluation of a plan context and the execution of its body when it starts with a critical section.
 - The **events for goal addition are queued** whenever they are selected but there are no applicable plans for them.
 - The events for a goal added with **the operator** `??` are queued when they are selected but there are no applicable plans for them.
- All aspects of the **distribution mechanism** described in Section 4.1:
 - The **agent containers** have unique names in the system. They support **merge and split operations**.
 - The **eJason agents** can be distributed on the different agent containers and possess a **unique name on the system**. Agents on different containers can interact (i.e. communicate, monitor, create other agents, ...) using their unique name.
- All aspects of the **monitoring mechanism** described in Section 4.2.1 are available for their use:
 - An agent can monitor the status of another agent irrespectively of the container(s) that host(s) them. This relation is unique and persists until the conditions for its extinction (specified by the programmer) are met or until it is explicitly disabled.
 - The changes on the execution state of a monitored agent that are detected and notified to the monitoring agent are: its death, its unreachability, its inexistence and its creation.
- The eJason interpreter implements the following aspects of the **supervision mechanism** of agents:
 - The establishment of a supervision relation between a supervisor agent and set of supervised agents. This mechanism has been thoroughly tested when all the agents involved belong to the same container. Distributed supervision trees are less reliable at the moment of writing.

- The enforcement of all the supervision policies (i.e. the ping policy, the unblock policy, the restart policy and the revival policy) and their corresponding corrective actions, along with a default restart strategy of `one_for_one`.

Due to time constraints the mechanisms to disable a supervision relation have not been implemented. Notice that this feature is much less relevant to show the contributions of the interpreter than the ability to establish supervision relations as well as the ability to enforce their corresponding supervision policies.

- The following **internal actions** are available: `.connect_to`, `.containers_list`, `.container_of_agent`, `.create_agent` (its implementation ignores the optional parameter `InitialBeliefs`), `.demonitor_agent`, `.disconnect_from`, `.find_agent`, `.kill_agent`, `.monitor_agent`, `.my_container`, `.my_name`, `.print`, `.send`, `.supervise_agents` and `.wait`.

6.1.3 Overview of the implementation

This section assumes the moderate familiarity of the reader to Erlang programming.

The eJason parser has been implemented using the parser generator Yecc¹. This generator is implemented in Erlang and takes as input a grammar in the well-known Backus-Naur Form (BNF). Therefore, a grammar for eJason, similar to the one presented in Section 5.1.1, has been created.

Each agent container in an eJason multi-agent system corresponds to a different Erlang node. The name of a container and its Erlang node are the same. Therefore, the words node and container are used indistinguishably in the remainder of this section. The following Erlang processes run on *each* of these Erlang nodes:

- A process registered locally as *AgentName* for each eJason agent running in the container, where *AgentName* is the unique name of the eJason agent. Therefore, the eJason agent and its corresponding Erlang process are namesakes. This process executes the reasoning cycle of the corresponding agent.
- A process registered locally as `ejason_distribution_manager` (using the function `erlang:register(Name, Pid)`). It is the distribution manager.

¹See <http://erlang.org/doc/man/yecc.html> for more information on this LALR-1 parser generator

This process handles all the procedures necessary to implement the eJason distribution with regards to its local node. It carries out all the necessary steps to spawn new eJason agents. It provides a white pages service that works as a local name registry to enable the communication from the agents in the container. This process also records the list of containers that are connected to its local one. This manager communicates with its homonym processes in the connected containers and is the main actor in the split and merge operations.

- A process registered locally as `ejason_supervision_manager` (using the function `erlang:register(Name, Pid)`). It is the supervision manager. This process manages all the procedures related to the monitoring and supervision of eJason agents. It records all the monitoring and supervision relations involving agents in its local container. It is the element in charge of detecting execution changes in its local monitored agents and start the procedure to notify the monitoring agents (both locally and remotely by sending messages to its namesake in the corresponding container). The supervision manager also enforces all the corrective actions of the supervision policies enforced over the agents in its container.

The following considerations regarding the Erlang implementation of the eJason components are worth noting:

- The two manager processes above represent a fundamental part of the eJason Runtime System. If any of them is stopped in some container, that container cannot provide its functionality and can be considered crashed.
- The Erlang node registry is only used as a convenience to implement the communication between eJason agents. A message sent between a sender agent to a any reachable receiver agent will trigger the exchange of an Erlang message between their corresponding Erlang processes.

Notice that not every process registered in the node registry of an agent container is an eJason agent. Only those processes that are registered in the white pages service of their local distribution manager are considered eJason agents. These Erlang processes are guaranteed to abide to the rules of eJason agents, i.e. implement the reasoning cycle of eJason.

- The connection between eJason containers uses the Erlang function `net_adm:ping/1`. For any given two connected containers, their corresponding Erlang nodes are connected as well. However, not all the Erlang nodes connected to them are considered agent containers. Only those nodes that contain a running distribution and supervision managers can be considered agent containers. Additionally, the connection between two containers must be registered by the corresponding distribution managers.
- The monitoring of two agents does not establish an Erlang monitor between their corresponding Erlang processes. This monitor is established between the supervision manager and the process for the monitored agent, using the Erlang function `erlang:monitor/2`. The supervision manager handles the monitor notifications and acts accordingly. This manager implements mechanisms to maintain alive monitoring relations that do not have an Erlang counterpart (e.g. when the monitored agent is unreachable or unknown).
- The supervision mechanism relies on the aforementioned Erlang monitor between the supervision manager and the processes for the agents in the supervised set in order to detect agent deaths and restarts. The different subpolicies that compose the supervision policy are implemented in the following way:
 - The ping policy is implemented by a different Erlang process (namely a *ping watcher*, that executes the code in the Erlang module `ping_policy_watcher`, that periodically pings the supervised agents. These pings are Erlang messages of the form `{signal, {ping, _}}` that the Erlang processes corresponding to the supervised agents always process at the beginning of each iteration of their reasoning cycle (i.e. they take precedence over all other messages in the agent's mailbox at the beginning of the iterations of the reasoning cycle). The agents that successfully respond to a ping respond with an Erlang message of the form `pong`. When the ping watcher detects the divergence of an agent, it notifies the local supervision manager in order to enforce the corresponding unblock policy.
 - The unblock policy is implemented by the supervision manager. When a supervised agent is suspected of divergence, the supervision manager sends a message of the form `{signal, ejason_unblock}` to the Erlang process of the supervised agent. During the evaluation of plan contexts and rules in the belief base, the agents frequently look for such messages. If they find this message, the resolution is cancelled as if all possible valuations had been tested, hence unblocking any possible infinite loop.

- The restart policy is carried by the supervision manager and amounts to killing the Erlang process of the supervised agent with the function `erlang:exit/2`). A new Erlang process is started immediately afterwards without a release of the agent name, hence avoiding any name clash that may happen in the meantime.
- The revival of an agent is identical to the procedure before, excepting that the agent is not killed by the supervision manager.

6.1.4 Installing and Running eJason

The eJason interpreter has been developed and tested on a server running Ubuntu Linux version 14.04 LTS (x686-64). The Erlang release installed on the system is Erlang/OTP 19². The eJason interpreter is guaranteed to run on a machine with these characteristics and is likely to run on later releases of the Operation System and Erlang.

The latest release of the eJason interpreter can be downloaded from the online repository at <https://github.com/avalor/eJason/>. On an Ubuntu system, it suffices to run:

```
> git clone https://github.com/avalor/eJason/
Cloning into 'eJason'...
remote: Counting objects: 248, done.
remote: Total 248 (delta 0), reused 0 (delta 0), pack-reused 248
Receiving objects: 100\% (248/248), 267.89 KiB | 275 KiB/s, done.
Resolving deltas: 100\% (101/101), done.
```

This creates a new folder *eJason* that contains all the code of the interpreter. The instructions to setup and run the interpreter are provided in the *README* file.

At the moment of writing this thesis work, the following instructions apply. To compile the eJason interpreter source code, navigate into the eJason folder and type:

```
make
```

²Erlang can be downloaded from, e.g., the official Ubuntu Software Repositories or, for a specific version, the webpage <https://www.erlang-solutions.com/resources/download.html>

This generates a set of “.beam” files (Erlang bytecode) inside the folder “./eJason/ebin”.

The folder “./eJason/agents” must contain all the “.asl” files with the program for the agents to run. This folder is included by default in the eJason path. To start an eJason system, execute the script *ejason* as follows:

```
./ejason [-c ContainerName]
```

The optional argument `ContainerName` is the name of the container that composes the newly created system. If the eJason multi-agent system is to be distributed on several machines, this parameter becomes mandatory. As an example, consider the command:

```
./ejason -c myContainer
```

Its execution will generate a container `myContainer@userHost`, where `userHost` is the name of the host where the command was executed.

The *eJason* script starts an Erlang shell invoking the command `erl`. Therefore, it also accepts all parameters that the `erl` command accepts (see <http://erlang.org/doc/man/erl.html>).

The *ejason* runtime system, including the distribution manager and the supervision manager, is automatically started.

The source code of the eJason agents (i.e. the ‘.asl’ files) that run in eJason must be placed in a directory belonging to the search path of eJason. By default, the folders “/eJason/ebin”, “/eJason/include” and “/eJason/agents” belong to it. Additional directories `Dir1... DirN` can be added via the parameter “-pa `Dir1 ... DirN`” to the *ejason* script.

To compile and run an agent (provided that its source code is in the eJason search path), the following command must be run after starting the eJason interpreter:

```
> ejason:(agentName).
```

To run the first simple eJason program, copy the file “/eJason/tests/counter/-

counter.asl” into “ejason/agents”. Then, run the following sequence of commands:

```
./ejason -c container1
```

```
ejason:crun(counter).
```

Note that the first command starts an agent container (i.e. an Erlang node) named `container1`. The second command (introduced in the Erlang shell) compiles and runs the eJason code for the agent `counter`. This agent does not stop running and it can be killed by using this command in the Erlang shell:

```
ejason:kill(counter).
```

6.2 Examples

This section provides a series of example Jason programs whose implementation and execution provide a benchmark to compare the performance of the eJason interpreter (version 1.0) and the Java implementation of Jason (version 2.0). All the tests described have been run on a 64-bit, 2GHz quad-core laptop with 8GB RAM using Erlang/OTP 19 and Java 1.7.0_111 on top of Ubuntu Linux 14.04 LTS. Unless otherwise specified, version 2.0 of Jason (release date 15/09/2016) is used in the different benchmarks. The use cases at the end of the section describe the potential use of the mechanisms introduced by eJason.

6.2.1 Peano Arithmetic

The first use case shows the performance of eJason when dealing with simple, yet computationally intensive, monolithic code. The Jason code tested implements an agent (namely Peano agent) whose belief base consists of inference rules that compute Peano arithmetic. These inference rules carry out Peano addition and multiplication, as well as the computation of the immediately next number (i.e. subsequent) of a given one. This code is shown in Figure 6.2 below.

Note that this code does not include any of the extensions introduced by eJason, hence allowing a comparison of the performance of the eJason interpreter and the

```

// Peano subsequent (eq. decimal +1)
next(0,succ(0)).

next(succ(X),succ(Y)) :-
    next(X,Y).

// Peano addition

sum(X,succ(Y),Z):-
    sum(succ(X),Y,Z).

sum(X,0,X):-
    true.

// Peano multiplication
mult(X,0,0):-
    true.

mult(0,Y,0):-
    true.

mult(succ(X),Y,Z):-
    mult(X,Y,Res1) & sum(Res1,Y,Z).

```

Figure 6.2: Inference rules of the Peano agent

Java implementation of Jason. Besides, for the sake of clarity, inference rules to convert numbers from/to Peano numbers are also implemented. The Jason code for these rules is displayed in Figure 6.3.

The different tests run are variations of the plan for Peano addition in Figure 6.4 (resp. in Figure 6.5 for Peano multiplication) which, in a nutshell, consists of the following steps:

- The initial goal initialises the variables *X* and *Y* with decimal numbers (varied in the different tests).

```
// Converts a Peano number into its decimal equivalent
peano_to_dec(PeanoNum,DecNum):-
    peano_to_dec(PeanoNum,DecNum,0).

peano_to_dec(0,DecNum,DecNum):-
    true.

peano_to_dec(succ(PeanoNum),DecNum,Acc):-
    NewAcc = Acc + 1 &
    peano_to_dec(PeanoNum,DecNum,NewAcc).

// Converts a decimal number into its Peano equivalent

dec_to_peano(DecNum, PeanoNum):-
    dec_to_peano(DecNum, PeanoNum, 0).

dec_to_peano(0, PeanoNum, PeanoNum):-
    true.

dec_to_peano(DecNum, PeanoNum, Acc):-
    NewDecNum = DecNum - 1 &
    dec_to_peano(NewDecNum, PeanoNum, succ(Acc)).
```

Figure 6.3: Inference rules implementing number conversion

- Convert the variables above into their Peano equivalents.
- Carry out the addition (resp. Peano multiplication) of the Peano variables.
- Convert the results of the addition (resp. multiplication) into decimal values and print the result on the standard output.

Note that these plans are instrumented with the internal action `.time` to properly measure the execution time (with a precision of seconds) of each test run.

The first batch of Peano addition tests was run with a fixed value of 10 for variable `X` and incremental values for variable `Y`. The results of these tests are

```
// Initial goals and plan base

!start(10, 10). // Initial goal

+!start (X, Y) : true <-
    .time(SH, SM,SS);
    .print("Start Time: ", SH,":",SM,":",SS);
    ?dec_to_peano(X, PeanoX);
    ?dec_to_peano(Y, PeanoY);
    ?sum(PeanoX,PeanoY, PSum);
    ?peano_to_dec(PSum, DSum);
    .print(X, " + ", Y," = ",DSum);
    .time(EH, EM, ES);
    .print("End Time: ", EH,":",EM,":",ES).
```

Figure 6.4: Jason plan that runs the different Peano addition tests

```
// Initial goals and plan base

!start(10, 10). // Initial goal

+!start (X, Y) : true <-
    .time(SH, SM,SS);
    .print("Start Time: ", SH,":",SM,":",SS);
    ?dec_to_peano(X, PeanoX);
    ?dec_to_peano(Y, PeanoY);
    ?mult(PeanoX,PeanoY, PMult);
    ?peano_to_dec(PMult, DMult);
    .print(X, " * ", Y," = ", DMult);
    .time(EH, EM, ES);
    .print("End Time: ", EH,":",EM,":",ES).
```

Figure 6.5: Jason plan that runs the different Peano multiplication tests

Y	Jason (seconds)	eJason (seconds)
10	1	1
100	1	51
200	1	639
300	1	3024
400	1	9307
1000	3	Aborted
1025	3	Aborted
> 1025	Stack Overflow	1.13

Table 6.1: Execution time of the Peano addition test batch (X=10)

shown in Table 6.1. Note that the execution time of the eJason tests increases exponentially (test running for 12 hours were aborted), while that for Jason ones increases linearly until a maximum of Y=1025. Higher values immediately yield a Java Stack OverFlow exception.

Similarly, a batch of Peano multiplication tests was run, again fixing the value X=10 and varying the value for variable Y. The results of these tests are shown in Table 6.2. Note that the execution time of the eJason tests increases exponentially, while that of the Jason ones slightly increases until a maximum of Y=180. Higher values yield a Java Stack OverFlow exception.

During the execution of the eJason tests, the consumption of computational resources by the Erlang Virtual Machine peaked to 100% of one core (i.e. 25% of the total CPU) and 1% of the Virtual Memory. Such resource usage is consistent with the multi-agent system run, as there is one agent (i.e. the Peano agent) carrying out most of the computations, hence not leveraging the underlying multi-core hardware architecture. This usage differs from the Jason tests, where the computational load of the Java Virtual Machine spread to all 4 cores (i.e. about 80% of the total CPU) and 8% of the Virtual Memory.

The figures corresponding to the tests above show how the Java implementation of Jason greatly outperforms eJason when the agent is intended to carry out computationally intensive tasks that do not involve any interaction (e.g. communication) to other agents. This difference in the performance may be alleviated by implement-

Y	Jason (seconds)	eJason (seconds)
10	1	17
20	1	126
40	1	1088
40	1	3334
50	1	8287
100	1	Aborted
150	2	Aborted
180	5	Aborted
> 180	Stack Overflow	1.13

Table 6.2: Execution time of the Peano multiplication test batch (X=10)

ing code optimisation techniques into eJason, like, e.g., the use of more efficient data structures (for instance, replacing the intensive use that eJason makes of Erlang’s `orddicts` and `ordsets` modules by the use of the more efficient `dicts` and `sets` modules instead)³ or the improvement of tail recursion. Note that the optimisation of the eJason interpreter, especially for monolithic code, lies beyond the scope of this thesis work. Instead, it is one of the claims in Chapter 1 that the efficient concurrent and distribution mechanism of the Erlang programming language carry over to eJason, as shown in the following section.

6.2.2 Token Ring

The second use case implements a classical token ring architecture similar to the one described in [Cardoso et al., 2013]. In a nutshell, this system is composed by W agents (namely workers), arranged so as to compose a ring, that pass a token between them until N cycles are completed. The Jason implementation of this system is composed by the following agents:

- a) A total of W worker agents whose behaviour amounts to receiving a token and passing it to the next worker a maximum of N times.
- b) A `tokenRing` agent that creates and arranges all the worker agents in the token ring and sends the token to the first agent.

³These Erlang modules are described at the Erlang OTP Reference Index <http://erlang.org/doc/man/>

The Jason code of the **worker** agent is included in Figure 6.6. The initial goal of these agents triggers a plan that inform the **tokenRing** agent about the successful creation of the worker, as well as its own name, with a corresponding belief addition event (see the description of the code of the **tokenRing** agent below). The belief base of the workers contains a belief **next_worker(AgentName)** that determines the next agent in the ring, obtained from the **tokenRing** agent, in order to forward the token to it. Additionally, the first worker also contains a belief **first_worker**. The token is represented by a belief **token(N)**, such that **N** determines the remaining cycles that the token must travel (i.e. the belief **token(0)** represents the token during its last cycle). The action of passing the token is represented by the emission of the corresponding message between agents. The plan base of the different **worker** agents contains four plans triggered by the addition of the belief representing the token (i.e. their triggering event is of the form **+token(N)**) that, depending on the context, implement the following behaviour in their body (in appearance order):

- If the token is not in its last cycle and the worker is not the first worker in the ring, send the token to the next worker (hence removing the corresponding belief).
- If the token is not in its last cycle and the worker is the first worker in the ring, decrement the variable that represents the remaining cycles of the token. Then forward the token to the next worker.
- If the token is in its last cycle and the worker is not the first worker in the ring, forward the token to the next worker. Finally, the agent kills itself.
- If the token is in its last cycle and the worker is the first worker in the ring, inform the **tokenRing** agent by telling it a belief **finished**. Finally, the agent kills itself.

The Jason code of the **tokenRing** agent is included in Figure 6.7. The initial belief base of this agent contains two beliefs that determine the number of agent workers in the ring (**numberOfWorkers**) and the number of times that the token is passed between them (**numberOfCycles**). The initial goal of this agent (**createRing**) is to set up the token ring. The agent's plan base consists of the following plans (in appearance order):

- The plan **!createRing** records the start time of the execution and adds the goal to start creating worker agents.
- The plans **!createWorker** check in their context whether all worker agents have already been created and depending on the outcome either:

- Decrement the counter of pending worker agents and spawn a new worker agent.
- Record the time invested in the creation of the token ring and send the token (via a Jason message) to the first worker.
- The plans `+new_worker[source(AgentName)]` are triggered whenever a new worker notifies its successful creation. Note that the annotation `source(AgentName)` allows the `tokenRing` agent to obtain the name of this newly created worker. In order to construct the ring, the plan selected as intended means varies depending on the plan context as follows (in appearance order):
 - If no worker has already been registered as the first one, the newly created worker is registered as the first worker and notified about this fact. Then, the `tokenRing` agent registers this new agent as the `previous_worker` and add the goal to create a new worker.
 - If there are more pending workers, notify the previous worker that this newly created worker is its next one (i.e. adds it to the ring). Finally register this new worker as the `previous_worker` and add the goal to create yet another worker.
 - If there are no more pending workers, notify the newly created agent that its next worker is the one registered as `first_worker` (hence closing the ring). Finally, add the goal to create a new worker (which will actually match the plan context that starts passing the token).

The different `tokenRing` tests were run with a fixed value of 50 for variable `W` (i.e. 50 workers) and incremental values for variable `N`. As shown in Table 6.3, these tests were run with different versions of Jason. These additional versions were the ones used for benchmarking during the different phases of the development of the present thesis work and are included as a witness of the great performance speedup introduced from Jason 1.3.10 onwards. Note that the execution time of both Jason (from version 1.3.10) and eJason tests increases linearly with respect to the number of cycles `N`. These versions yield different Out of Memory exceptions at high values. Note that the execution time of Jason version 1.3.9 grows exponentially with respect to the number of cycles `N` until it yields a Java Stack Overflow exception.

The experiments run show that the latest releases of Jason (i.e those above version 1.3.9) outperform eJason in the Token Ring scenario described above. However, the difference in the performance is much smaller than the one observed

```

// Worker in the tokenRing benchmark system

!start. // Initial Goal

+!start<-
// Notify the tokenRing agent
.send(tokenRing, tell, new_worker).

// Pass the token (not first worker)
+token(N): N > 0 & not first_worker & next_worker(Next)<-
    -token(N);
    .send(Next, tell, token(N)).

// Pass the token (first worker, then decrease
// the token cycle)
+token(N): N > 0 & first_worker & next_worker(Next)<-
    -token(N);
    M = N -1;
    .send(Next, tell, token(M)).

// Pass the token (last cycle, but not first worker)
+token(0): not first_worker & next_worker(Next) <-
    .send(Next, tell, token(0));
    .my_name(Self);
    .kill_agent(Self).

// Pass the token (last cycle and first worker)
+token(0): first_worker & next_worker(Next)<-
    .send(tokenRing, tell, finished);
    .my_name(Self);
    .kill_agent(Self).

```

Figure 6.6: Complete Jason code of the worker agents

in the Peano use case described before. This Token Ring use case shows the remarkable performance improvement that the Java implementation of Jason has undertaken of late, which resembles a similar improvement of the Java Virtual

```

// Number w of workers in the Ring
numberOfWorkers(50).

// Number of cycles (N)
numberOfCycles(10).

// Initial Goal: create the Token Ring
!createRing.

// Plan for the initial goal
+!createRing: numberOfWorkers(W) <-
    .time(H,M,S);
    .print("Start Time: ",
        H,":",M,":",S);
    +remaining_workers(W);
    !!createWorker.

// Not last worker yet
+!createWorker: remaining_workers(W) &
    W > 0 <-
    PendingWorkers = W - 1;
    -+remaining_workers(PendingWorkers);
    .create_agent(Worker, "./worker.asl").

// Last worker created, send the token
+!createWorker: remaining_workers(W) &
    W < 1 & first_worker(First) &
    numberOfCycles(N) <-
    .time(H,M,S);
    .print("Workers Created: ",
        H,":",M,":",S);
    .send(First, tell, token(N)).

// Message from the first worker
+new_worker[source(NewWorker)]:
    not first_worker(First)<-
    +first_worker(NewWorker);
    +previous_worker(NewWorker);
    .send(NewWorker, tell,
        first_worker);
    !!createWorker.

// Message from a worker that is
// neither the first nor the last
+new_worker[source(NewWorker)]:
    first_worker(First) &
    remaining_workers(W) & W > 0 &
    previous_worker(Previous) <-
    -new_worker[source(NewWorker)];
    .send(Previous, tell,
        next_worker(NewWorker));
    -+previous_worker(NewWorker);
    !!createWorker.

// Message from the last worker
+new_worker[source(NewWorker)]:
    first_worker(First) &
    remaining_workers(0) &
    previous_worker(Previous) <-
    .send(Previous, tell,
        next_worker(NewWorker));
    -previous_worker(NewWorker);
    //Close the ring
    .send(NewWorker, tell,
        next_worker(First));
    !!createWorker.

+finished <-
    .time(H,M,S);
    .print("Finish Time:",
        H,":",M,":",S).

```

Figure 6.7: Complete Jason code for the tokenRing agent

Machine with regards to its support to the efficient execution of multithreaded programs. The resource consumption of the Java Virtual Machine spread among two cores (around 30% of total CPU consumption) and took the 10% of the Virtual Memory. The resource consumption of the Erlang Virtual Machine only span to a single core (25% of total CPU consumption) and 2% of the Virtual memory.

This Token Ring scenario, while communication-intensive, is not highly concurrent, as all worker agents are idle excepting the one that receives and forwards the token. This phenomena is again consistent with the resource consumption described above for the Java and Erlang Virtual Machines. Moreover, after an

N	Jason 2.0 (s)	Jason 1.3.10a (s)	Jason 1.3.9 (s)	eJason (s)
10	1	1	1	1
100	1	1	1	2
1000	2	2	62	16
10000	18	22	4425	164
50000	83	88	Stack Overflow	843
75000	134	130		Out of Memory
95000	206	199		
97500	Out of Memory	Out of Memory		

Table 6.3: Execution time of the Token Ring test batch (W=50)

agent receives the token, the corresponding intention does not compete with any other intention of the agent. Hence maintaining its focus of attention. Therefore, in order to simulate a more realistic scenario (i.e. one that addresses the multiplicity of foci of attention inherent to Jason agents) a variation of this Token Ring usecase is considered. In this variation, the worker agents apart from passing the token, tend to other chores. The code of this use case is identical to the one above except for the replacement of the plan `+:start` in the plan base of the worker agents as well as the addition of the new plan (a simple one that just keeps the agent busy with easy computations), included in Figure 6.8.

```
+:start<-
// Notify the tokenRing agent
.send(tokenRing, tell, new_worker);
.wait(1000); // Do not interfere with the ring creation
!!loop. //the agent is kept busy with other tasks

// Simple computations that keep the agent busy
+:loop <-
  A = 10;
  B = 10;
  C = A + B;
  !!loop.
```

Figure 6.8: Variations to the Worker agents for the second test batch

The execution of this variation of the Token Ring produced the results shown in Table 6.4. This new test batch shows that eJason outperforms even the latest release

W	N	Jason 2.0 (s)	Jason 1.3.10a (s)	eJason (s)
50	10	1	1	7
50	100	82	99	71
50	1000	2584	2631	701
50	10000	26244	26728	7288
100	10	1	1	25
100	100	1095	987	291
100	1000	13461	12483	3325
100	10000	Aborted after 24h	Aborted after 24h	31223

Table 6.4: Execution time of the second Token Ring test batch

of Jason (version 2.0 at the moment of writing) in this example, which involves not only communication, but a higher degree of concurrency as it involves not only the interaction between agents due to message-passing, but also their competition for processor time. This enhanced performance of the eJason agents stems from the efficient implementation of the underlying Erlang light-weight processes, whose high-performance is leveraged by eJason. The resource consumption of the Java Virtual Machine span to all four cores (around 96% of total CPU consumption) and required the 8% of the Virtual Memory available. The resource consumption of the Erlang Virtual Machine span to all four cores (around 99% of total CPU consumption) and required the 0.5% of the Virtual Memory available. Nevertheless, the purpose of these benchmark is to provide some insights on the performance of the eJason interpreter but shall not be overweighted, as an increase in the performance of the Multi-Agent Systems run by eJason is not an objective of this thesis work, but a desirable feature.

6.2.3 Fault Tolerant Client-Server System

This use case shows how the monitoring mechanism of eJason can be used to implement a fault-tolerant instance of a classical client-server system. It addresses issues such as the identification of the degradation of the execution state of agents (i.e. deaths) as well as the recognition of agents after network disconnections. Moreover it includes requirements that preserve the Quality of Service of this system.

Consider a classic system composed by a server agent, also named **server**, that processes requests from an arbitrary number of (client) agents. These agents are distributed in a number of agent containers that varies over time (i.e. there can be split and merge operations). The requirements of this system are:

- Each client must start a (stateful) connection session with the server before it can start sending requests. If a session for that client already exists, the state of the session is not restarted but resumed.
- A state is maintained by the server for each session. The state of a session evolves after the processing of each of the requests corresponding to that session.
- A session terminates automatically if either the client or server agent dies. A session must persist after disconnections of clients and be reactivated once the connection is restored. This session is temporarily suspended (e.g. liberating resources somehow allocated to the session, for efficiency reasons).
- The service provided by **server** must be implemented so that a best-effort is carried out to ensure the least possible downtime without replicating the server. Besides if the response time of the **server** grows over 10 seconds, the server must be restarted.

To fulfil the requirements of this system, the following issues must be solved:

- a) The server and client agents must be aware of the deaths of each other in order to terminate their corresponding sessions.
- b) The disconnection from a client does not terminate a session, but suspends it until the reconnection takes place.
- c) After a reconnection, the server must *recognise* the client, (i.e. check whether the agent is the same agent that started the session or if it is a new agent that uses the same name) as it is possible for an agent to be restarted or revived after the disconnection.
- d) The clients need to be aware of a reconnection to the server and request the start of a session when the reconnection happens in order to resume it.
- e) The behaviour of the server must be observed in order to guarantee that it meets its service level agreement. These requirements address the minimum availability of the server as well as its maximum response time.

The code provided in Figures 6.9 and 6.10 provides the skeleton of a client-server system that solves the issues listed above. Note that there is no requirement in the behaviour of the clients (other than that imposed on the communication by the server's plan base), hence a sample one is implemented for illustrative purposes. For simplicity, the business layer of the system is abstracted (i.e.

```

//server_rule#1
check_id(Client, ClientID, new_client):-
    not state(Client, SomeID, State).

//server_rule#2
check_id(Client, ClientID,
    restarted_client):-
    state(Client, CID, State) &
    ClientID \== CID.

//server_rule#3
check_id(Client, ClientID, same_client):-
    state(Client, ClientID, State).

//server_plan#1
+!start_session(ClientID)
[source(Client)] <-
.print("New Request from ", ClientID);
?check_id(Client,ClientID,
    Result);
!process_start(Client,ClientID,
    Result).

//server_plan#2
+!process_start(Client,ClientID,
    new_client) <-
    .monitor_agent(Client,
        persist(unreachable_agent));
{{!init_session(Client, InitState);
+state(Client,ClientID, InitState)}};
.send(Client,tell,state(InitState)).

//server_plan#3
+!process_start(Client,ClientID,
    restarted_client)<-
    !delete_session(Client);
{{
!init_session(Client, InitState);
-state(Client, _, _);
+state(Client,ClientID,
    InitState)}};
.send(Client,tell,state(InitState)).

//server_plan#4
+!process_start(Client,ClientID,
    same_client):
    state(Client, ClientID,
        State) <-
        .send(Client,tell,
            state(State)).

//server_plan#5
+!request(ClientID,Request)
[source(Client)]:
    state(Client,ClientID,
        State)<-
        !process_request(State,
            Request, NewState);
        {{-state(Client, ClientID,
            State);
        +state(Client, ClientID,
            NewState)}};
        .send(Client,tell,
            state(NewState)).

//server_plan#6
+agent_down(Client)
[reason(unreachable_agent)] <-
    !suspend_session(Client).

//server_plan#7
+agent_down(Client)
[reason(Reason)]:
    Reason == dead_agent |
    Reason == restarted_agent |
    Reason == revived_agent<-
    !delete_session(Client).

//server_plan#8
+agent_up(Client) <-
    !reactivate_session(Client).

```

Figure 6.9: Code for the server agent

the nature of the session state and the set of requests accepted by the server). This abstraction corresponds to the actual implementation of the relevant plans for the achievement goals `!init_session`, `!delete_session`, `!suspend_session`, `!reactivate_session` and `!process_request`. A sample implementation of these plans is provided at the end of this section. The different issues listed above are solved by the eJason code as described next:

```

!start.
//#client_plan#1
+!start <-
    .time(H,M,S);
    .my_name(Name);
    .my_container(Container);
    //Client's ID
    ID = id(H,M,S,Name, Container);
    +my_id(ID);
    .monitor_agent(server,
        demonitor(dead_agent));
    .send(server,achieve,
        start_session(MyID)).

//#client_plan#2
+state(NewState)[source(server)]:
    my_id(MyID) <-
        +state(NewState);
        !prepare_request(NewState,
            Request);
        -+last_request(Request);
        .send(server,achieve,
            request(MyID,Request)).

//#client_plan#3
+agent_down(server)[reason(Reason)]:
    Reason == restarted_agent |
    Reason == revived_agent <-
        .send(server,achieve,
            start_session(MyID)).

//#client_plan#4
+agent_down(server)[reason(dead_agent)]:
    my_name(Name) <-
        .kill_agent(Name).

//#client_plan#5
+agent_up(server) <-
    .send(server,achieve,
        start_session(MyID)).

```

Figure 6.10: Code for the client agents

- a) The monitoring mechanism of eJason solves this issue. Upon reception of a request to start a session from a client (**server_plan#1**), and after determining that a new session must be started, the server establishes a monitoring relation with the new client (**server_plan#2**). Note that this relation persist after disconnections of the client. The server agent monitors then all the client agents that have started a session and is therefore able to automatically detect not only their death, but also their restart or revival, in order to remove the corresponding session (**server_plan#7**). It avoids faults like the association of a session to the wrong agent (e.g. if an agent name is reused) or wasting resources to maintain the state for the session of a dead agent. Similarly, all the client agents monitor the server(**client_plan#1**). Therefore, they can detect the death of the server (**client_plan#4**) and react accordingly (killing themselves in the sample code).
- b) As describe immediately above, the server monitors all the clients. Hence, if a client is disconnected from the server the server suspends the session (**server_plan#6**) of that client.
- c) The solution to this issue is more complex. The server must recognise a client after they have been disconnected for a certain amount of time. The name of the client is not enough to recognise it, as a restart or revival of this agent

could have occurred while they were disconnected (hence the session for this client shall be terminated, not resumed). The recognition is implemented using a predicate `id/5` composed, in this order, of a timestamp (Hours, Minutes, Seconds), the name of the agent and that of its container (`client_plan#1`). This predicate is created for each spawned client by its initial goal, hence varying for namesake agents. This identifier is sent to the server along with the request of the start of a session (`client_plan#1`). Notice that each session is represented by a belief `state(Client, ClientID, State)` where `Client` represents the name of the client agent, `ClientID` represents the client's identifier and `State` the session's state. Upon reception of each `start_session` request, the server checks whether this identifier corresponds to a client that already started a session (`server_plan#1`), using the test goal `check_id/3` and binding the result to the variable `Result`. If the variable is bound to the value `new_client` (i.e. `server_rule#1` holds), a new session is created (`server_plan#2`). If the variable is bound to the value `restarted_client` (i.e. `server_rule#2` holds because a session for a namesake client existed but the identifiers differ), the session is deleted and a new one is started (`server_plan#3`). Finally, if the variable `Result` is bound to the value `same_client` (i.e. `server_rule#3` holds) the session is resumed(`server_plan#4`).

- d) The clients monitor the server. Thus, they receive a notification when the connection to the server is restored and request the start of the session (`client_plan#5`). As an improvement, the agents also request the start of a session when they detect the restart and revival of the server (`client_plan#3`) as they are aware that their corresponding session has been then terminated.
- e) The supervision mechanism enables the implementation of corrective measures towards the guarantee of the Quality of Service. The code in Figure 6.11 implements an agent `supervisor` that supervises the server (`supervisor_plan#1`). The supervision policy enforced meets the availability and responsiveness requirements . As specified by this policy, the server is pinged every 2 seconds (`Frequency=2000`) with a ping time of 3 seconds (`PingTime=3000`). Two consecutive ping failures (`MaxPings=1`), i.e. 10 seconds of unavailability, trigger a restart of the server (`unblock(never)` and `restart(always)`). Besides, the server is revived every time it dies (`revive(always)`), maximising the availability of the service without replicating it.

The eJason code above can be used as the basis for a wide variety of client-server system architectures. In order to run it, a simple business layer is implemented in Figures 6.13 and 6.12. The state of each session in this sample system is

```
!start.

//#supervisor_plan#1
+!start<-
  .supervise_agents(server,
    supervision_policy([ping(2000,3000,1), unblock(never),
      restart(always), revive(always)]))).
```

Figure 6.11: Code for the supervisor agent

a predicate `counter(Number)` where `Number` is an integer value initialised as 0 that is incremented and decremented by the server as requested by the client agents.

This eJason code has been run by the eJason interpreter using two different agent containers with client agents spread among both of them. The agents (both the server and clients) in the different scenarios tested were manually stopped to successfully assess the correct behaviour of the supervision policy enacted.

6.2.4 Overcoming Race Conditions

This use case shows how some race conditions are avoided by using the supervision mechanism implemented in eJason.

Consider a system composed by the agents `alice` and `bob`. The characteristics of this system are:

- `alice` creates agent `bob`.
- `alice` must maintain `bob` alive by creating it again whenever it dies.

This behaviour is implemented including these plans in the plan base of `alice`:

```
!create_bob <-
  .create_agent(bob,"bob.asl");
  .monitor_agent(bob).

+agent_down(bob)[reason(dead_agent)] <-
  !create_bob.
```

The behaviour implemented is vulnerable to race conditions derived from name clashes. For instance, consider an agent `carol` that executes the internal action

```

// Initialize the counter
+!init_session(Client, InitState) <-
  InitState = counter(0).

// Delete the session state
+!delete_session(Client)<-
  -state(Client, _,_).

// Do nothing
+!suspend_session(Client)<-
  Client = Client.

// Do nothing
+!reactivate_session(Client)<-
  Client = Client.

// Increment the counter
+!process_request(
  counter(Number),
  increment,NewState) <-
  {{NewNumber = Number +1;
   NewState = counter(NewNumber)}}.

// Reset the counter
+!process_request(
  counter(Number),
  reset,NewState) <-
  NewState = counter(0).

```

Figure 6.12: Sample business layer for the server agent

```

// If counter < 10 then increment
+!prepare_request(counter(Number), Request):
  Number < 10 <-
  .sleep(2000);
  Request = increment.

// If counter >= 10 then reset counter
+!prepare_request(counter(Number), Request):
  Number >= 10 <-
  .sleep(2000);
  Request = reset.

```

Figure 6.13: Sample business layer for the client agents

`.create_agent(bob,"agentCode.asl")`. This execution will not interfere with the agent `bob` above as long as it is executed while `bob` is alive (a name clash will occur and `carol` will have to handle it). If it is executed in the time lapse spanning between the death and the creation of `bob`, a name clash occurs and `bob` will not be created. Hence `alice` fails to maintain the agent `bob` alive.

The use of the supervision mechanism instead of the monitoring mechanism can be used to avoid this kind of race conditions

```

!create_bob <-
  .create_agent(bob,"bob.asl");
  .supervise_agents([bob], supervision_policy(
    [revive(always)]))).

```

Now, `alice` is the supervisor of `bob` and will revive it automatically every time that it dies (as specified by the supervision policy `revive(always)`). This revival

is supported by the runtime system and maintains the identity of `bob` after the revival, hence disallowing name clashes with the revived agent. Therefore, the race condition above cannot occur anytime that `bob` is revived.

6.2.5 Overcoming Divergence

This example shows how the supervision mechanism of eJason can be used to overcome the divergence of an agent without killing or restarting it.

Consider an agent `bob` that possesses the following rule in its belief base:

```
even(2):-
    true.

even(X) :-
    X mod 2 == 0.

even(X) :-
    even(Y) &
    X = Y +2.
```

This rule enables the generation of *all* positive even numbers (an infinite set). Consider then a plan that uses this rule in its context:

```
+!plan : even(X) & X < 0 <- .print(X).
```

This plan is not applicable under any circumstances (its context cannot be matched because there is no positive even number lower than zero). Whenever the resolution of the plan context is attempted, an infinite loop is executed. Hence `bob` will be alive but totally unresponsive to stimuli (i.e. it diverges).

This problem may occur during the resolution of the conditions in a plan context, as well as that of test goals, as it often requires the resolution of some rules in the belief base of the agent. As described in Section 2.1.2, the rules used to derive new knowledge. However, their resolution may incur in infinite loops that, in turn, cause the divergence of the agent. Such is the case in the example above. However, the identification of a possible source of an infinite loop is not always as trivial as in this case.

The supervision mechanism can be used to overcome this divergence. For instance, consider an agent **alice** that supervises the agent **bob**. The agent **alice** tests the divergence of **bob** every 10 seconds (**Frequency**), granting **bob** 25 seconds (**PingTime**) to respond to ping signals. If **bob** fails to answer to two consecutive (**MaxPings**) ping signals, **alice** sends an unblock signal to **bob**. The reception of this unblock signal stops the resolution of the plan context, hence breaking the infinite loop. If no other applicable plans for the event being processed were found, the corresponding plan failure event is generated. This supervision policy is implemented using the following internal action (which **alice** eventually executes):

```
.supervise_agents(bob, supervision_policy([ping(10000,25000,1),  
                                           unblock(always)])).
```

This example has been implemented and run in Jason and eJason with the agent **bob** incurring in a divergence in both cases. However, such divergence was overcome in the eJason implementation using the supervision mechanism as described above.

Chapter 7

CONCLUSIONS

The development of distributed and fault-tolerant multi-agent systems that leverage the excellent capabilities of Erlang can be accomplished via the use of eJason. This novel development framework enables the implementation of multi-agent systems spread across several hosting servers with relative ease for the programmer. The governance of the agents that compose such systems is facilitated by different fault-tolerance mechanisms tailored to specific orchestration and availability needs. As shown throughout this thesis document, the realisation of eJason as a (mathematically specified) programming language and interpreter stems from the initial hypothesis that the capabilities of multi-agent systems may be enhanced by borrowing the strengths from already-existing technologies that excel in several aspects relevant to the agent development community. The first of these technologies is the agent-oriented programming language Jason, well-known for the useful abstractions provided by its language constructs, as well as the maturity of its (mathematically specified) program semantics. The second technology is the general-purpose programming language Erlang, based on the actor-model, that provides excellent support to the distribution of programs across several machines while providing mechanisms to ensure their robustness.

The language extensions introduced by eJason strengthen the control of the programmer over the behaviour of each individual agent in a simple and elegant way, as is expected from the declarative programming paradigm. The first of these extensions allows the programmer to define critical sections within the body of plans. These critical sections represent a mechanism to effectively tackle the undesirable effects of data dependencies from the multiplicity of foci of attention inherent to Jason agents. The language constructs that trigger this synchronisation mechanism can be utilised without neither increasing the complexity of the

code nor diminishing its readability. In contrast, the corresponding mechanism provided by Jason requires the division of the functionality across several plans and the implementation of the synchronisation via atomic plans. The second language extension is represented by a slight, yet highly-impacting, change to the interpretation of means-ends reasoning. This change couples the evaluation time of a guard and the execution time of the corresponding guarded code. This change introduces strong guarantees with regards to the behaviour of the agents, which increases the trust from the programmer, hence augmenting the reliability of the eJason code. In order to introduce a similar functionality in Jason, the programmer must first reimplement part of its interpreter (i.e. the intention selection function) using the Java programming language. The third extension modifies the treatment of achievement goal events that are not applicable in order to always retry them at a later moment. The prediction of the applicability of a plan for an achievement goal at implementation time is not a trivial task (or most likely impossible) in a concurrent environment, such as an agent with multiple foci of attention. These goals are vulnerable to race conditions that affect the applicability of their relevant plans. The aforementioned variation relieves the programmer from the task of implementing cumbersome mechanisms in Jason to retry the execution of plans for achievement goals or deal with the corresponding failure events. It may be argued that the intention of the programmer implies the generation of plan failures if an achievement goal in a plan body lacks an applicable plan for it. Should that be the case, it is the position of this thesis work that such goal should not be an achievement goal but a test goal instead. The fourth and last extension tackles the negative effects of race conditions on test goals. This extension enables the programmer to specify whether a failed test goal shall be retried or not, hence diminishing the impact of race conditions on the agent's behaviour. All these language extensions provide the programmer with a better control over the behaviour of each agent, hence providing more guarantees on the outcome of the computations carried out within the whole multi-agent system. These extensions are completely novel and despite their seamless introduction into Jason, are not imported from neither Jason nor Erlang programming languages.

The eJason framework introduces a distribution mechanism that enables the distribution of agents across different hosts without any penalty to their ability of interaction. On the contrary, this agent distribution is transparent to the programmer, hence dropping one major concern, specially in the presence of highly-distributed multi-agent systems. Following the distribution schema implemented, the agents, in order to exchange message, only need to know the name of the addressee, delegating all the routing tasks to the runtime system of eJason. The

distribution mechanism of eJason allows the delegation of the naming of agents to the runtime system, guaranteeing the absence of name clashes. The use of this feature is recommended for all agents excepting those that play a major role in the multi-agent system, e.g. service providers whose name is used as a contract of their services. Another feature of this distribution mechanism is its dynamism. The dynamics of the multi-agent systems allow both the inclusion of new hosts to the multi-agent system (e.g. by merging different isolated multi-agent systems into a new one) as well as the split of a multi-agent system into several isolated ones. The split of a multi-agent system may respond to an explicit request from the eJason code or happen due to connection problems between the hosts in the underlying architecture. In the latter case, it is important to notice that such disconnection does not cause the whole system to crash. Instead, the isolated entities become fully-operational multi-agent systems themselves that may attempt to be merged again after the connection is restored. The distribution schema introduced is inspired by that of Erlang but addresses the specific needs of the eJason multi-agent systems.

The multi-agent systems implemented by eJason may utilise several mechanisms for fault-detection and fault-recovery in order to enhance their robustness. The failures detected range from the degradation of the functionality of an agent, including its death, to the connection (or lack thereof) between agents. These mechanisms, which are designed to address different facets of fault-tolerance, provide different functionality and are thus intended for different necessities. The monitoring mechanism enables an agent (namely monitoring agent) interested in the execution state of another agent (namely monitored agent) to be aware of its execution state, as well as its availability for interaction, by establishing a monitoring relation. When the monitoring agent starts a monitoring relation with the monitored agent, the former is notified by the eJason runtime system about the death, disconnection, restart or even the presence in the system of the latter. This information is specially valuable when the functionality of the monitoring agent somehow depends on that of the monitored agent, e.g. if they must interact frequently. Therefore, the monitoring mechanism can be used to *avoid the propagation of faults* between agents. A monitoring relation does not impose any kind of additional dependency to neither the monitoring nor the monitored agents, hence it can be used discretionally. Even though the monitoring mechanism of eJason is inspired by that of Erlang, it incorporates several improvements. The monitoring relations in eJason persist the death and disconnection of the monitored agent, hence allowing several notifications during its lifetime. The observational power of the monitoring agent is increased as the monitoring relations can now be used to notify the monitoring agent about the reconnection of the monitored agent should

it get disconnected or not exist at all when the relation was established. Moreover, the monitoring agent can now detect the restart or revival of the monitored agent. The supervision mechanism allows the implementation of complex *fault-recovery* behaviours. An agent (namely supervisor agent) that starts a supervision relation with a set of agents (namely supervised set) requests the runtime system of eJason to enforce a series of measures (namely the supervision policy) in order to maintain the availability of the agents in the supervised set up to some extent. The use of the supervision mechanism is recommended as a means to guarantee the availability of an agent or set of agents that carry out tasks relevant in the multi-agent system, e.g. service providers. In order to avoid the impact from hardware disconnections in the supervision mechanism, it is recommended that both the supervisor and its supervised set run in the same agent container. Even though the supervision mechanism of eJason is inspired by that of Erlang, it introduces several improvements. The corrective measures that can be taken are not restricted to the revival of agents that die unexpectedly, but also include now the restart of misbehaving agents. One of the most novel contributions of this mechanism is the possibility to test and unblock the *divergence* of agents, i.e. the undesired and unexpected unresponsiveness of an agent, frequently caused by its internal execution of infinite loops. The mechanisms that test the divergence of an agent ensure that the unresponsiveness of such agent is not caused by a work overload. Another improvement is the decoupling of the supervisor agent and the supervised set. The supervisor agent does not necessarily spawn the agents in its supervised set and does not require any description of their behaviour (in contrast to Erlang's child specifications). Therefore, the supervision trees in eJason are more dynamic than Erlang ones as the supervisor agent may maintain different supervision policies for different disjoint supervised sets. Besides, the agents in the supervised set do not die if their supervisor does, hence deleting a strong dependency between them.

The monitoring and supervision mechanisms, although intended for different purposes, possess several common characteristics. In both cases, a citizen approach is implemented where the detection of faults is specifically enabled by an agent (the monitoring and supervisor agents, respectively) and delegated to the runtime system of eJason. Therefore, the detection of faults only takes place if it produces value to some element of the infrastructure, thus augmenting the efficiency of the system. Another advantage of these mechanisms is that their functionality is provided by the runtime environment of eJason. The programmer just needs to know the constructs of the language that enable their use, delegating the responsibility of their execution to the aforementioned eJason runtime environment. Note that this approach agrees with the declarative spirit of the eJason programming

language. The monitoring and supervision relations persist after the restart or revival of the monitoring or supervisor agents, hence being pervasive in the presence of fault-recovery actions (recall that the restart and revival of an agent only take place as result of the enforcement of a supervision policy). As described before, both mechanisms are inspired by the homonymous Erlang ones but surpass their functionality in order to cater to the needs of the multi-agent systems.

The semantics of the eJason programs have been formalised in order to describe in an unambiguous way the behaviour of both the eJason agents and the multi-agent system. This semantics describes to a programmer exactly which behaviour can be expected with regards to the service level of an implementation. The different eJason mechanisms are described in such a way that they can be extrapolated to other agent technologies. For instance, consider the notification about the death of an agent in eJason. It is represented by the addition of a belief to the belief base of an agent interested in such death, which is specific to Jason. In a different agent architecture, which does not implement belief bases, such notification could still be represented, e.g. by the reception of a message. This separation of concerns permits reflection on how techniques successfully used in general purpose distributed systems (e.g., distribution and fault handling) can be re-interpreted from an agent perspective. The semantics provides then a recipe, and a reference, for other researchers interested in importing such distributed systems mechanism into other agent-oriented programming languages.

The implementation of an actual interpreter for eJason provides the tools to empirically validate the soundness of all the extensions introduced by eJason, i.e. the mechanisms that enhance the programmer's control over Jason agents, the distribution mechanism and the fault-tolerance techniques. The interpreter of eJason is available open-source and free of charge in order to allow anyone interested to access and test its functionality. This implementation of the interpreter includes the main functionality of eJason on top of the functionality of Jason. Therefore, it represents a full re-implementation of the core functionality of Jason as well, which includes a new language parser, the handling of all language constructs and terms, the communication mechanisms and the reasoning cycle, among others. This implementation is not trivial then and, among the contributions of this thesis work, it is the one that demands more effort to be complete.

The contributions of this thesis work, described through this document, show that the initial hypothesis, namely that there exists synergies among strengths of off-the-shelf technologies for the development of agent-oriented programming

languages and those for the development of systems based in the actor model. This synergy is embodied by eJason (both the language extension and its interpreter), along with several new features that exceed the initial expectations, as shown in this document by the different use cases and examples. They represent proof of the potential usage of the framework.

Among the different future lines of work, the most immediate is the implementation and testing of the supervision mechanism when the supervision tree is distributed across several computers. As stated in Section 6.1.2, this feature has been thoroughly tested when all the agents belong to the same container. Developing and executing a suitable testsuite for distributed supervision trees in search for corner cases, remains to be completed.

Another future line of work comprises the study of the integration of eJason agents into the development and execution framework JaCaMo [Boissier et al., 2013, Boissier et al., 2016]. This framework integrates three different dimensions of the development of multi-agent systems. The first dimension corresponds to the agent development capabilities of the programming language Jason. The second dimension corresponds to the environment programming capabilities of the CArtaGo framework [Ricci et al., 2009], which defines agent environments on the basis of the so-called artefacts and workspace abstractions. The third and last dimension corresponds to the management of the organisational model of the agents provided by the framework *MOISE*, which enables the programming of the normative systems that governs the social interaction among the agents.

Finally, the last future line of work identified corresponds to the study and potential implementation of the latest extensions to the Java implementation of Jason. Among these extensions, the concurrent execution of plans and the asynchronous execution of the agent's reasoning cycle are specially relevant subjects of study. These extensions further increase the concurrency within a single agent's behaviour, potentially introducing additional race conditions similar to the ones described in Chapter 3. Tackling these new threats would be the stepping stone of a potential extension of eJason to accommodate the aforementioned concurrent plans and asynchronous reasoning cycle.

Bibliography

- [Agha and Hewitt, 1987] Agha, G. and Hewitt, C. (1987). Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74.
- [Aknine et al., 2004] Aknine, S., Pinson, S., and Shakun, M. F. (2004). An extended multi-agent negotiation protocol. *Autonomous Agents and Multi-Agent Systems*, 8(1):5–45.
- [Armstrong, 2007] Armstrong, J. (2007). Programming Erlang: Software for a concurrent world). The Pragmatic Bookshelf.
- [Bellifemine et al., 2005] Bellifemine, F., Bergenti, F., Caire, G., and Poggi, A. (2005). JADE - a Java agent development framework. In Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer.
- [Bellifemine et al., 2007] Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. Wiley.
- [Boissier et al., 2013] Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6):747–761.
- [Boissier et al., 2016] Boissier, O., Hübner, J. F., and Ricci, A. (2016). The jacamo framework. In *Social coordination frameworks for social technical systems*, pages 125–151. Springer.
- [Bordini and Hübner, 2006] Bordini, R. H. and Hübner, J. F. (2006). Bdi agent programming in agentspeak using jason. In *Computational logic in multi-agent systems*, pages 143–164. Springer.

- [Bordini and Hübner, 2010] Bordini, R. H. and Hübner, J. F. (2010). Semantics for the jason variant of agentspeak (plan failure and some internal actions). In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, pages 635–640.
- [Bordini et al., 2005] Bordini, R. H., Hübner, J. F., and Vieira, R. (2005). Jason and the golden fleece of agent-oriented programming. In *Multi-agent programming*, pages 3–37. Springer.
- [Bordini et al., 2007] Bordini, R. H., Wooldridge, M., and Hübner, J. F. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons.
- [Bratman, 1987] Bratman, M. (1987). Intention, plans, and practical reason.
- [Bratman et al., 1988] Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355.
- [Busetta et al., 1999] Busetta, P., Rönquist, R., Hodgson, A., and Lucas, A. (1999). Jack intelligent agents-components for intelligent agents in java. *AgentLink News Letter*, 2(1):2–5.
- [Butenhof, 1997] Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Professional.
- [Cardoso et al., 2013] Cardoso, R. C., Hübner, J. F., and Bordini, R. H. (2013). Benchmarking communication in actor-and agent-based languages. In *International Workshop on Engineering Multi-Agent Systems*, pages 58–77. Springer.
- [Cesarini and Thompson, 2009] Cesarini, F. and Thompson, S. (2009). *Erlang Programming*. O’Reilly Media.
- [Clocksin and Mellish, 2003] Clocksin, W. and Mellish, C. S. (2003). *Programming in PROLOG*. Springer Science & Business Media.
- [Colmerauer and Roussel, 1996] Colmerauer, A. and Roussel, P. (1996). The birth of prolog. In *History of programming languages—II*, pages 331–367. ACM.
- [Dastani, 2008] Dastani, M. (2008). 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248.
- [Dennett, 1989] Dennett, D. C. (1989). *The intentional stance*. MIT press.

- [D’Inverno et al., 2000] D’Inverno, M., Hindriks, K., and Luck, M. (2000). A formal architecture for the 3apl agent programming language. *ZB 2000: Formal Specification and Development in Z and B*, 1878:168–187.
- [D’Inverno et al., 2004] D’Inverno, M., Luck, M., Georgeff, M., Kinny, D., and Wooldridge, M. (2004). The dmars architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53.
- [Ductor et al., 2011] Ductor, S., Guessoum, Z., and Ziane, M. (2011). Adaptive replication in fault-tolerant multi-agent systems. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011 IEEE/WIC/ACM International Conference on*, volume 2, pages 304 –307.
- [Erlang, 2013] Erlang (2013). Documentation for the supervisor behaviour of erlang. http://www.erlang.org/doc/design_principles/sup_princ.html.
- [Fedoruk and Deters, 2002] Fedoruk, A. and Deters, R. (2002). Improving fault-tolerance by replicating agents. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, AAMAS ’02, pages 737–744, New York, NY, USA. ACM.
- [Fernández-Díaz et al., 2010] Fernández-Díaz, Á., Benac Earle, C., and Fredlund, L.-A. (2010). Implementing a multiagent negotiation protocol in erlang. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang, Baltimore, Maryland, USA, September 30, 2010*, pages 69–74.
- [Finin et al., 1994] Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM.
- [Fipa, 2002] Fipa, A. (2002). Fipa acl message structure specification. *Foundation for Intelligent Physical Agents*, <http://www.fipa.org/specs/fipa00061/SC00061G.html> (30.6. 2004).
- [Georgeff and Lansky, 1986] Georgeff, M. P. and Lansky, A. L. (1986). Procedural knowledge. *Proceedings of the IEEE*, 74(10):1383–1398.
- [Gosling, 2000] Gosling, J. (2000). *The Java language specification*. Addison-Wesley Professional.

- [Haegg, 1997] Haegg, S. (1997). A sentinel approach to fault handling in multi-agent systems. In *Revised Papers from the Second Australian Workshop on Distributed Artificial Intelligence: Multi-Agent Systems: Methodologies and Applications*, pages 181–195, London, UK, UK. Springer-Verlag.
- [Hewitt et al., 1973] Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc.
- [Hindriks et al., 2001] Hindriks, K., de Boer, F., Van Der Hoek, W., and Meyer, J. (2001). Agent programming with declarative goals. *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 248–257.
- [Hindriks et al., 1999] Hindriks, K. V., De Boer, F. S., Van der Hoek, W., and Meyer, J.-J. C. (1999). Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401.
- [Hoare, 1978] Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.
- [Huber, 1999] Huber, M. J. (1999). Jam: A bdi-theoretic mobile agent architecture. In *Proceedings of the Third Annual Conference on Autonomous Agents, AGENTS '99*, pages 236–243, New York, NY, USA. ACM.
- [Ingrand et al., 2007] Ingrand, F., Lacroix, S., Lemai-Chenevier, S., and Py, F. (2007). Decisional autonomy of planetary rovers. *Journal of Field Robotics*, 24(7):559–580.
- [Ingrand et al., 1996] Ingrand, F. F., Chatila, R., Alami, R., and Robert, F. (1996). Prs: A high level supervision and control language for autonomous mobile robots. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 43–49. IEEE.
- [Ingrand et al., 1992] Ingrand, F. F., Georgeff, M. P., and Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE expert*, 7(6):34–44.
- [Jtc, 2011] Jtc, I. (2011). Sc22/wg14. iso/iec 9899: 2011. *Information technology Programming languages C*. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm.
- [Kinny, 1993] Kinny, D. (1993). The distributed multi-agent reasoning system architecture and language specification. *Australian AI Institute, Level*, 6:171.

- [Klein and Dellarocas, 1999] Klein, M. and Dellarocas, C. (1999). Exception handling in agent systems. In *Proceedings of the third annual conference on Autonomous Agents*, AGENTS '99, pages 62–68, New York, NY, USA. ACM.
- [Klein et al., 2003] Klein, M., Rodriguez-Aguilar, J.-A., and Dellarocas, C. (2003). Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7:179–189.
- [Kumar and Cohen, 2000] Kumar, S. and Cohen, P. R. (2000). Towards a fault-tolerant multi-agent system architecture. In *Proceedings of the fourth international conference on Autonomous agents*, AGENTS '00, pages 459–466, New York, NY, USA. ACM.
- [Lee et al., 1994] Lee, J., Huber, M. J., Durfee, E. H., and Kenny, P. G. (1994). Um-prs: An implementation of the procedural reasoning system for multirobot applications. In *NASA CONFERENCE PUBLICATION*, pages 842–842. NASA.
- [Marin et al., 2001] Marin, O., Sens, P., Briot, J. P., and Guessoum, Z. (2001). Towards adaptive fault-tolerance for distributed multi-agent systems. In *Proceedings of the 3rd. European Research Seminar on Advanced Distributed Systems (ERSADS'2001)*, pages 195–201.
- [Morandini et al., 2009] Morandini, M., Penserini, L., and Perini, A. (2009). Operational semantics of goal models in adaptive agents. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '09, pages 129–136, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.
- [Myers, 2001] Myers, K. (2001). Prs-cl: A procedural reasoning system. *Users Guide., SRI International, Center, Menlo Park, CA*.
- [Myers, 1997] Myers, K. L. (1997). User guide for the procedural reasoning system. *SRI International AI Center Technical Report. SRI International, Menlo Park, CA*.
- [Olsen et al., 2008] Olsen, M. M., Siegelmann-Danieli, N., and Siegelmann, H. T. (2008). Robust artificial life via artificial programmed death. *Artif. Intell.*, 172(6-7):884–898.
- [Potiron et al., 2008] Potiron, K., Taillibert, P., and Seghrouchni, A. E. F. (2008). A step towards fault tolerance for multi-agent systems. In *Languages, Methodologies and Development Tools for Multi-Agent Systems*, pages 156–172. Springer.

- [Rao, 1996] Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In Van de Velde, W. and Perram, J. W., editors, *Agents Breaking Away*, volume 1038 of *LNCSS*, pages 42–55. Springer. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAA-MAW'96), Eindhoven, The Netherlands, 22-25 January 1996, Proceedings.
- [Rao and Georgeff, 1995] Rao, A. S. and Georgeff, M. P. (1995). BDI agents: From theory to practice. In *In Proceedings of the first International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319.
- [Ricci et al., 2009] Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009). Environment programming in cartago. In *Multi-Agent Programming*, pages 259–288. Springer.
- [Sardina and Padgham, 2011] Sardina, S. and Padgham, L. (2011). A bdi agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70.
- [Searle, 1969] Searle, J. R. (1969). *Speech acts: An essay in the philosophy of language*, volume 626. Cambridge university press.
- [Stefano and Santoro, 2003] Stefano, A. D. and Santoro, C. (2003). eXAT: an experimental tool for programming multi-agent systems in Erlang. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2003)*.
- [Stefano and Santoro, 2004] Stefano, A. D. and Santoro, C. (2004). Designing collaborative agents with exat. *Enabling Technologies, IEEE International Workshops on*, 0:15–20.
- [Suna, 2005] Suna, A. (2005). *CLAIM et SyMPA : Un environnement pour la programmation d'agents intelligents et mobiles*. PhD thesis, LIP6 - Universit Paris 6 - Pierre et Marie Curie.
- [van Riemsdijk et al., 2005] van Riemsdijk, M. B., Dastani, M., and Meyer, J.-J. C. (2005). Semantics of declarative goals in agent programming. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 133–140, New York, NY, USA. ACM.
- [Vieira et al., 2007] Vieira, R., Moreira, Á. F., Wooldridge, M., and Bordini, R. H. (2007). On the formal semantics of speech-act based communication in an agent-oriented programming language. *J. Artif. Intell. Res.(JAIR)*, 29:221–267.

- [Wooldridge, 2000] Wooldridge, M. (2000). Reasoning about rational agents. MIT Press.