UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Maicon Rafael Zatelli

EXPLOITING PARALLELISM IN THE AGENT PARADIGM

Florianópolis (SC)

2017

Maicon Rafael Zatelli

EXPLOITING PARALLELISM IN THE AGENT PARADIGM

Thesis submitted to the Post-Graduation Program in Automation and Systems Engineering to obtain the Doctor's degree in Automation and Systems Engineering.
Advisor: Prof. Jomi Fred Hübner, Dr.
Co-advisor: Prof. Alessandro Ricci, Dr.

Florianópolis (SC)

2017

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Zatelli, Maicon Rafael
Exploiting Parallelism in the Agent Paradigm / Maicon Rafael Zatelli ; orientador, Jomi Fred Hübner ; coorientador, Alessandro Ricci. - Florianópolis, SC, 2017.
179 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Engenharia de Automação e Sistemas, Florianópolis, 2017.

Inclui referências.

1. Engenharia de Automação e Sistemas. 2. Sistema Multiagente. 3. Arquitetura de Agente. 4. Concorrência e Paralelismo. 5. Arquitetura Multi Core. I. Hübner, Jomi Fred. II. Ricci, Alessandro. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. IV. Título.

Maicon Rafael Zatelli

EXPLOITING PARALLELISM IN THE AGENT PARADIGM

This thesis was judged approved to obtain the Doctor's degree in "Automation and Systems Engineering", and approved in its final form by the Post-Graduation Program in Automation and Systems Engineering of the Federal University of Santa Catarina.

Florianópolis (SC), 31 de march 2017.

Prof. Daniel Ferreira Coutinho, Dr.
Coordinator

Prof. Jomi Fred Hübner, Dr.
Advisor

Prof. Alessandro Ricci, Dr.
Co-advisor

Defense Committee:

Prof. Rafael Heitor Bordini, Dr.

Profa. Jerusa Marchi, Dra.

Prof. Carlos Barros Montez, Dr.

Prof. Rômulo Silva de Oliveira, Dr.

Profa. Ingrid Nunes, Dra.

AGRADECIMENTOS

I would like to thank everyone that contributed directly or indirectly to this research:

- to God, for your immense love and grace;

- to my advisor, prof. Jomi Fred Hübner, for his competence, knowledge, discussions, suggestions, and all efforts in the role of advisor of this work;

- to my co-advisor, prof. Alessandro Ricci, for his competence, knowledge, discussions, suggestions, and all efforts in the role of co-advisor of this work;

- to my family, for its incentive and support as always;

- to CAPES and CNPq, for the financing support;

- to all professors, technicians, and other colleagues of the Department of Automation and Systems Engineering, for the exchanging of knowledge and experience along these years.

The important thing is to not stop questioning. Curiosity has its own reason for existing.

Albert Einstein

RESUMO EXPANDIDO

Muitas aplicações de Sistemas Multiagentes (SMA) requerem que os agentes reajam prontamente às mudanças no ambiente, respondam mensagens rapidamente, e processem outras atividades de alto custo, e todas ao mesmo tempo. O modelo de concorrência adotado no SMA assim como a implementação da plataforma de execução de SMA tem um impacto direto nessas questões. Enquanto a maioria das pesquisas em SMA focam em questões abstratas de alto nível (por exemplo, compromissos), questões de baixo nível, relacionadas ao desenvolvimento de plataformas de execução, ainda precisam de uma investigação mais profunda e avanços. Como consequência, as plataformas atuais de execução faltam com desempenho, escalabilidade ou reatividade em certos cenários onde elas não são capazes de obter propriamente os benefícios da concorrência. Nesta tese, damos um passo em direção a uma plataforma mais flexível para explorar paralelismo em SMA e melhorar o uso dos recursos paralelos de um computador. Analizamos diferentes aspectos que podem ser considerados para melhor tirar vantagem de computadores *multi-core* e *hardwares* paralelos relacionados. A análise resultou em direções para enriquecer plataformas de execução de SMA que melhor suportam concorrência. Um modelo e uma arquitetura concorrentes de SMA e agentes são propostos, onde detalhamos como várias funcionalidades de concorrência inspiradas na análise podem ser combinadas. Para implementar e avaliar a proposta, estendemos uma plataforma de execução de SMA concreta com um conjunto mais rico de funcionalidades concorrentes. A avaliação é feita por meio de experimentos, que consistem no desenvolvimento de aplicações que cobrem cenários chaves para investigar os benefícios e inconvenientes das diferentes configurações para executar o SMA. Os resultados dos experimentos reforçam a importância do desenvolvimento de plataformas de execução de SMA que permitam um desenvolvedor configurar um SMA para melhor explorar concorrência de acordo com os requisitos, demandas e características intrínsecas de cada aplicação.

Palavras-chave: Sistema Multiagente. Arquitetura Multi-Core. Concorrência. Paralelismo. Arquitetura de Agente. Desempenho. Escalabilidade. Reatividade.

ABSTRACT

Many Multi-Agent System (MAS) applications require that agents react promptly to changes in the environment reply messages fast, process other high-cost activities, and all that at the same time. The model of concurrency adopted in the MAS as well as the MAS execution platform implementation can have a direct impact on these issues. While most researches in MAS focus on high level abstraction issues (e.g., commitments), low level issues, related to the development of execution platforms, still need a deeper investigation and advances. As a consequence, current execution platforms lack performance, scalability, or reactivity in certain scenarios where they are not able to properly take benefits from concurrency. In this thesis we make a step towards a platform to exploit the parallelism in MAS and improve the use of the parallel resources of a computer. We analyze different aspects that can be considered to better take advantage of multi-core computers and related parallel hardware. The analysis resulted in directions to enrich MAS execution platforms that better support concurrency. A MAS and agent concurrent model and architecture are proposed, where we detail how several concurrency features inspired on the analysis can be combined. In order to implement and evaluate the proposal, we extended a concrete MAS execution platform with a richer set of concurrency features. The evaluation is performed by means of experiments, which consist in the development of small applications that cover key scenarios to investigate the benefits and drawbacks of different configurations run the MAS. The results of the experiments reinforce the importance of developing MAS execution platforms that allows a developer to configure a MAS to better exploit concurrency according to the requirements, demands, and intrinsic characteristics of each application.

Keywords: Multi-Agent Systems. Multi-Core Architecture. Concurrency. Parallelism. Agent Architecture. Performance. Scalability. Reactivity.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATION AND ACRONYMS

CONTENTS

1 INTRODUCTION

Among the desirable set of characteristics of Multi-Agent System (MAS) applications, we highlight that agents have to react promptly to changes in the environment, reply to messages fast, and process other high-cost activities, all at the same time (LEE et al., 1999). Besides the implementation of the MAS application itself, the choices adopted in the implementation of the underlying MAS execution platform, like the agent architecture and the agent reasoning cycle execution model, can impact on these characteristics.

With the introduction of parallel hardware, such as multi-core architectures, the development of multi-threaded applications started to become even more important as the number of computer cores is increasing. The use of threads better exploits the computational power of this kind of architecture, since a proper implementation of multi-threaded applications allows that different activities can be executed in parallel by different cores (TANENBAUM, 2007; MATTSON; SANDERS; MASSINGILL, 2004). Likewise, a wider number of different strategies to develop MAS execution platforms can be adopted. A proper exploitation of these hardware architectures by the MAS execution platforms can improve the execution of MAS applications (ZATELLI; RICCI; HÜBNER, 2015b, 2015a).

As demonstrated in some experiments and surveys, current execution platforms[1] lack performance, scalability, and reactivity in certain scenarios (CARDOSO et al., 2013; CARDOSO; HÜBNER; BORDINI, 2013; RICCI; SANTI, 2012; BORDINI et al., 2005; MASCARDI; MARTELLI; STERLING, 2004; BORDINI et al., 2006; ALBEROLA et al., 2010; FERNÁNDEZ et al., 2010; BEHRENS et al., 2010; MULET; SUCH; ALBEROLA, 2006; BURBECK; GARPE; NADJM-TEHRANI, 2004; VRBA, 2003). In addition, they do not properly exploit the current parallel hardware available in modern computers. Several strategies have been adopted by execution platforms to run a MAS and, besides the existence of these strategies, a proper analysis of their conceptual and practical aspects was never done in order to investigate if execution platforms are properly exploiting the available hardware or how to better exploit the underlying hardware architecture. We also observed that current execution platforms have individually adopted specific strategies without a proper exploitation of parallelism in certain scenarios. For example, some execution platforms have defined that a different thread will be assigned for each agent,

[1] Hereafter, consider execution platform as MAS execution platform.

and do not have any configurable option to change it (e.g., 2APL). This choice compromises applications that have limitations if an application has hundreds of agents running in a computer with few available cores (MUSCAR, 2011; MUSCAR; BADICA, 2011). Thus, although some BDI-based execution platforms already exploit parallelism for executing multiple agents (using different strategies), they may not improve the execution of individual agents (e.g., 2APL, JACK, Jadex, Jason, JIAC, simpAL).[2]

The agent reasoning cycle, for example, is usually implemented as a sequential execution of steps, harming the reactivity in some scenarios (ZHANG; HUANG, 2008, 2006a; KOSTIADIS; HU, 2000; COSTA; BITTENCOURT, 2000; ZATELLI; RICCI; HÜBNER, 2015b). Experiments performed in (ZATELLI; RICCI; HÜBNER, 2015b, 2015a) also demonstrated that, according to the concurrency *configuration* used to run a MAS, we can obtain different results in terms of performance, reactivity, and scalability. By *configuration*, we mean the parameterization adopted to run a MAS (e.g., the number of threads used in the MAS and the strategy to assign threads to agents). There is not an ideal *configuration* for all scenarios.

Several factors should be considered to decide how to execute a MAS, such as the characteristics of the applications and their requirements, as well as the underlying hardware architecture where the application will be executed. In addition, an execution platform aiming to execute applications developed using a MAS language (e.g., Jason, GOAL, 2APL) should not impose limitations in certain application scenarios, but the platform should be conceived independently of the application that will be executed. Thus, it is reasonable to provide options to configure how an application should be executed according to its needs. The main challenge to support this flexibility is to conceive an execution model and architecture that, through the use of concurrency techniques, support different configurations to execute applications, and, at the same time, keeps the same semantics independent of the configuration. This is one of the main points that must guide an analysis of the execution of BDI agents, and how we can exploit concurrency in its different elements as well as manage the execution of agents. Addressing this issue is not only a matter of implementation, but it requires a careful analysis focused on the understanding of the conceptual level of execution of BDI agents until how to concretely execute them on an underlying hardware architecture.

[2] All the references related to the agent languages are presented in Appendix A.

1.1 SCOPE

This thesis focus on agents following the BDI model, which is highly adopted in current agent languages and theories.[3] We assume an agent model where the execution of the agent and the external world (i.e., the environment) are independent. External actions performed by an agent (e.g., an action in the environment) are carried on by the environment (with its own threads and independent execution) and do not interfere with the agent execution. Thus, the execution of agents can be considered CPU bound and its reasoning cycle can be executed continuously, without getting blocked due to I/O operations (e.g., by waiting the result of the execution of an external action). Finally, the research is focused on exploiting the concurrency in a single computer.

1.2 RESEARCH QUESTION

The research question that we want to answer with this thesis is "Which concurrency techniques and models can be exploited to develop MAS execution platforms to better take advantage of multi-core architectures and related parallel hardware?".

1.3 OBJECTIVES

The overall objective of this thesis is to investigate concurrency techniques and models to properly design and deploy MAS execution platforms that better take advantage of parallelism provided by multi-core architectures and related parallel hardware. The specific objectives of this thesis are:

- To identify features of a MAS and BDI agent to better exploit parallelism;
- To conceive a concurrent BDI agent and MAS *model* and *architecture*, including agent reasoning cycle, that gather the main BDI concepts and elements to better exploit parallelism;
- To integrate the proposed agent architecture and other identified

[3] Although the focus is on the BDI model, inspiration can be also considered from other agent models as well as the thesis contributions can be used to extend other agent models and architectures.

concurrency features and mechanisms in an MAS execution platform;

- To characterize the MAS execution and propose metrics to evaluate the execution of applications.

1.4 CONTRIBUTIONS AND RELEVANCE

According to the review of the state-of-the-art and the scope of this thesis, we found only few works related to how to exploit parallelism in the development of execution platforms as well as to address the limitations of current platforms regarding to the execution of applications in some scenarios. Thus, this thesis brings some practical and theoretical contributions.

The theoretical contribution is the analysis of different aspects of a MAS that could be relevant to improve execution platforms to better exploit parallelism to run applications. The analysis is done in the MAS, agent, and intention levels of concurrency, which means that not only external aspects of agents are considered, but also how to exploit parallelism to improve the execution of each individual agent. Some of these aspects include different ways to launch intentions, to perform the reasoning cycle, and to distribute threads among agents and the internal components of an agent. Other theoretical contributions are the conception of a concurrent model and architecture for BDI agents and MAS.

Our practical contribution is Jason(P), the extension of the Jason execution platform considering the integration of the different concurrency features identified in the previous analysis. Jason(P) allows a developer to evaluate different concurrency configurations to run an application, and chooses the most suitable configuration according to the MAS application and its particular requirements and characteristics. Thus, another practical contribution of the thesis is the characterization of the MAS execution (e.g., based on the communication load), which allows us to identify which are the relevant aspects that can influence on the choices for the configuration. Furthermore, the extensions are not strictly limited to the specific execution platform extended in this thesis, but it can be used as inspiration to extend other execution platforms.

On the one hand, this thesis is relevant in the MAS field bringing theoretical and practical contributions and advances regarding the exploitation of parallelism. On the other hand, this thesis is also rele-

vant in the field of engineering of automation, where robots and other machines used in industry can have since limited resources until very powerful multi-core hardware. The different characteristics provided by these robots and machines force the MAS developer to use certain configurations in order to better exploit their hardware.

1.5 DOCUMENT OUTLINE

The rest of this document is structured as follows. Chapter 2 presents some general concepts related to agents, MAS, and concurrency. Chapter 3 presents the result of our analysis of different aspects of execution platforms that, considering some concurrent techniques, can be useful to better take advantage of multi-core computers and related parallel hardware. We analyze conceptual and practical aspects that can be considered when developing execution platforms. The analysis considers the MAS, agent, and intention levels. The MAS level is related to how the execution platform can manage the execution of several agents, such as how threads can be distributed among the several agents of a MAS. The agent level is related to how the platform manages the execution of the internal elements of each agent individually, such as its reasoning cycle. The intention level is related to how the platform manages the execution of the agent intentions. Other details about the state-of-the-art are presented in Appendix B.

Chapter 4 presents our proposal of a model and architecture that aims to exploit concurrency by improving the use of the CPU and its cores while minimizing the overheads caused by the adoption of certain concurrency mechanisms. A set of algorithms to execute BDI agents is also conceived to specify how some features and the reasoning cycle can be actually implemented considering the model and architecture. In addition, we apply different techniques to schedule the execution of agents and to assign threads among them. Appendix C presents the integration of our BDI agent and MAS architecture in a concrete execution platform. We take Jason as a reference for BDI agents and propose Jason(P), which extends the Jason agent architecture, reasoning cycle, and execution platform to enable MAS developers to configure an application, considering the features and parameters provided by our proposal.

The evaluation of the thesis is performed in Chapter 5 and aims to experimentally identify the benefits or drawbacks of some features that can be configured to run an application. We propose metrics and iden-

tify application characteristics that are useful to evaluate the execution of applications. We also identify some key scenarios and applications that have different demands, thus requiring different configurations to improve its execution.

The results of the thesis and a discussion about the choices made during the development is presented in Chapter 6. Finally, Chapter 7 presents our final considerations, resulted publications, and future work.

2 THEORETICAL UNDERPINNINGS

This chapter briefly presents theoretical underpinnings that are useful to understand the present thesis. Sec. 2.1 presents some concepts related to agents and MAS. Sec. 2.2 presents concepts and challenges related to techniques to exploit concurrency.

2.1 MULTI-AGENT SYSTEMS

This section presents some concepts related to MAS as adopted in this thesis. While Sec. 2.1.1 introduces the basic concepts related to agents, Sec. 2.1.2 presents a practical implementation of agents in a concrete platform.

2.1.1 Agents

An agent is a computational system settled in an environment and it is able to act *autonomously* in this environment in order to accomplish its project aims. The agent gets data from the environment by means of sensors and produces, as output, actions that affect the environment. An agent differs from the models based on objects because it has certain properties that do not satisfy the conditions for an agent to belong to the class of simple objects (WOOLDRIDGE, 2002). According to Wooldridge (1995), an intelligent agent must have the following properties:

- autonomy: the autonomy is the capability for an agent to act in an independent way in order to achieve the goals that were delegated to it. Thus, at least, an autonomous agent makes independent decisions about how to accomplish the aims that were imposed to it. In addition, the agent has control over its own execution, usually by means of its own thread;

- proactivity: the agent is able to exhibit behavior towards the goal. If a goal was delegated to an agent, the agent will try to make decisions focusing on achieving the goal;

- reactivity: being reactive means being sensible to changes in the environment, that is, it is the capability that the agent has to answer to stimulus from the environment in a reflexive way;

- social ability: it is the capability that the agents have to cooperate and coordinate the activities with other agents in order to accomplish the aims of the system. The social ability is not simple byte exchanges between the agents, but also a knowledge exchange, that is, the agents are able to communicate between them their believes, goals, and plans.

Considering the several existing agent models, this thesis adopts the *Belief-Desire-Intention* (BDI) model (BRATMAN; ISRAEL; POLLACK, 1988), which is inspired on the theory of human practical reasoning of Bratman (BRATMAN, 1987). The central idea of the BDI model is that the behavior of a situated computer system (called *agent*) is determined by a sort of mental state. The BDI model uses three attitudes to determine this mental state: beliefs, desires, and intentions. *Beliefs* are the information that agents have about the world and are stored in a structure called *belief base*. Beliefs can be inaccurate and incomplete. *Desires* and *Intentions* are two notions related to goals. A desire is a goal that the agent is not committed to bring about. To have a desire does not imply that an agent is working to achieve it. A desire can be thought of as the representation of the *motivational* state of the agent. An *intention* is a goal that an agent decided to bring about.

In this thesis, we consider two more concepts that can be added to the BDI model to practically implement it: *events* and *plans*, which are already used in BDI agent languages, such as Jason. An *event* happens while an agent is in execution and refers to belief updates, goal adoptions, failures, changes in the environment, etc. An agent program is based on *plans* that compose the agent *plan library*. A plan is essentially a mean to handle some *event*. A plan is composed of a unique *name*, a *trigger*, a *context*, and a *body*. The *trigger* is the event that the plan can handle (e.g., the adoption of certain goal). The set of plans with trigger e is called the *relevant* plans for event e. The *context* is used to specify the conditions for the application of the plan, it is a logical formula that is evaluated according to the agent beliefs. The relevant plans that have their context condition satisfied are called *applicable* plans, and they can be actually chosen by the agent to handle the event. The *body* is a sequence of *deeds*[1] that, if successfully executed, will be considered as the event being properly handled. These deeds can refer to the adoption of a new goal, to the update of a belief, to the sending of a message to another agent, to the execution an action in the environment, etc.

[1]The term *deed* is used in the same way as in (DENNIS et al., 2012) and it refers to the kinds of formula that appear in a plan body.

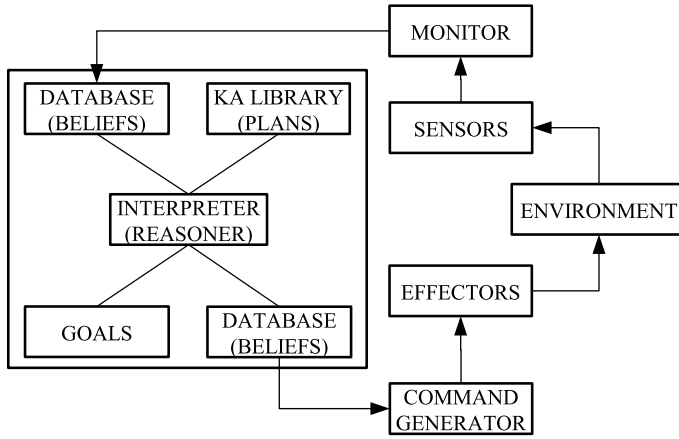

Figure 1 – Procedural Reasoning Cycle (GEORGEFF; INGRAND, 1989).

One of the best-known agent architectures is the Procedural Reasoning System (PRS), developed by Georgeff and Lansky (GEORGEFF; LANSKY, 1987) and based on the BDI model. PRS consists of a set of elements (Fig. 1). The *database* contains current *beliefs* (or facts) about the world and the internal state of the system. The *goals* contains the set of goals to be achieved by the system. The *KA Library* contains the *plans* (also called *Knowledge Areas (KAs)*) that are used to achieve the goals or react to particular situations. Finally, the *intention structure* contains plans that have been chosen for an eventual execution. The system interacts with its environment, including other systems, through its database (which is updated in response to changes in the environment) and through the actions that it performs according to the intentions that are being executed (GEORGEFF; INGRAND, 1989; INGRAND; GEORGEFF; RAO, 1992; GEORGEFF; LANSKY, 1987).

An *interpreter* (or inference mechanism) manipulates these components, by cyclically selecting appropriate plans based on the beliefs and goals, placing the selected ones on the intention structure, and executing them. At any particular time, certain goals are active in the system and certain beliefs are held in the system database. Given these goals and beliefs, a subset of plans in the system will be applicable (i.e., will be invoked). One or more of these applicable plans will be chosen for execution and will be placed on the intention structure. An intention of the intention structure is selected for a further execution. Since the execution of any course of action to completion increases the risk that a significant change will occur during this execution, and the

agent could fail to achieve the intended objective, only one deed of the intention is executed (RAO; GEORGE, 1995). A deed of an intention can be either a primitive action or one or more unelaborated sub-goals. If the former, the action is directly initiated; if the latter, these sub-goals are posted as new goals of the system. The execution of primitive actions can affect not only the external world but also the internal state of the system. For example, a primitive action can operate directly on the beliefs, goals, or intentions of the system. Alternatively, the action may indirectly affect the state of the system as a result of the knowledge gained by its interaction with the external world. At this point, the cycle begins again: the new goals and beliefs trigger new plans, one or more of these are selected and placed on the intention structure, and finally an intention is selected from the intention structure and partially executed (GEORGEFF; INGRAND, 1989; INGRAND; GEORGEFF; RAO, 1992; GEORGEFF; LANSKY, 1987).

As an extension for the BDI model proposed in (BRATMAN, 1987), in (COHEN; LEVESQUE, 1987, 1990), an intention is seen as composed of two more basic concepts, *choice* (or goal) and *commitment*. The notion of commitment is useful for BDI agents to commit with previous decisions. A commitment balances reactivity and goal-directedness of an agent. In a continuously changing environment, commitment lends a certain sense of stability to the agent reasoning process. A commitment usually has two parts: one is the condition that the agent is committed to maintain, called the commitment condition, and the second is the condition under which the agent gives up the commitment, called the termination condition. Thus, an agent can commit to an intention based on the object of the intention being fulfilled in one future path (RAO; GEORGE, 1995).

Some interesting properties of intentions can be defined with the use of commitments. The agent should adopt intentions that it believes are feasible and forego those believed to be unfeasible; keep (or commit to) intentions, but not forever; discharge those intentions believed to have been satisfied; alter intentions when relevant beliefs change; and adopt subsidiary intentions during plan formation. Adopting an intention has many effects on the agent mental state. Thus, intentions affect the beliefs, commitments to future actions, and other interdependent intentions. Intentions can also be seen as a kind of *persistent goal*. An agent has a persistent goal if it has a goal that will be kept at least as long as certain conditions hold. If either of those circumstances fail, the agent drops its commitment to achieving the goal. Persistence involves an agent internal commitment to a course of events over time (COHEN;

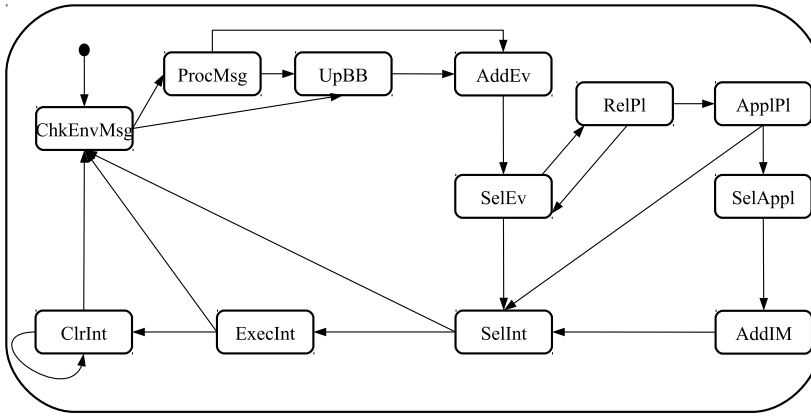

Figure 2 – BDI agent reasoning cycle in Jason.

LEVESQUE, 1987, 1990).

2.1.2 A Practical Implementation of BDI

In order to present a practical implementation of the BDI agent reasoning cycle, we illustrate the implementation done in the Jason platform, which is conceived based on the PRS. The reasoning cycle can be conceptually divided in three main stages: the *sense*, *deliberate* (or *think*), and *act* (Fig. 11).

In the sense stage, the agent senses the environment and receives messages from other agents (`ChkEnvMsg`). As the result of the perception of the environment, the agent gets all percepts that represent the state of the environment. Each percept represents a particular property of the current environment state (e.g., the temperature) and the belief base is updated in order to reflect that state (`UpBB`). Messages received by the agent are processed according to their performatives, which can imply the addition or removal of plans, goals, or beliefs (`ProcMsg`). In both cases (changes in the belief base or messages received by the agent), events are produced and added in a queue (`AddEv`).

In the deliberation stage, the agent handles such events. Only one pending event is selected to be handled in each reasoning cycle (`SelEv`). The next step is to retrieve all plans from the plan library that are relevant for handling the event (`RelPl`). For example, let the plans in Fig. 3 be the current plan library of the agent, where the basic syntax

```
@p1 +pressure(P): P > 1024 <- ...
@p2 +temperature(T): T < 10 <- ...
@p3 +temperature(T): T >= 10 & T <= 20 <- ...
@p4 +temperature(T): T > 20 <- ...
@p5 +temperature(T): T > 30 <- ...
```

Figure 3 – Example of plans in the plan library.

of a plan is given by `@name trigger: context <- body`. If a belief `temperature(32)` is just included in the agent belief base, an event `+temperature(32)` is produced. When the event `+temperature(32)` happens, only plans `@p2`, `@p3`, `@p4`, and `@p5` can be initially considered to handle it (i.e., they are relevant).

From the relevant plans, those that are applicable in the current context (`ApplPl`) are selected, that is, those that can be used to handle the event given the agent's current beliefs. Continuing with the example of the event `+temperature(32)`, only two plans are applicable (`@p4` and `@p5`). They are applicable because after the unification, the variable `T` is bound to `32`, which is higher than `20` (`@p4`) and `30` (`@p5`).

The agent can still have several applicable plans and any of them could be chosen to (hopefully) handle the event successfully. The next step is to select one applicable plan to commit to (`SelAppl`). Currently, Jason opts for the first applicable plan. After selecting the plan, the agent finally has the *intention* of pursuing the course of deeds determined by that plan (`AddIM`).

Several intentions can be active at the same time and compete for the agent attention. The active intentions are placed in a queue and in each reasoning cycle, only one intention is selected to be executed (`SelInt`). By default, an intention is selected using a round-robin scheduling mechanism, which means that each turn *one* different intention is selected and *only one* of its deeds is executed (`ExecInt`). The reason for executing only *one* deed of *one* intention, besides allowing multiple intentions to carry on concurrently, is that the execution of any course of deeds to completion increases the risk that a significant change may occur during this execution and the agent could fail to achieve the intended objective (RAO; GEORGE, 1995).

An intention in Jason is a stack of instantiated plans (i.e., *intended means*), where the plan on the top of the stack is the one that is currently ready to be executed. What an intention will actually execute depends on the type of deed at the beginning of the body of the plan that is on the top of the stack. If the deed causes the instantiation

of a new plan, this plan will be placed on the top of the stack of the intention. The instantiation of a new plan can happen, for example, when the deed at the beginning of the plan body causes the adoption of a goal and there is an applicable plan for achieving that goal. The remaining plans, below the top of the stack, will continue their executions only when the new instantiated plan terminates successfully. In this case, an analogy between the execution of an intention and the execution of a method in object-oriented languages can be made. If a method (e.g., α) calls another method (e.g., β), the method (α) continues its execution only after the called method (β) finishes its execution. If the execution of a deed fails, the intention also fails. Thus, if the last deed of the plan executes successfully, the intention is considered terminated with success. An intention can be active or suspended. The suspended intentions are not considered by the round-robin scheduling mechanism. Finally, the last step of the reasoning cycle is to remove the empty intentions (finished ones) (`ClrInt`).

Finally, along the years, several agent languages inspired by the BDI model were proposed. Some of the first agent languages proposed are AGENT0, METATEM, AgentSpeak(L), APRIL, INTERRAP, Agent Factory, Ciao Prolog, JIAC Agent Framework, 3APL, Jinni, IndiGolog, ConGolog. More recent languages and frameworks include Jason, GOAL, 2APL, IMPACT, Jadex, ASTRA, JACK, ALOO, CLAIM, SARL, MINERVA, etc. A more detailed survey about agent languages can be found in (SADRI; TONI, 1999; BORDINI et al., 2006; MASCARDI; MARTELLI; STERLING, 2004; BORDINI et al., 2009, 2005).

2.2 TECHNIQUES TO EXPLOIT CONCURRENCY

While a sequential program has a single line of execution (thread or process), a concurrent program has multiple lines of execution. One reason for parallelization is that multiple activities can be executed concurrently and exploit the different cores of a computer. Different from a parallel execution, that literally executes different activities at the same time (e.g., the number of activities is equal or lower than the number of cores of a computer), in a concurrent execution, the activities do not necessarily execute at the same time, that is, the activities compete for the use of the computer cores (e.g., several activities executing on a single-core computer). Furthermore, in both cases, the activities can be blocked from time to time (e.g., waiting some event), thus some applications are better modeled decomposing the several activities in multiple

lines of execution, which allows some activities to continue executing while others are blocked (TANENBAUM, 2007).

When working with concurrent programming it is also necessary to deal with some issues in order to ensure that the execution of the whole system is correct (RAYNAL, 2013). Andrews (1999) makes an analogy between concurrent programming and car traveling. Suppose that several cars want to go from point `A` to point `B`. If they are competing on the same road then they can follow each other or compete for positions, however the competition for positions can bring some accidents. They could also drive in parallel lanes, thus they will not compete and be able to arrive at point `B` at the same time. Alternately, they could also choose to use different routes, using different roads. The tasks that must be executed in a system are represented by the cars moving. Each task can be executed at each time in a single processor (roads) or they can be executed in parallel on multiple processors (lanes in a road). They could also be executed on distributed processors (separate roads). As on a road, the tasks also need to synchronize to avoid accidents, stop at traffic lights, and respect the signs (ANDREWS, 1999). Thus, some of issues related to concurrent programming are deadlocks (CAMPBELL, 2011), livelocks (GOUDA; CHOW; LAM, 1984), starvation (TANENBAUM, 2007), and race conditions (PRAUN, 2011). Several mechanisms have been proposed to deal with these issues. The most common and classic mechanisms are semaphores (SCOTT, 2011; ANDREWS, 1999), mutexes (TANENBAUM, 2007), monitors (SCOTT, 2011; TANENBAUM, 2007; ANDREWS, 1999), message passing (SCOTT, 2011; TANENBAUM, 2007; ANDREWS, 1999), barriers (TANENBAUM, 2007; SCOTT, 2011), and transactional memory (HERLIHY; MOSS, 1993; HERLIHY; SHAVIT, 2008; HERLIHY, 2011). More details about concurrent programming and multiprocessors architectures and operational systems can be found in (HERLIHY; SHAVIT, 2008; ANDREWS, 1999; TANENBAUM, 2007; PADUA, 2011; RAYNAL, 2013).

In this section, we present some concepts and techniques to exploit concurrency. First of all, we present the basic terminology adopted in the remaining of this section (Sec. 2.2.1). Sec. 2.2.2 briefly introduces task decomposition. Sec. 2.2.3 presents scheduling techniques to execute the tasks. Finally, we explore thread pools, which is currently a quite adopted technique to execute tasks (Sec. 2.2.4).

2.2.1 Basic Terminology

Each paragraph of this section briefly introduces a term that will be adopted.

Task. A task is then a sequence of instructions that operate together as a group. This group corresponds to some logical part of an algorithm or program (MATTSON; SANDERS; MASSINGILL, 2004).

Unit of Execution (UE). A task need to be mapped to either a process or a thread to be executed. Threads run in the same address space in a quasi-parallel way. The advantage about using threads is that they share all data among themselves, while processes must communicate by means of messages because they run in separated memory spaces (TANENBAUM, 2007; MATTSON; SANDERS; MASSINGILL, 2004). In this section, we refer to UE as either a process or a thread.

Processing Element (PE). A stream of instructions is executed by a hardware element, which can be a workstation in a cluster of workstations, or each individual processor in a single workstation, depending on the context (MATTSON; SANDERS; MASSINGILL, 2004).

Load balance. Tasks must be mapped to UEs, and UEs to PEs. The way in which this mapping is done can have a significant impact on the performance. It is desirable to avoid that some PEs are doing most of work while others could be idle. Thus, load balance refers to how well this mapping is done. Load balancing can be done statically or dynamically (MATTSON; SANDERS; MASSINGILL, 2004).

2.2.2 Task Decomposition

The first step to design a parallel program is to break the problem in tasks. Task decomposition is a technique that aims to decompose an algorithm or program in several sub-tasks that can be carried on concurrently. The tasks originated of the decomposition should meet two main criteria: (1) the number of tasks should be equal or greater than the number of UEs; (2) the computation associated with each task must be large enough to offset the overhead associated with managing the tasks and handling any dependencies (MATTSON; SANDERS; MASSINGILL, 2004).

Tasks in a program can be found in different places. For example, they can be a call to a function or each iteration in a loop within an algorithm. In an imaging processing algorithm where each pixel is updated independent, the task definition can be individual pixels, im-

age lines, or even whole blocks in the image. On a system with a small number of nodes connected by a slow network, tasks should be large enough to offset the high communication latencies, so blocks of images could be more appropriate in this case. The same problem could be found in a system that contains a large number of nodes connected by a fast (low-latency) network. In this case, smaller tasks are desirable in order to keep all the PEs occupied (MATTSON; SANDERS; MASSINGILL, 2004).

Dependencies among the tasks can have a major impact on the task decomposition decisions and on the emerging algorithm design. They can be classified in two categories: *ordering* and *shared data*. Ordering constraints are those task groups that must be handled respecting a required order. Thus, some tasks must be computed before other tasks start. Shared-data dependencies refers to the situation where data is shared among different tasks. It is ideal that the dependencies among tasks are eliminated or at least minimized (MATTSON; SANDERS; MASSINGILL, 2004).

2.2.3 Scheduling

The tasks are assigned to UEs by means of a scheduling strategy that must consider load balance as a main issue. Two classes of scheduling strategies are used in parallel algorithms: static schedules and dynamic schedules (MATTSON; SANDERS; MASSINGILL, 2004).

In static schedules, the tasks are distributed to the threads at the start of the computation and does not change. This is more often used when computational resources available are predictable and stable over the course of the computation (e.g., the computing system is homogeneous). Thus, if the set of times required to complete each task is narrowly distributed about a mean, the sizes of the blocks should be proportional to the performance of the UEs (e.g., in a homogeneous system, they are all the same size). When the effort associated with the tasks varies considerably, the number of blocks assigned to UEs must be much greater than the number of UEs. Static schedules incur the least overhead during the parallel computation and should be used whenever possible (MATTSON; SANDERS; MASSINGILL, 2004).

In dynamic schedules, the distribution of the tasks among the UEs varies as the computation proceeds. This kind of schedule is used when (1) the effort associated with each task varies widely and is unpredictable or (2) when the capabilities of the UEs vary widely and

unpredictably. The most common approach used for dynamic load balancing is to define a task queue to be used by all UEs. When an UE completes its current task, it removes another task from the task queue. Faster UEs or those receiving lighter-weight tasks will access the queue more often an thereby be assigned more tasks (MATTSON; SANDERS; MASSINGILL, 2004).

Multiple queues can be used in order to minimize contention. Thus, instead of threads compete to access the same queue, the number of threads per queue is reduced in order to also reduce the thread competition. In this line, work stealing is a dynamic schedule technique that aims to distribute the works about the threads more fair. In a multiple queue approach some queues can have more works than others, or even some queue could be empty. In work stealing, tasks are distributed among the UEs at the start of the computation. Each UE has its own work queue. When the queue is empty, its UE becomes a *thief* and tries to steal work from the queue on some *victim* UE (where the *victim* UE is usually randomly selected). In many cases, it produces an optimal dynamic schedule without incurring the overhead of maintaining a single global queue. Work stealing does not make any guarantee of the order in which tasks are selected to be executed (MATTSON; SANDERS; MASSINGILL, 2004; BLUMOFE; LEISERSON, 1999).

2.2.4 Thread Pools

Multi-threading architectures range from *thread-per-request* to *thread-pool* architectures. In a thread-per-request architecture, threads are created according to the requests that are arriving (e.g., to perform a job), and are destroyed after finishing the request. In a thread pool, a certain number of threads is created and maintained in a pool of threads. Thus, a thread pool is a collection of *worker* threads that are available for computational *jobs* and that can be recycled. When a request arrives, a free thread in the pool is used to handle that request, returning to the pool after finishing the request. Thread pools are specially useful in server applications where jobs are typically short-lived and the number of incoming work is large. It is common to adopt a thread pool pattern for designing scalable multi-threaded and distributed systems. The thread pool pattern uses pools of pre-existing and reusable threads to limit thread life cycle overhead (thread creation and destruction) and resource trashing (thread proliferation) (LING; MULLEN; LIN, 2000; SCHMIDT, 1998; SYER; ADAMS; HASSAN, 2011). Figure 4 presents an

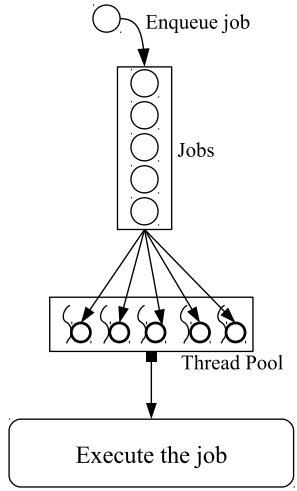

Figure 4 – Thread pool.

example of a thread pool composed of 5 threads to execute the jobs from a queue. The circles `O` are the jobs, while the strings in format of `S` are the threads, and `So` mean that a thread is executing a job. The rectangle with circles inside mean the queue of jobs from where threads select the jobs. The arrow departing from that rectangle mean the action of a thread selecting a job from the queue. The arrow with a square in an end and a triangle in another end means what a thread is doing (i.e., executing the job).

If a thread pool is too big, threads compete for scarce CPU and memory resources, resulting in higher memory usage and possibly resource exhaustion. If it is too small, throughput suffers as processors go unused despite available work. Sizing properly a thread pool means to understand the computer environment, the resource budget, and the nature of the tasks. Questions like how many processors does the system have, how much memory, if tasks perform mostly computation, I/O, or some combination, must be answered. Thus, the number of threads in the pool depends on the number of available cores and the nature of the jobs in the queue. In the case of CPU-bound jobs, the maximum utilization of the CPU can be reached with the number of threads equals to the number of cores. It is also advisable to set one thread more in case of threads occasionally take a page fault or pause for some other reason, so an “extra” runnable thread prevents CPU cycles from going

unused when it happens. For jobs that may wait for I/O operations to complete (e.g., read an HTTP request from a socket), the number of threads can be higher than the available cores because not all threads will be working at the same time. In this case, the number of threads in the pool can be defined according to the formula $n * (1 + wt/st)$, where n is the number of computer cores, wt is the waiting time, and st is the service (or computation) time (GOETZ, 2002; GOETZ et al., 2006).

3 AN ANALYSIS OF CONCURRENCY IN MAS EXECUTION

In this chapter, we analyze different aspects of a MAS that can be considered when developing execution platforms to better take advantage of multi-core computers and related parallel hardware. When designing concurrent systems, the first thing to do is to explicitly identify the independent parts of the system that can thus be executed by different UE in a way that improves the CPU usage. In a MAS execution, agents are independent parts, however the agent execution itself has internal dependencies. We can identify ordering constraints and shared-data dependencies. In ordering constraints, the order in which things happen is strict, and if the order is not respected, the execution can reach an inconsistent state. The main ordering constraint is the execution of the reasoning cycle, where certain steps cannot start before others finish. In data-shared dependencies, some data can be used by different parts of the agent and if these parts are executed concurrently, a control access mechanism needs to be conceived to read and write that data, otherwise, the execution can also reach an inconsistent state. In a MAS execution, shared-data dependencies can be seen in the data structures used by an agent, such as its belief base, which can be accessed in several moments of the execution of its reasoning cycle.

We proceed with an analysis and discussion about how to exploit parallelism in the context of MAS, starting from a MAS level perspective (Sec. 3.1) and then going to an agent level perspective (Sec. 3.2) and finally reaching an intention level perspective (Sec. 3.3). The MAS level concurrency is related to how the execution platform manages the execution of several agents, such as how UEs are distributed among the several agents of a MAS. The agent level concurrency is related to how the execution platform manages the execution of the internal elements of each agent individually, such as its reasoning cycle. The intention level concurrency is related to how the execution platform manages the execution of intentions. We only present the most important concepts considering the scope of this thesis. Details about the related work can be found in Appendix B or their respective references. Finally, we present an analysis of the state-of-the-art and situate this thesis in it (Sec. 3.4). At the end of this chapter, we present some conclusions (Sec. 3.5).

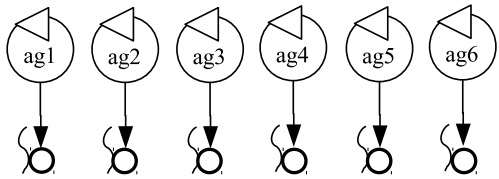

Figure 5 – Each agent owns a UE.

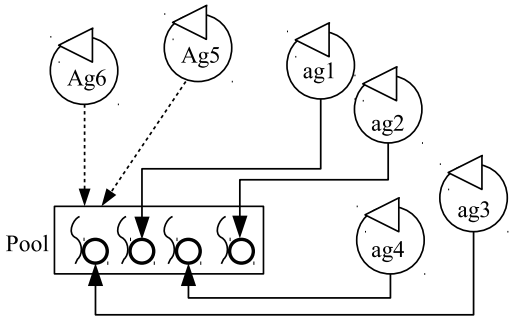

Figure 6 – Agents are executed by the UEs shared in a pool.

3.1 MAS LEVEL CONCURRENCY

At a MAS level, agents are autonomous entities encapsulating an internal logical flow and keeping control over its internal structures and reasoning cycle. As the execution of an agent is independent from other agents and from the external world, agents can be executed concurrently. The mapping of this logical level onto a physical level can be reified in different forms. At an extreme side, a MAS can be executed using a single UE, where all agents are executed sequentially, one after the other. At the other extreme side, each individual agent can be executed using its own UE(s), which means that if the MAS is composed of six agents, agents will be executed by six or more UEs (Fig. 5 illustrates an example, where a circle is a job that a UE executes, which, in this case, mean the execution of an agent). This is a typical choice in many platforms, such as JADE, Jadex, Jason, 3APL, 2APL, METATEM, and JIAC Agent Framework. While at the first extreme, a parallel hardware is clearly not being exploited, at the other extreme, serious drawbacks can be identified in terms of scalability and performance as soon as the number of agents increases (i.e., substantially greater than the number of available PEs).

A more efficient choice is to introduce a pool, whose job is to carry on the execution of all agents in the MAS. Agents that are ready to be executed can be enqueued and UEs select agents from the queue to execute one reasoning cycle (like adopted in Jason, simpA, simpAL, ALOO, Agent Factory Framework, JACK, GOAL, SARL, and ASTRA). For example, 10,000 agents could be created, however all agents will be executed by the same four UEs available in a pool. Fig. 6 presents an example where the six agents of the previous scenario are now executed by four UEs. The arrows connecting to the jobs of the UEs in a pool — illustrated as So — mean that the agent is currently being executed by the corresponding UE, while the arrows connecting to the border of the pool (not directly to the circles) mean that the agents are waiting to be selected by a UE from the pool. While four agents are being actually executed, the other two are waiting. The number of UEs is typically chosen considering the number of PEs and the kind of job that UEs execute. However, the number of UEs can be changed dynamically, supporting ways of self-adaptation, which is useful to improve performance. Thus, the execution platform could use three UEs for executing an agent and five UEs for executing another agent according to the demand of each agent. The relevance of this feature is demonstrated in (FRANCESQUINI; GOLDMAN; MÉHAUT, 2013), in the context of actors, where some actors have a high demand of communication while others do not have such a demand.

3.2 AGENT LEVEL CONCURRENCY

Current execution platforms typically adopt a sequential *synchronous* execution of the reasoning cycle, involving a sequence of steps that compose the reasoning cycle. The main example is the PRS cycle (INGRAND; GEORGEFF; RAO, 1992), which its synchronous reasoning cycle is inspiration for most execution platforms, such as (PEREIRA; QUARESMA; CENTRE, 1998; KOWALSKI; SADRI, 1999; SHANAHAN, 2000; BARAL; GELFOND, 2000; BORDINI; HÜBNER; WOOLDRIDGE, 2007; RICCI; VIROLI, 2007; RICCI; VIROLI; PIANCASTELLI, 2011; RICCI; SANTI, 2012, 2013; SANTI; RICCI, 2013; POKAHR; BRAUBACH; LAMERSDORF, 2005b; POKAHR; BRAUBACH; JANDER, 2010; EVERTSZ et al., 2003; WINIKOFF, 2005; EITER; SUBRAHMANIAN; PICK, 1999; DIX; ZHANG, 2005; SHOHAM, 1991; COLLIER; RUSSELL; LILLIS, 2015a; RODRIGUEZ, 2005; VIKHOREV; ALECHINA; LOGAN, 2011; ALECHINA; DASTANI; LOGAN, 2012). In this kind of execution, firstly, new goals and facts are obtained, then plans

are triggered considering the new beliefs and one or more applicable plans are selected becoming an intention. At the end, the selected intentions are executed. A new cycle cannot start until the current cycle has finished. In a large MAS (e.g., where an agent can have a lot of percepts and messages to handle), such cycle can take a long time to execute and reactivity may not be ensured. The result is that agents lose reactivity, which can be a drawback in a dynamic scenario with several messages arriving or constant changes in the environment happening. This is a common reason for introducing an *asynchronous* execution for the agent reasoning cycle. In the asynchronous execution, the cycle is divided in stages that can be executed concurrently, so that a stage does not need to wait for the termination of other stage to start its execution (ZHANG; HUANG, 2008, 2006a; KOSTIADIS; HU, 2000; COSTA; BITTENCOURT, 2000; GONZALEZ; ANGEL; GONZALEZ, 2013).

The amount of time that an agent holds the CPU is usually related to how long the agent reasoning cycle takes to execute. In an extreme, and highly adopted by current execution platforms, a UE can execute *one* reasoning cycle every time that an agent is selected for execution. However, in several scenarios and considering the hardware infrastructure, the execution of a single reasoning cycle is fast enough to introduce overheads when adopting a pool strategy, especially related to the competition of UEs for accessing the queues of agents (e.g., to enqueue and dequeue agents) every time that an agent completes the reasoning cycle. The reasoning cycle could be executed several times when an agent is selected for execution, which can increase the load of the job that a UE executes, also reducing the number of times that the operations to enqueue and dequeue agents are performed. On the one hand, such approach could harm the fairness in the MAS (e.g., an agent could execute more actions before another agent executes). Agents should have the same chances to be executed in a quasi-parallel form. On the other hand, the execution can be improved due to a heavier job that a UE needs to execute. It can bring positive results in the case of a greater number of PEs, minimizing the overhead caused by the concurrent access to the agents queue and the overhead of the operations to enqueue and dequeue agents every time that an agent completes its reasoning cycle.

The dependencies among the different reasoning cycle stages are clearly identifiable. The deliberate stage cannot execute if there are no events produced by the sense stage (e.g., when new beliefs are added in the belief base) and the act stage cannot execute if there are no active intentions produced by the deliberate stage. Moreover, an agent

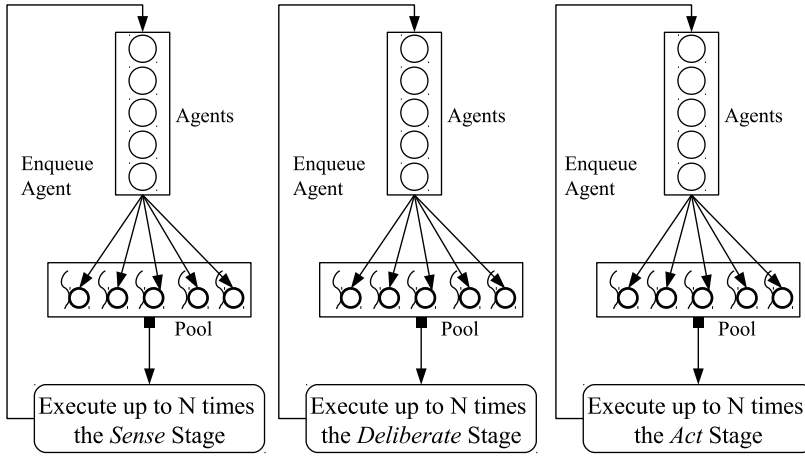

Figure 7 – Asynchronous execution of the reasoning cycle.

cannot deliberate about an event considering that the belief base is not fully updated. Said that, two main directions can be taken in order to improve the execution of the reasoning cycle considering the asynchronous execution of its stages. The first one is to weaken some of the dependencies so that the stages can be carried on independently, where different pools (and queues) can be used for the execution of the three reasoning cycle stages asynchronously. As illustrated in Fig. 7, an agent can be present in the three queues simultaneously, and be selected for the execution by the UEs from each stage at the same time. In the case of Fig. 7, we consider only weakening the dependencies that can keep the agent reasoning consistent. For example, the deliberate stage can be continuously executed concurrently while it has events to handle and the belief base is not being updated by the sense stage. The deliberate stage does not need to interrupt its execution every time that the sense or act stages are being executed, thus new intentions can be produced continuously. The act stage can also be executed concurrently while it has active intentions to execute. The concurrent (asynchronous) execution of the stages brings both benefits and drawbacks. The main benefit is that agents can react to emergencies promptly. The challenges are mostly related to how to keep the same semantic of a program for both the synchronous and the asynchronous reasoning cycles execution.

The concurrent and uncontrolled access to the same data structures by different stages of the asynchronous reasoning cycle could easily lead the MAS to an inconsistent state. The critical part to keep consis-

tency is to use control access mechanisms to avoid interferences among UEs from different stages (e.g., when accessing the belief base both for reading (deliberate stage) and writing (sense and act stages)). Interferences can happen in several points:

- on the addition and selection of events, which, at some point, use a shared data structure to store the events that will be accessed by UEs from the sense stage (e.g., to add events related to new beliefs) and by UEs from the deliberate stage (e.g., to select an event to handle);
- on the addition and selection of intentions, which, at some point, use a shared data structure to store the intentions produced by the deliberate stage and that will be executed in the act stage;
- on the belief update and the deliberate stage. In this case, the belief base is updated atomically, otherwise the agent could reason about partial updates of the belief base even in scenarios where it should not be allowed (e.g., in a chess game, the agent needs to know the situation of the whole board updated in order to make the best decision). Moreover, updates in the belief base related to a single percept needs to be done atomically, otherwise, in the deliberate stage, the agent could see states of the belief base that do not correspond to any possible state of the world.

The second direction consists in keeping the dependencies among the stages (i.e., the synchronous execution), but executing the three stages of the reasoning cycle in different moments (Fig. 9) instead of executing the three stages everytime that an agent is selected by a UE (Fig. 8). A single pool and a single queue could be used for this purpose. A UE would pick an agent from the queue, would execute a stage of the reasoning cycle (e.g., sense), and then enqueue the agent again for the execution of the next stage (e.g., deliberate), applying optimization when possible (i.e., skipping stages if there is no need to execute them). In a variation of this approach, different pools (and queues) could be used for the three stages. Each stage of the reasoning cycle could be executed by UEs of its own pool and agents are enqueued in three different queues: the sense queue, the deliberate queue, and the act queue. However, the same agent can be only present in one queue at the same time. A UE from a certain stage (e.g., sense) would pick an agent from its queue (e.g., the sense queue), execute the stage (e.g., sense), and then enqueue the agent in the queue of the next stage (e.g., deliberate), applying optimization when possible. The main expected

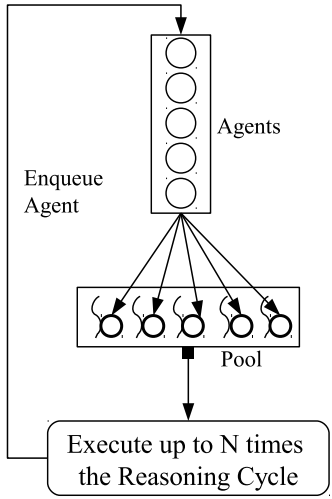

Figure 8 – Synchronous execution of the reasoning cycle.

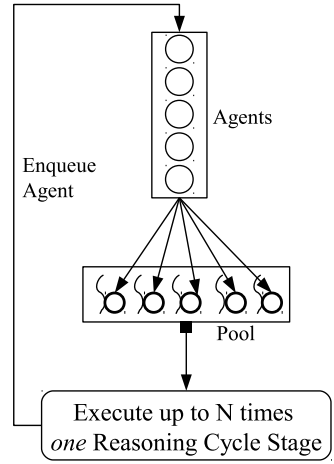

Figure 9 – Synchronous execution of the three reasoning cycle stages in different moments.

result is that agents have a closer response time to one another, since firstly all agents sense, then all agents deliberate, and finally all agents act.

Moreover, the same stage of the reasoning cycle could be executed more than once. For example, when an agent is selected from the deliberate queue, the UE can execute the deliberate stage several times before to enqueue the agent again. It can bring benefits in the sense that intentions can be created earlier and the overheads caused by the concurrent access to the deliberate queue and by enqueueing and dequeueing the agents from the deliberate queue can be minimized. The same analysis is also valid for the sense and act stages.

3.3 INTENTION LEVEL

The analysis of the dependencies can be also done at the intention level, leading us to further refine the granularity. A first consideration is that the agent goals are typically carried on by *independent intentions* that can be executed in two different forms. (1) Intentions can be executed concurrently by their own UEs (DELOACH, 2001). Everytime that the agent creates a new intention, a new UE is launched to execute

the intention, i.e., if the agent has three intentions, the agent will use three UEs to execute them. Languages that support this feature are Go!, ConGolog, IndiGolog, JADE, Jadex, JACK, Ciao Prolog, GAEA, Jinni. (2) Intentions can be executed concurrently without using a UE for each one. For example, each agent can have a queue of intentions and the execution platform interleaves their executions by means of selecting them in turns, which is controlled by the agent reasoning cycle. In each execution of the reasoning cycle, one intention from the queue is selected and the main UE of the agent executes part of it (e.g., one deed). Such feature provides internal concurrency without a high number of UEs, which is useful for the agent to perform several activities in an almost *simultaneous* form. This strategy is adopted by languages like Jason, simpA, simpAL, ALOO, JADE, Jadex, JACK, 3APL, 2APL, IMPACT, AgentSpeak(L), ASTRA, N-2APL, AgentSpeak(RT), and Agent Factory Framework.

Another important point that needs to be considered is when concurrent intentions can simultaneously produce actions and operations that could be executed in a *single shot* (i.e., instead of only executing *one* action or operation of *one* intention, *several* actions or operations from different intentions could be executed). Although intentions are independent courses of actions, the semantics for the concurrent execution of actions and operations needs to be carefully specified (and implemented) to avoid interferences. On the one hand, operations that change the internal state of the agent (e.g., operations to add and remove a belief or plan) should be executed in a way that results are consistent. On the other hand, we can consider that the concurrent execution of environment actions should be managed by the environment (like in CArtAgO (RICCI et al., 2009)), thus the concurrent production of environment actions by the agent should not lead the environment to an invalid state.

A further feature to improve the execution of intentions is to allow forking the plan body of an intention in more plan bodies that can be executed concurrently. This improves the level of concurrency when independent courses of deeds can be identified in the plan body and thus carried-on concurrently. Such a similar feature is supported in languages like ConGolog, IndiGolog, ViP, simpA, simpAL, ALOO, JACK, Blueprint, and JIAC Agent Framework.

The concurrent execution of intentions can lead the agent to conflicts. Sec. 3.3.1 makes an analysis of how to handle conflicting intentions.

3.3.1 Conflicting Intentions

When executing intentions concurrently it is important to consider the detection and avoidance of the execution of *conflicting intentions* (THANGARAJAH; PADGHAM; HARLAND, 2002; THANGARAJAH; PADGHAM; WINIKOFF, 2003a; RIEMSDIJK; DASTANI; MEYER, 2009; WANG; CAO; WANG, 2012; ALECHINA; DASTANI; LOGAN, 2012; COLLIER; RUSSELL; LILLIS, 2015a). Conflicting intentions are, for example, to go search for victims and to go charge the battery, which try to move the agent in opposite directions. If executed concurrently, they can interfere with each other and the agent would remain moving back and forth intermittently, failing to successfully complete any intention.

The concept of conflicts in the agent paradigm is not something novel, and it may be related to some extent to problems in other paradigms, such as critical sections or atomic blocks in multi-threaded programming, once intentions can be seen as equivalent to *threads* and plans are equivalent to methods (COLLIER; RUSSELL; LILLIS, 2015a). A first simple and quite adopted approach for handling conflicting intentions is the developer to explicitly specify plans as atomic, as provided in languages like 2APL, Jason, and JIAC. For example, in Jason, at runtime, when an intention contains an atomic plan (becoming an atomic intention), its execution disables the round-robin scheduling mechanism of the reasoning cycle, and the atomic intention is the only one selected for execution until the termination of the atomic plan. The main drawback of this approach is that it would also constrain the concurrent execution of non-conflicting intentions. A *lighter* version of atomicity is implemented in N-2APL, AgentSpeak(RT), and ALOO, where atomic plans are not executed concurrently, but non-atomic plans can be executed concurrently together with one atomic plan. For example, if intention α and intention β are atomic, the execution platform does not execute them concurrently. However, intention γ, which is not atomic, can still be executed concurrently with any of them.

ASTRA gives a further step on handling conflicting plans in the agent paradigm by allowing the specification of multiple critical sections (inspired by the `synchronized` mechanism in Java), thus only preventing the concurrent execution of conflicting pieces of code with the same critical section identifier. Likewise, other works also consider the conflicts only among certain goals (POKAHR; BRAUBACH; LAMERSDORF, 2005a; RIEMSDIJK; DASTANI; MEYER, 2009; WANG; CAO; WANG, 2012), that is, a conflict among α and β can be handled without interfere with a conflict among γ and δ. In (POKAHR; BRAUBACH; LAMERSDORF,

2005a; WANG; CAO; WANG, 2012), the authors use Jadex as an agent platform to support explicit specification of which goals are conflicting. In (RIEMSDIJK; DASTANI; MEYER, 2009), the authors focus on representing and specifying conflicts among goals.

While in (POKAHR; BRAUBACH; LAMERSDORF, 2005a; COLLIER; RUSSELL; LILLIS, 2015a; DASTANI; TORRE, 2004) the developer have to explicitly specify which goals or plans are conflicting, in (RIEMSDIJK; DASTANI; MEYER, 2009), besides to let that a developer explicitly specify which goals are conflicting, conflicts are detected automatically if goals are logically inconsistent (e.g., $\neg g$ and g). A similar mechanism to detect conflicts among logically inconsistent goals is implemented in the GOAL language (BOER et al., 2007). The authors also highlight that although the goals $\neg g$ and g cannot be achieved at the same time, they can perfectly coexist in the goal base of the agent, once these goals can be achieved in different moments. Except by the approaches adopted in (COLLIER; RUSSELL; LILLIS, 2015a), whose conflicts are handled in the plan level, the works that handle the conflict in the goal level do not take benefits from the existence of non-conflicting alternative plans that can achieve the same goal. All plans to achieve the goals are considered conflicting.

A further way to detect conflicts is presented in a series of papers by Thangarajah and his colleagues (THANGARAJAH et al., 2002; THANGARAJAH; PADGHAM; HARLAND, 2002; THANGARAJAH; PADGHAM; WINIKOFF, 2003b, 2003a; THANGARAJAH; PADGHAM, 2011). The authors propose a mechanism to detect interferences among intentions considering negative and positive interferences, which means that not only the execution of conflicting intentions are avoided, but the execution of intentions that have positive interferences are motivated. A positive interference happens, e.g., when two or more intentions have a common sub-goal, such as to be in the same place in a city, which means that if its execution is scheduled properly, the achievement of a single instance of that sub-goal can benefit all those intentions (i.e., a different instance of the sub-goal is not necessary to be achieved for each different intention). Other works that deal with positive interferences are (HORTY; POLLACK, 2001; THANGARAJAH; PADGHAM; WINIKOFF, 2003b; COX; DURFEE, 2003).

The detection of conflicts in (THANGARAJAH et al., 2002; THANGARAJAH; PADGHAM; HARLAND, 2002; THANGARAJAH; PADGHAM; WINIKOFF, 2003b, 2003a; THANGARAJAH; PADGHAM, 2011) is done automatically by a mechanism that uses *summary information* about goals and plans. In the summary information, we can find information about *effects*,

preparatory-effects, *in-conditions* of goals and plans, as well as the *resources* required to achieve a goal or to execute a plan. The effects are the conditions that are expected to be true after the execution of a goal or plan. A preparatory-effect is an effect e caused by the execution of a plan $p1$, and e is a pre-condition for a plan $p2$ starts to execute. In this case, a *dependency-link* between e and $p2$ exists. The in-conditions are the conditions that have to remain true while the goal or plan is active. At run-time, a data structure (*GuardedSet*) stores the current status of the resources as well as the information about which in-conditions and dependency-links are active and need to be guarded. In-conditions and dependency-links remain guarded until the termination of the associated plan or goal. Before an agent adopts a new goal or execute a new plan, their in-conditions, effects, and required resources are analyzed against the *GuardedSet* and the current state of the resources. If they are not compatible, i.e., if their executions could undo the effect of some dependency-link or make false the in-condition of some active goal or plan, the new goal or plan should be re-considered when their incompatibility is solved (e.g., when the dependency-link that is causing the incompatibility is complete).

The technique of summary information adopted in (THANGARAJAH et al., 2002; THANGARAJAH; PADGHAM; HARLAND, 2002; THANGARAJAH; PADGHAM; WINIKOFF, 2003b, 2003a; THANGARAJAH; PADGHAM, 2011) is also used in previous work in the agents literature, such as in the work by Clement et al. (CLEMENT; DURFEE, 1999a, 1999b), however with the difference that conflicts are solved and the scheduling for the plans execution is done before the execution (i.e., a new plan is produced by means of merging the agent plans). Similar approaches to solve conflicts and merge the execution of plans before the execution are presented in (BOUTILIER; BRAFMAN, 1997; SUGAWARA et al., 2005). The main limitation of this kind of approach is that autonomous agents that act in a dynamic environment may not know in advance all the plans that might be actually executed (e.g., through communicating new plans or creating new plans through planning) because summary information are produced at compile time. A mechanism to deal with dynamic plan libraries is an essential feature of realistic autonomous agents. Finally, some of the work above consider also inter-agent conflicts (CLEMENT; DURFEE, 1999a, 1999b; BOUTILIER; BRAFMAN, 1997; SUGAWARA et al., 2005), where plans from different agents can conflict if executed together.

In (SHAPIRO et al., 2012), the authors propose a way to handle conflicts by considering alternative plans to achieve the goals. In this

case, if the first option of plan to achieve the goal conflicts with other active intention, the agent can try a second option, and so on. The main benefit is that the agent can pursue and achieve more goals, once the agent can always search a non-conflicting plan to achieve the goal. Different from the previous works, the proposal in (SHAPIRO et al., 2012) does not only refers to the concurrent execution of intentions, but also to intentions that may be not achieved even if executed sequentially. For example, an agent may have the intention to buy a laptop and a printer, however such agent may have a limited amount of money to buy both and, as a solution, the agent may opt either for different brands in order to buy both a laptop and a printer or give up of some of them.

Another important point is what to do when a conflict is detected. While a common approach is to simply suspend the intention that is trying to execute a conflicting plan, according to (MILLER; TRIBBLE; SHAPIRO, 2005), two kinds of plans can be defined: plans that execute immediately and plans that execute eventually. In plans that execute immediately, an intention α that is trying to execute a new plan has preference and the intentions that conflict with α will be put aside on a *to-do* list. The intentions in the *to-do* list will be processed when α terminates. Instead, in plans that execute eventually, the intention α is suspended in the case of conflict. In (THANGARAJAH; PADGHAM; HARLAND, 2002; THANGARAJAH; PADGHAM; WINIKOFF, 2003b, 2003a), the authors defend that when two intentions conflict, a more reasonable way to handle it is by allowing them to be pursued concurrently, but monitoring the execution of the deed of the intentions which cause the conflict and scheduling them in a way that they do not interfere with each other. In their work, it is done by guarding the in-conditions and dependency-links that are active when new intentions start to execute new plans. The in-conditions are conditions that have to remain *true* while the intention is active. The dependency-links are created when an early deed of the intention *pave* the way for the execution of later deeds. For example, when two plans $p1$ and $p2$ are utilized to satisfy goal g, $p1$ executes before $p2$, and $p1$ brings about an effect e, which is a pre-condition for the execution of $p2$, then there is a dependency-link between the preparatory effect e and the dependent-plan $p2$. These preparatory effects achieved by a plan are protected from the effects of new plans until the relevant dependency-links complete, i.e., when the dependent plans begins the execution (e.g., $p2$ in the example before).

Summing up the different works that address the problem of conflicting intentions, we observed that there are three main conceptual

levels where the relation of conflicts can be considered. (1) The conflicts can be among goals; (2) among plans; or (3) among parts of plans. Table 1 presents a summary of the presented works and their choices. The names in italic are works that are not related to an agent language or tool.

Work	(1)	(2)	(3)
ASTRA			✓
simpA		✓	
simpAL		✓	
ALOO		✓	
Jason		✓	
JIAC		✓	
AgentSpeak(RT)		✓	
2APL		✓	
3APL		✓	
N-2APL		✓	
JACK		✓	
Shapiro et al.		✓	
Thangarajah et al.		✓	
Clement et al.		✓	
Boutilier et al.		✓	
Sugawara et al.		✓	
GOAL	✓		
Jadex	✓		
Riemsdijk et al.	✓		

Table 1 – Conceptual levels of conflicts.

The conflicts can be detected in three main ways. (1) The developer can explicitly specify the conflicts (by means of meta-information or annotation). (2) Conflicts can be detected by means of identifying logically inconsistent goals. (3) Conflicts can be detected by means of inspecting the code of the plans at run-time, looking forward for possible interferences caused by the deeds of the plan bodies, or by means of summary information, when compiling the necessary information for automatic detection of conflicts at run-time. Table 2 presents a summary of the presented works and their choices.

Work	(1)	(2)	(3)[1]
ASTRA	✓		
simpA	✓		
simpAL	✓		
ALOO	✓		
Jason	✓		
JIAC	✓		
AgentSpeak(RT)	✓		
2APL	✓		
3APL	✓		
N-2APL	✓		
JACK			✓
Shapiro et al.	✓		
Thangarajah et al.			✓
Clement et al.	✓		
Boutilier et al.	✓		
Sugawara et al.	✓		
GOAL		✓	
Jadex	✓		
Riemsdijk et al.	✓	✓	

Table 2 – Detection of conflicts.

The conflicts can be handled in three main ways. (1) If an atomic intention is in execution, no other intentions can execute concurrently until the atomic intention terminates. (2) Non-atomic intentions can be executed concurrently even if an atomic intention is in execution. (3) Intentions can be executed concurrently if they do not have the same identifier of conflict or they do not have a direct relation with the conflict (e.g., in the case of logically inconsistent goals). Table 3 presents a summary of the presented works and their choices.

Some languages (e.g., simpAL, JADE, Jason, and Jadex) also provide operations that can be performed over intentions, such as suspend, resume, and inspect their current state (e.g., check if some intention is suspended), so that an agent can be explicitly programmed to deal with possible interferences and conflicts among the execution of

[1]No work can detect conflicts at run-time without any meta-information or annotation provided by the developer.

two or more intentions. With such kind of operations, the agent can control its activities that are active in a certain moment.

Work	(1)	(2)	(3)
ASTRA			✓
simpA		✓	
simpAL		✓	
ALOO		✓	
Jason	✓		
JIAC	✓		
AgentSpeak(RT)		✓	
2APL	✓		
3APL	✓		
N-2APL		✓	
JACK			✓
Shapiro et al.			✓
Thangarajah et al.			✓
Clement et al.			✓
Boutilier et al.			✓
Sugawara et al.			✓
GOAL			✓
Jadex			✓
Riemsdijk et al.			✓

Table 3 – Handling conflicts.

3.4 ANALYSIS

Considering our perspective for concurrency in MAS as presented in this chapter, Table 4 summarizes the related work and the features identified to deal with concurrency in MAS (These works are detailed in Appendix B). In the table, we only consider works that provide a complete tool for MAS development, that is, at least both a language and an execution platform. In order to keep the table columns in a single page we give a letter for each feature identified. Moreover, in order to simplify the analysis, we considered *behavior*, *task*, *job*, *activity* like *plans* and *intentions*, once all them refer to what the agents do. We

refer to plan as the specification of a sequence of deeds that agents need to perform, while intentions are instances of plans.

The meaning of the letters and the identified features are presented bellow. The features are ordered according to some characteristics. Features `A`, `B`, `C`, and `D` are related to the different ways to launch intentions. Features `E`, `F`, `G`, and `H` deal with intentions running concurrently and related issues, such as conflict avoidance and priorities. Features `I`, `J`, and `K` are related to the agent architectures. Finally, features `L`, `M`, and `N` give an overview of how UEs are allocated to the agents in a MAS.

A: Intentions can run concurrently even without one UE for each one. The interpreter of the language interleaves the execution of the intentions. A bit of each intention is executed each time. For example, if two intentions (α and β) are executing, a bit of intention α is executed and then a bit of intention β is executed.

B: Each intention can run in a different UE. For example, everytime that the agent creates a new intention, it will run the intention in a different UE. If the agent is running 6 intentions, the agent will use 6 UEs.

C: The developer can explicitly decide when to create a new UE to execute an intention. For example, it is possible to explicitly create a new UE to run intention α.

D: Mechanisms for fork and join (creating or not a UE). For example, it is possible to write a plan α that calls the plan β and δ to run concurrently (in the same or different UEs) and waits both plans (β and δ) terminate to proceed with the execution.

E: Operations over intentions, such as suspend and resume their executions, and inspect their current state.

F: Atomic plans. For example, if the agent has plan α and β in its code, it is possible to define that only plan β is atomic.

G: Intentions can be executed according to priorities. Thus, agents can focus their attention on specific intentions. For example, intention α may have a higher priority than intention β.

H: Detect and avoid the execution of conflicting intentions. For example, if intention α and intention β conflict, the language can detect the conflict and avoid them to execute concurrently.

I: Asynchronous approach for the loop of the reasoning cycle, where the agents are composed of different components running concurrently. For example, there are different components to manage beliefs and intentions.

J: Agents can be composed of other agents. For example, the subagents could be responsible for controlling specific parts of higher level agent, such as its beliefs or its reactive behavior.

K: The communication can be managed by dedicated UEs. That is, the sending and receiving mechanisms are not handled by the agent's main UE. Each communication channel could be managed by different UEs. For example, if agent Ana communicates with agents Bob and Carl, there are specific UEs for the channels Ana-Bob and Ana-Carl.

L: One dedicated UE can be used for each agent (each agent is a UE). For example, if the MAS has 5 agents, it also uses 5 UEs.

M: All the agents can share a pool (usually with less UEs than agents). For example, the developer can specify that 100 agents will use 5 UEs that will be shared among them.

N: Each agent can use more than one UE.

Work	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Jason	✓				✓	✓						✓	✓	
simpA														
simpAL	✓			✓	✓	✓							✓	
ALOO														
JADE	✓	✓	✓		✓							✓		✓
Jadex	✓	✓	✓		✓	✓						✓		✓
JACK	✓	✓	✓	✓				✓					✓	✓
3APL														
2APL	✓				✓	✓	✓					✓		
N-2APL														
CLAIM	✓									✓				
Qu-Prolog														
QuP++		✓	✓	✓							✓			✓
Go!														
MINERVA	✓									✓				
ConGolog		✓	✓	✓			✓							✓
IndiGolog		✓	✓	✓			✓							✓
ViP				✓										
Ciao Prolog		✓	✓											✓
GAEA		✓	✓											✓
Jinni		✓	✓											✓
Blueprint	✓			✓									✓	
IMPACT	✓													
METATEM												✓		
E_{hhf}		✓												✓
Agent Factory Framework	✓												✓	
JIAC Agent Framework	✓	✓	✓	✓		✓						✓		✓
Vivid agents									✓					✓
GOAL							✓						✓	
ASTRA	✓					✓							✓	
SARL		✓								✓			✓	✓
AgentSpeak(RT)	✓					✓	✓					✓		

Table 4 – Summary of concurrency features provided in agent languages and platforms.

According to Table 4 it is possible to see that current works still cover a limited subset of the identified features. Each single line of the table, which represents a work, can only provide few features and, for example, it is not possible to get the best of a computer to solve certain kinds of problems using only one of them. Another interesting aspect about the works presented in Table 4 is that most works do not let the MAS developer to change configurations related to concurrency aspects of the agents. For example, to choose between to define a number of UEs to be shared among the agents (feature `M`) and to define that each agent has its own UE (feature `L`).

Table 5 and Table 6 compact the data presented in Table 4. The aim of Table 5 is to show the relation between number of features (line *Features*) and the number of works that provide such number of features (line *Works*). Thus, we can see that 3 works provide only 1 feature, while 5 works provide 3 features, and so on. The maximum of features implemented by some work is 7. Finally, the last line of the table (*Accumulated (works)*) presents the number of works that provide N or more features. Thus, we can see that only 12 works provide 4 or more features while the other 14 works provide less than 4.

Features	7	6	5	4	3	2	1
Works	3	1	6	2	5	6	3
Accumulated (works)	3	4	10	12	17	23	26

Table 5 – Number of features versus number of works.

Table 6 shows the number of works that provide certain feature. Thus, we can see, for example, that 12 works allow the agent to launch intentions in dedicated UEs (feature `B`), while only 1 work use an asynchronous approach for the agent reasoning cycle (feature `I`). It is interesting to note that 13 works allow the use of more than one UE for each agent (feature `N`). This is easily explained because the works that support the features `B`, `C`, `I`, and `K` also support the feature `N`. Another relation that we can make is that if the MAS developer cannot change the configuration to enable and disable the features `B`, `I`, `K`, and `N`, the language cannot support the feature `L`. This happens because even if the agent is created with a single UE, it will easily use more UEs during its execution, for example to handle some message (feature `K`) or to execute some intention in a dedicated UE (feature `B`).

Considering only the last three features of the table (`L`, `M`, and

`N`) it can already give us an idea about how the works would perform in a MAS composed of few agents and in a MAS with a large number of agents. Notice that if such works do not let the MAS developer to change between these three configurations, the MAS would perform much worse in some scenarios. For example, in a MAS composed of 10,000 agents, works that support the feature `M` are expected to perform much better than works that only support feature `L`, once the number of UEs for feature `L` would be quite high, as already demonstrated in experiments (MUSCAR, 2011; MUSCAR; BADICA, 2011).

The same analogy can be done for features `A` and `B`, which can measure the level of internal concurrency in an agent. Taking account these two features we can have an idea about how the works would perform according to the number of intentions that the agents execute concurrently. Works that only consider feature `B` are expected to perform worse in scenarios where the agents concurrently execute a number of intentions significantly higher than the available PEs.

Feature	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Works	14	12	10	8	5	7	5	1	1	3	1	7	8	13

Table 6 – Number of works for feature.

The numbers presented in these tables are especially related to the fact that most current works provide concurrency features according to the problem that is being solved using agents, which means that some works deal with very specific scenarios and it is out of their scope to better exploit concurrency in other scenarios. This is the point where this thesis can enrich the current state-of-the-art. The main difference between this thesis and the existing works is that this thesis is not focused on specific scenarios, but its aim is to develop a solution that covers a wide set of scenarios that could have their executions improved according to the concurrency feature that is being used by the MAS. Therefore, the resulting architecture and execution platform needs to support as much concurrency features as possible, allowing the MAS developer to use the most suitable configuration according to the requirements and characteristics of the final application.

In addition, other important features, which are not considered by any of the presented works, can be identified. One of them is that it is not possible to *define* which plans conflict among each other and only execute concurrently plans that do not conflict. This feature is different from feature `H` in the sense that the control to define the conflicts is put

on the MAS developer hands.

Although the number of features adopted by a work does not mean that it is better than others, the capability to choose among the different features that one can use to execute a particular application is certainly an important characteristic. For example, all works analyzed that support the feature to run internal components of an agent concurrently cannot execute the intentions of an agent concurrently without assign one UE for each intention, which can be clearly a limitation when many intentions are being executed by an agent.

3.5 CONCLUSION

In this chapter, we analyzed different means to exploit concurrency in the MAS execution aiming to take advantage of parallel (multicore) architectures. The main result of this analysis is the identification of the aspects in a MAS, agent, and intention levels where the concurrency can be exploited to bring benefits for the MAS execution and each individual agent. Among the concurrency features identified in the current works, agents can use one (Feature `L`) or more UEs (Feature `N`) or even share UEs among themselves (Feature `M`); intentions can run on dedicated UEs (Features `B` and `C`), run concurrently even without a dedicated UE for each intention (Feature `A`), or even fork the intentions that are in execution (Feature `D`); the agent architecture can be composed of different components running concurrently and managed by different UEs (Features `I` and `K`) or be composed of other agents to perform different activities concurrently (Feature `J`). Finally, different features are also provided to deal with intentions that run concurrently in order to avoid certain conflicting intentions to run concurrently. Some works provide mechanisms to define plans as atomic (Feature `F`), to identify an avoid conflicting intentions (Feature `H`), to perform operations over intentions such as resume and suspend their executions (Feature `E`), to define priorities in order to decide which intentions to execute first (Feature `G`).

The analysis also showed that there is not an agreement about which is the best way to exploit concurrency in the agent paradigm. Each work only provides a small set of features, which the authors consider important for the scenarios where their work will be applied. Thus, the features can vary from work to work. However, such analysis is important and raised directions to enrich an execution platform to better support concurrency.

In the next chapter, we put together different concurrency features by proposing an BDI agent and MAS model and architecture, which aims to fill the gap of adopting certain concurrency features and being flexible to let the MAS developer to decide which feature to exploit and configure it according to the needs of the application.

4 A BDI MAS CONCURRENT MODEL AND ARCHITECTURE

This chapter presents our BDI agent and MAS model and architecture focused on concurrency. We start from the model and architecture adopted in Jason and then we modify them to achieve a platform without the drawbacks that we identified in Chapter 3. Jason is chosen as the starting point because it already supports some of the concurrent features identified in the state-of-the-art and due to our familiarity and experience with Jason. Sec. 4.1 presents an overview of how concurrency is exploited in the MAS level. Sec. 4.2 presents how the concurrency is exploited in the agent level. Sec. 4.3 presents how the concurrency is exploited in the intention level. Finally, conclusions are presented in Sec. 4.4.

4.1 MAS LEVEL

Fig. 10 presents an overview of our MAS model. A MAS is composed of agents and pools. Pools are composed of UEs that execute the agents. Several UEs can be used by the MAS in order to better exploit the PEs. The number of UEs can be greater than the number of PEs, which means that while some UEs "own the PE", others will be "sleeping". While some agents can be executed by their own UE, in other cases UEs are grouped in pools to execute all agents of the MAS. The execution of each agent by means of its own UE satisfies feature `L` (Sec. 3.4), which says that each agent can use one dedicated UE. Likewise, the use of pools to execute all agents of a MAS satisfies feature `M`, which says that agents can share the same pools.

We assume that the environment where agents are situated have their own UEs. How the environment uses its UEs is not the focus of this thesis and it remains as a future work.

4.2 AGENT LEVEL

In this section, we present how agent level features are integrated. In Sec. 4.2.1, we revise the BDI agent reasoning cycle to support synchronous and asynchronous execution. In Sec. 4.2.2, we present the BDI agent architecture, with the reasoning cycle and its auxiliary data structures.

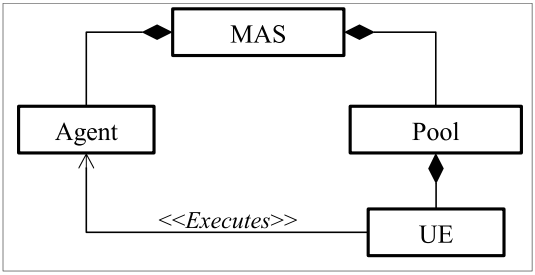

Figure 10 – MAS conceptual model.

4.2.1 Revising the Structure of the Reasoning Cycle

The agent reasoning cycle is conceived based on the BDI reasoning cycle introduced in Sec. 2.1.2 and inspired on the concurrent agent architectures proposed in (ZHANG; HUANG, 2008, 2006a; KOSTIADIS; HU, 2000; COSTA; BITTENCOURT, 2000; GONZALEZ; ANGEL; GONZALEZ, 2013), where agents are composed of different internal components that run concurrently. The Jason reasoning cycle presented in Sec. 2.1.2 is now explicitly divided in three main stages: the *sense* (left), the *deliberate* (middle), and the *act* (right). The distribution of the steps of the reasoning cycle among its three stages is based on which data those steps work on. In the sense stage, we gather the steps related to handling the input data of the agent (i.e., its percepts and messages) and producing events. In the deliberate stage, we gather the steps related to handling an event and producing an intention. In the act stage, we gather the steps related to the intention execution. Each stage can be executed independently and asynchronously in order to improve the reactivity of the agent and let the agent to continuously deliberate and act without stop sensing. Although the stages can be executed asynchronously, the full reasoning cycle (sense-deliberate-act) is still preserved, because the perception of a change in the environment is firstly updated in the belief base, then an event related to the belief update is produced, an intention is instantiated to handle the event, and finally the intention is executed.

The sense stage is executed almost in the same way as presented in Fig. 2. Thus, the agent starts by checking the environment and messages (`ChkEnvMsg`), then processing received messages (`ProcMsg`), updating belief base (`UpBB`), and producing events about belief changes and message exchanges (`AddEv`). However, at the end of the sense stage,

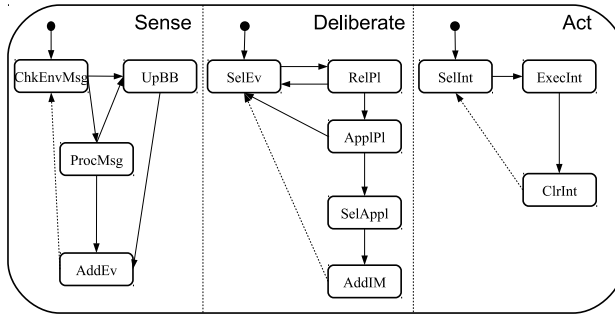

Figure 11 – New Jason reasoning cycle.

instead of starting to deliberate, the agent checks new percepts and messages again. Once the sense stage can be executed as a non-terminating cycle, events for the deliberate stage can be produced as fast as possible ensuring that emergencies or other higher priority situations be handled promptly.

As in the sense stage, the deliberate stage is executed similarly as in Fig. 2. Thus, the deliberate stage starts by selecting an event from the set of pending events (`SelEv`), then retrieving all relevant plans (`RelPl`), checking which of those are applicable (`ApplPl`), selecting one particular applicable plan (`SelAppl`), and adding the selected one to the set of intentions (`AddIM`). However, at the end of the deliberate stage, instead of start acting, the agent proceeds by handling another pending event. The main gain with the cyclical execution of the deliberate stage is that intentions are added in the act stage continuously. Thus, that emergency or high priority situation previously handled by the sense stage continues being promptly handled also in the deliberate stage, guaranteeing that an intention will be instantiated to handle it as soon as possible.

The changes in the act stage, compared to the act stage presented in Fig. 2, are also quite direct. Like the sense and deliberate stages, the act stage also executes cyclically by continuously selecting one of the active intentions (`SelInt`) and executing one of its deeds (`ExecInt`). After performing the last step of the act stage to remove the empty intentions (`ClrInt`), the act stage restarts its cycle. Such cycle in the act stage is the final step to guarantee that the emergencies and more priority situations be effectively handled promptly.

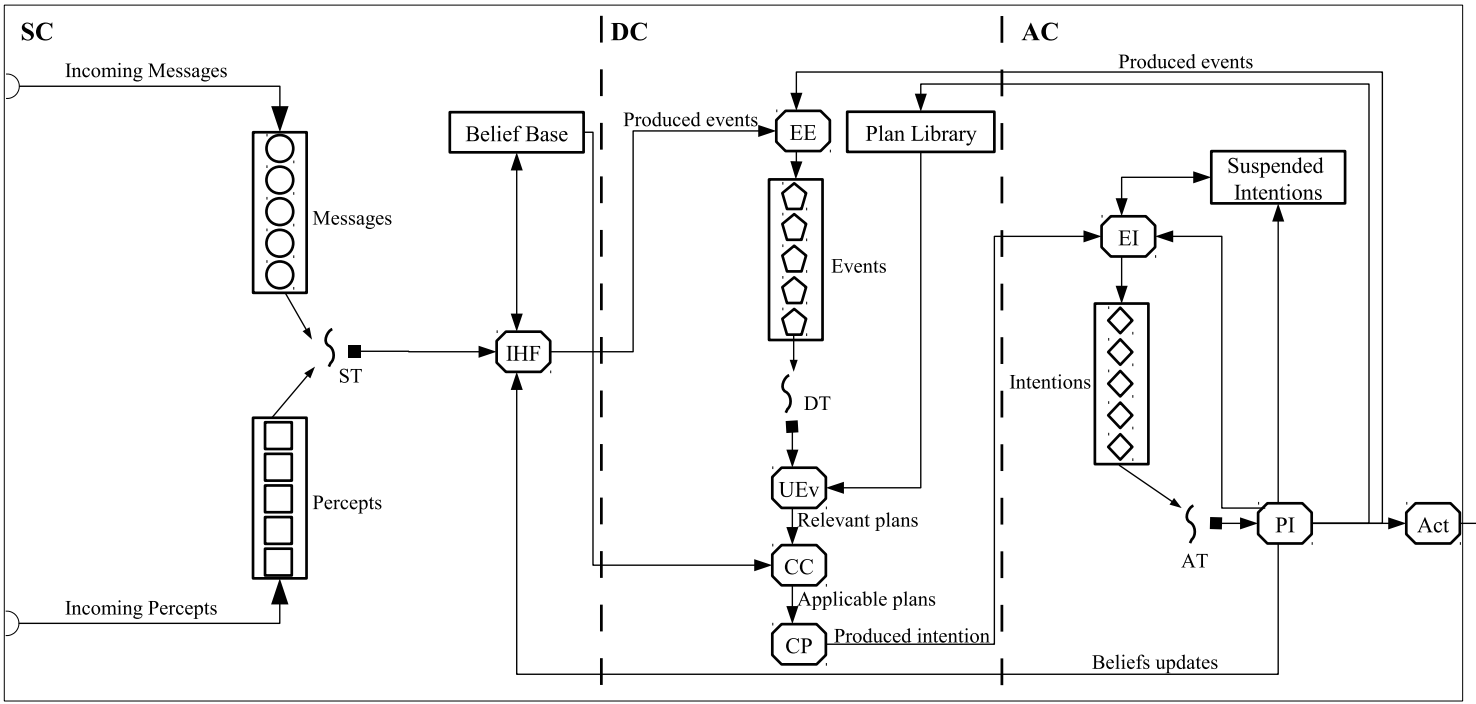

Figure 12 – Agent architecture.

4.2.2 Agent Architecture

Figure 12 presents the agent architecture for a concurrent BDI agent. Our agent architecture is based on some parallel BDI models (ZHANG; HUANG, 2005; KOSTIADIS; HU, 2000; COSTA; BITTENCOURT, 1999, 2000) and the BDI model adopted in Jason, which is an usual BDI model existing in agent languages. The idea of the concurrent architecture is to improve reactivity by allowing the agent to concurrently handle messages and incoming events from the environment; handle internal events produced by the arrival of messages, belief updates, goal adoptions, etc; and continue executing its intentions.

The agent is basically divided in three main components that can run concurrently, depending on the configuration. The *Sense Component* (SC) is responsible for receiving the inputs from the environment (*percepts*) and from other agents (*messages*), updating the belief base, and generating events. The *Deliberate Component* (DC) is responsible for reasoning about the events and produce new intentions to handle the events. Finally, the *Act Component* (AC) is responsible for executing the intentions. Such architecture satisfies feature `I` (Sec. 3.4), which says that agents use an asynchronous approach for the loop of the reasoning cycle.

In order to improve the visualization, the elements of each component are separated by the dashed lines. Each component can have its own UE, represented by the *Sense Thread* (ST), *Deliberate Thread* (DT), and *Act Thread* (AT). This UE can be the single UE that executes the agent, or a UE from a pool that aims to execute all agents from the MAS. The main reason for choosing only one UE to execute each component of an agent is to avoid contention that would be caused due to executing light jobs, such as to handle a single percept, which would cause several clashes of UEs trying to get percepts from the same queue of the pool. Moreover, considering that ST, DT, and AT are different UEs, the same agent can be executed by more than one UE at the same time, satisfying feature `N`, which says that each agent can use more than one UE.

```
while TRUE do
    sense()
    deliberate()
    act()
```

Algorithm 1: Synchronous execution when agents have their own UE.

```
i ← 0
while i < nCycles do
    sense()
    deliberate()
    act()
    i ← i + 1
```

Algorithm 2: Synchronous execution when agents are executed by a UE from a pool.

```
if current_stage = SENSE then
    sense()
    current_stage ← DELIBERATE
else if current_stage = DELIBERATE then
    deliberate()
    current_stage ← ACT
else if current_stage = ACT then
    act()
    current_stage ← SENSE
```

Algorithm 3: Synchronous execution when agents are executed by a UE from a pool and the three reasoning cycle stages are executed in different moments.

The agent reasoning cycle can be executed in two distinct ways: synchronous and asynchronous. In the synchronous execution, each component finish its execution before the other component starts its execution (i.e., the reasoning cycle is executed sequentially, step-by-step). The synchronous execution can be executed in three different ways. The first way considers the situation when each agent has its own UE (Algorithm 1). The UE executes the sequence sense-deliberate-act forever, i.e., until the termination of the agent. Thus, the job presented in Algorithm 1 is executed like depicted in Fig. 8, considering $N = \infty$. The second way to execute the reasoning cycle considers that an agent is executed by a UE from a pool (Algorithm 2). In this case, the UE that selects an agent, executes the sequence sense-deliberate-act for up to N times. Like the first form, the job presented in Algorithm 2 is executed like depicted in Fig. 8. The third way also considers that an agent is executed by a UE from a pool and everytime that a UE selects an agent, it executes only one stage of the reasoning cycle (e.g., sense) (Algorithm 3). The job presented in Algorithm 3 is executed like depicted in Fig. 9.

In the asynchronous execution, all components run concurrently and do not wait for other components to finish their executions before

```
job 1
  sense()
job 2
  deliberate()
job 3
  act()
```

Algorithm 4: Asynchronous execution.

doing something (i.e., steps of the reasoning cycle that belongs to different components are executed concurrently). Algorithm 4 presents the possible jobs that can be executed by a UE when selecting an agent for executing its reasoning cycle asynchronously. Considering that agents are enqueued in different queues (one for each stage of the reasoning cycle), each UE executes only one of the jobs to execute, which corresponds to which queue it selects the agent (e.g., from the pool to execute the sense component). The execution happens like depicted in Fig. 7. The support to asynchronous execution satisfies feature `I` (Sec. 3.4), which says that different components of an agent can be executed concurrently.

In Fig. 12, the elements represented by rectangles mean data sets, where data related to the concepts presented above are stored (such as *Messages*, *Percepts*, *Belief Base*, *Events*, *Plan Library*, *Intentions*, and *Suspended Intentions*). *Messages*, *Percepts*, *Events*, and *Intentions* are placed in queues and processed by the UE in their respective components. These queues could be implemented with some priority policy in order to process emergencies promptly (e.g., an event notifying low battery in a robot). Finally, elements represented by octagons mean functions that are used during the reasoning cycle of the agent. Such functions are used, for example, to act in the environment or manipulate the data sets.

More details about each component are explained in the following sections. Sec. 4.2.2.1 presents details about the SC, while Sec. 4.2.2.2 presents the DC and Sec. 4.2.2.3 presents the AC.

4.2.2.1 The Sense Component

The SC is responsible for the first steps of the reasoning cycle of the agent. The environment enqueues the messages and percepts for the agent. Percepts and messages are then processed by ST. ST executes the *Input Handler Function* (IHF) for each percept, message, and belief

update.

The IHF adds new beliefs related to percepts that are not currently in the belief base and removes beliefs that are no longer in the percepts from the environment (i.e., outdated beliefs). The addition and removal of beliefs always produce events that are included in the *Events* data set (by means of the function *Enqueue Event* (EE)) to be processed afterwards. The IHF also updates beliefs related to some kinds of messages. Agents can induce other agents to believe or to disbelieve something. Thus, according to the kind of message, the IHF adds or removes the beliefs. In addition, all received messages also produce events, even if they do not change the belief base (e.g., a message asking for some information). Algorithm 5 presents a pseudo-code for the function IHF while Algorithm 6 presents a pseudo-code for the function EE, and Algorithm 7 presents a pseudo-code executed by ST. In order to ensure coherence in the execution of the asynchronous reasoning cycle, control access mechanisms are necessary to be introduced due to the concurrent access to shared data structures, such as the belief base (e.g., the selection for an applicable plan and the belief update access the belief base). Although we assume that the belief base is thread safe in the sense that a single belief is updated atomically and the operations of read and write are controlled by the data structure of the belief base, a main issue is that, in some scenarios, the beliefs need to be considered as a whole, and not isolated. For example, in a game of chess, the agent needs to consider the situation of the whole board in order to make the best decision. Thus, a concurrency access mechanism is still necessary to ensure that the whole belief base will be updated atomically before the agent deliberates. In order to do it, the Algorithm 7 already introduces a lock. The same lock is also used in the deliberate stage, when the belief base is considered to select an applicable plan (Algorithm 9).

```
Procedure IHF(input)
    if input.kind = PERCEPT then
        newBelief ← produce_belief(input)
        if newBelief ∉ BeliefBase then
            BeliefBase.add(newBelief)
            EE(newBelief, belief_added)
        else if is_an_update(newBelief) then
            currentBelief ← BeliefBase.get(newBelief)
            BeliefBase.remove(currentBelief)
            EE(currentBelief, belief_removed)
            BeliefBase.add(newBelief)
            EE(newBelief, belief_added)
        if is_last_percept(input) then
            forall belief ∈ BeliefBase and belief.source = PERCEPT
            and is_outdated(belief) do
                BeliefBase.remove(belief)
                EE(belief, belief_removed)
    else
        newMessage ← produce_message(input)
        EE(newMessage, message_received)
```

Algorithm 5: Code for the function IHF.

```
Procedure EE(event, kind)
    Events.enqueue(event, kind)
```

Algorithm 6: Code for the function EE.

```
Procedure sense()
    i ← 0
    while i < nCyclesSense and (Percepts ≠ ∅ or Messages ≠ ∅) do
        bb.writeLock.lock()
        while Percepts ≠ ∅ do
            IHF( Percepts.dequeue() )
        bb.writeLock.unlock()
        while Messages ≠ ∅ do
            IHF( Messages.dequeue() )
        i ← i + 1
```

Algorithm 7: Code executed by ST.

4.2.2.2 The Deliberate Component

The DC is responsible for processing new events by including new intentions to handle them. The events in the *Events* queue are individually processed by DT. The first step to process an event is to

find the relevant plans to handle the event. It is done by retrieving all plans where the trigger can be unified with the event. The function *Unify Event* (UEv) is responsible for finding these plans. Algorithm 8 presents a pseudo-code for the function UEv.

```
Function UEv(event, plans)
    relevantPlans ← ∅
    forall plan ∈ plans do
        if unify(plan.trigger, event) then
            relevantPlans.add(plan)
    return relevantPlans
```

Algorithm 8: Code for the function UEv.

Next, the relevant plans are verified according to the their context, by means of the function *Check Context* (CC). The context of a plan determines if the plan can be applied or not in certain moments, depending on the current state of the beliefs of the agent. Thus, the CC function retrieves which plans, from the relevant plans, are currently applicable considering the current state of the beliefs of the agent and the context of the plans. Algorithm 9 presents a pseudo-code for the function CC.

```
Function CC(relevantPlans, beliefs)
    applicablePlans ← ∅
    bb.readLock.lock()
    forall plan ∈ relevantPlans do
        if is_applicable(plan.context, beliefs) then
            applicablePlans.add(plan)
    bb.readLock.unlock()
    return applicablePlans
```

Algorithm 9: Code for the function CC.

Several applicable plans can still be appropriate for handling the event, which means that the agent could choose any of them to handle the event successfully. Thus, the function *Choose Plan* (CP), by default, selects the first non-conflicting plan considering the order in which they appear in the plan library. If all applicable plans conflict with some already running intention, the first one is chosen. The plan library keeps the same order of the plans according to the source code. In addition, plans received from other agents, by means of messages, are added on the arrival order. The algorithm for the function CP as well as details about how conflicts are managed are presented in Sec. 4.3.1.

Finally, an intention is produced with the chosen plan and it is added in some of the *Intentions* data sets of the agent (by means of

the function *Enqueue Intention* (EI)) for a further execution. The EI includes the produced intention in the *Intentions* queue. Algorithm 10 presents a pseudo-code for the function EI while Algorithm 11 presents a pseudo-code executed by DT. The queue of intentions is another shared structure that needs to have an access control mechanism. We assume that the queue of intentions is thread safe, so that, an explicit concurrent access control is not necessary for the algorithm to enqueue an intention. The execution of the deliberate stage *nCyclesDeliberate* times also lets an agent to process *several* events in each reasoning cycle.

```
Procedure EI(intention)
    Intentions.enqueue(intention)
    testExecutableIntention(intention)
```

Algorithm 10: Code for the function EI.

```
Procedure deliberate()
    i ← 0
    while i < nCyclesDeliberate and Events ≠ ∅ do
        event ← Events.dequeue()
        if event ≠ NULL then
            relevantPlans ← UEv(event, PlanLibrary)
            applicablePlans ← CC(relevantPlans, BeliefBase)
            intention ← CP(applicablePlans)
            EI(intention)
        i ← i + 1
```

Algorithm 11: Deliberate.

4.2.2.3 The Act Component

The AC is responsible for the execution of intentions. The intentions are executed following the round-robin scheduling mechanism, where a different intention is selected everytime that the act is executed. Such scheduling mechanism satisfies feature `A` (Sec. 3.4), which aims to run intentions concurrently even without one UE for each one. The active intentions remain in the queue of *Intentions*, while suspended intentions are placed in the *Suspended Intentions* set, remaining there until the agent resumes or drops their executions.

The AT executes one deed of certain intention at once by means of the function *Process Intention* (PI). In each deed of an intention the agent can perform some action in the environment, send messages to other agents, update its beliefs, adopt or drop goals, or execute any other internal action. When a deed is executed, the PI can also produce

events. For example, when an agent adopts a new goal, an event related to it is produced and added in the *Events* data set. Finally, the intention is updated and placed at the end of the *Intentions* queue for the execution of the remaining deeds. Algorithm 12 presents a pseudo-code for the function PI while Algorithm 13 presents a pseudo-code executed by AT. In order to reduce contention when a UE is selecting an agent for executing the act stage, the act stage is executed $nCyclesAct$ times before to enqueue the agent again in the act queue. The execution of the act stage $nCyclesAct$ times also allows that *several* deeds from different intentions be executed by the agent before to restarts the execution of the reasoning cycle.

```
Procedure PI(intention)
    deed ← intention.top.pop
    if deed.kind = EXTERNAL_ACTION then
        act(deed.parameter)
    else
        if deed.operation = suspend then
            result ← suspend_intention(get_intention(deed.parameter))
        else if deed.operation = resume then
            result ← resume_intention(get_intention(deed.parameter))
        else if deed.operation = add_belief then
            result ← IHF(deed.parameter, belief_inclusion)
        else if deed.operation = remove_belief then
            result ← IHF(deed.parameter, belief_removal)
        else if deed.operation = add_plan then
            result ← PlanLibrary.add(PlanLibrary.get(deed.parameter))
        else if deed.operation = remove_plan then
            result ←
            PlanLibrary.remove(PlanLibrary.get(deed.parameter))
        else if ... then
            {...}
        else
            result ← execute_internal_action(deed)
    if result = SUCCESS then
        EI(intention)
    else
        EE(intention, intention_failed)
```

Algorithm 12: Code for the function PI.

```
Procedure act()
    i ← 0
    while i < nCyclesAct and Intentions ≠ ∅ do
        intention ← Intentions.dequeue()
        if intention ≠ NULL then
            PI(intention)
        i ← i + 1
```

Algorithm 13: Act.

4.3 INTENTION LEVEL

In order to improve the level of concurrency when executing intentions, besides to allow that intentions are carried on concurrently (as presented in Sec. 4.2.2.3), we handle conflicting intentions (Sec. 4.3.1) and allow that plan bodies of the same intention have their executions carried-on concurrently by means of fork and join mechanisms (Sec. 4.3.2).

4.3.1 Handling Conflicting Intentions

As discussed in Sec. 3.3.1, when proposing a technique to handle conflicting intentions, the first point that we need to decide is in which conceptual level the relation of conflict is considered. In our approach, we consider the conflicts in the plan level, once plans are used to achieve the agent goals and different plans can be used to achieve the same goal (e.g., a plan that uses a critical resource vs others that do not), thus allowing that alternative plans be chosen in the case of conflict. While the use of goals would prevent that alternative plans be chosen to achieve the goal, the conflict in the level of parts of plans would spread conflicting controls inside the plan body, thus harming concern separation. If a developer desires to consider the conflicts in the level of part of a plan p, like in (COLLIER; RUSSELL; LILLIS, 2015a), we propose to break down the plan p in several smaller sub-plans and specify the conflict with the specific sub-plan that corresponds to the part of the plan p where the conflict should happen. For example, if a plan $p1$ is composed of a sequence of actions $a1$, $a2$, $a3$, $a4$ and only the action $a4$ is critical and conflicting with another plan (e.g., $p2$), then one could write a new plan $p3$ composed only of the action $a4$ and with the conflict specified with $p2$. Plan $p1$ is now composed of the sequence of actions $a1$, $a2$, $a3$, *call* $p3$, where $p3$ is a sub-plan of $p1$.

Plan	Conflict set
$p1$	$\{p1, p2, p3, ..., pN\}$
$p2$	$\{p1, p2, p3, ..., pN\}$
$p3$	$\{p1, p2, p3, ..., pN\}$
...	$\{p1, p2, p3, ..., pN\}$
pN	$\{p1, p2, p3, ..., pN\}$

Table 7 – Example of CSs of plans.

The second point is the detection of conflicts. In our approach, the detection of conflicts is performed based on explicitly informing the conflicting plans in the agent program. The reason for our choice is performance, since the detection by inspecting the plan bodies can bring an extra overhead, as also stated in (RIEMSDIJK; DASTANI; MEYER, 2009). Thus, in our approach, the developer is responsible for explicitly specifying the conflicts. The detection of conflicts considers that each plan has a *conflict set* (CS), where all conflicts related to the plan are stored. The relation among conflicting plans is symmetric. If one specifies that plan α conflicts with plan β, then β also conflicts with α. The elements contained in the CS to specify conflicts are the plan names. Table 7 illustrates an example of CSs from plans $p1$ until plan pN, where all plans conflict with one another and each plan conflicts with itself.

To help the conflict detection, each agent keeps the information of which plans are being executed by which intended means of which intention. We use the notation $i[im]$ to write an intended means of an intention, where i is the intention and im is the intended means of intention i, and we use the notation $im.i$ to get the intention related to an intended means. The intended means that execute a plan are stored in a set of instantiated intended means (II), which is an attribute of each plan p. Thus, every time that an intention starts to execute a plan, the intended means related to the plan is added in the set of instantiated intended means of the plan. Table 8 presents an example of a *snapshot* of execution. A conflict can be easily detected by checking the relations among plans and intended means. While $p1$ is already being executed by the intended means $i_1[im_1]$, there is no other plan being executed at the moment. Based on the table, any plan that does not conflict with $p1$ can be instantiated and executed concurrently with i_1. In our proposal, we consider that conflicts only happen among different intentions, thus if two plans are instantiated in the stack of the same intention (i.e., they

Plan	$p1$	$p2$	$p3$	...	pN
Intended means	$\{i_1[im_1]\}$	$\varnothing$	$\varnothing$	...	$\varnothing$

Table 8 – Plans and their intended means.

```
Function conflict(p, i)
    forall c ∈ p.CS do
        forall im ∈ c.II do
            if im.i ≠ i then
                return im
    return NULL
```

Algorithm 14: Check if there is a conflict with plan p at run-time.

are in the stack of intention i), they never conflict with each other.

Algorithm 14 presents the algorithm to detect if a plan p, which is trying to be executed by intention i, conflicts with any intention.[1] The function checks for each conflict c in the CS of plan p ($p.CS$), whether there is already some intended means of c being executed by another intention different than i. The function returns the first conflicting intended means, if a conflict is detected, or *null*, if no conflict is detected for plan p. If no conflict is detected, then the CS of p is considered *satisfied* and p can be instantiated and immediately executed.

Finally, the third point is how to handle the conflicts once they are detected. Our approach supports all ways to handle conflicts presented in Sec. 3.3.1, since they can be useful for different purposes. A plan p is considered atomic if conflicts are specified among plan p with all other plans in the plan library and once an intention starts the execution of the atomic plan, the intention is not interrupted by any other intention until the termination of the intended means related to the atomic plan. Table 9 presents an example where $p1$ is specified as an atomic plan. The support to specify plans as atomic satisfies feature `F` (Sec. 3.4), which aims to support the definition of plans as atomic.

In order to allow that non-conflicting intentions are executed concurrently, then the specific plans that are conflicting among one another need to be indicated. For example, one can specify that all plans to

[1] The computational complexity of the algorithm is $O(CM)$, where C is the size of the CS of the plan p, and M is the number of intended means related to the conflicting plan c.

Plan	Conflict set
$p1$	$\{p1, p2, p3, ..., pN\}$
$p2$	$\{p1\}$
$p3$	$\{p1\}$
...	$\{p1\}$
pN	$\{p1\}$

Table 9 – Specifying $p1$ as an atomic plan.

Plan	Conflict set
$p1$	$\{p3\}$
$p2$	$\{p3\}$
$p3$	$\{p1, p2\}$
...	$\varnothing$
pN	$\varnothing$

Table 10 – Conflicts among goals $g1$ and $g2$, where $p1$ and $p2$ are plans to achieve $g1$ and $p3$ is a plan to achieve $g2$.

achieve the goal $g1$ conflict with all plans to achieve the goal $g2$. Plans that do not achieve the goals $g1$ or $g2$ can be executed concurrently with plans that achieve the goals $g1$ or $g2$. Table 10 presents an example where $p1$ and $p2$ are plans to achieve $g1$, $p3$ is a plan to achieve $g2$, and all plans to achieve $g1$ are conflicting with all plans to achieve $g2$. Any plan different than $p1$, $p2$, and $p3$ (e.g., pN) can be executed concurrently even when $p1$, $p2$, or $p3$ are already in execution.

Considering the structure of a plan presented in Sec. 2.1.1, composed of a name, trigger, context, and plan body, our proposal adds a further condition for the plan to be actually executed. While the context of the plan expresses the situation in which a plan is applicable, the conflicts express when the plan can be actually executed (i.e., the plan is *executable*). An agent can choose a plan based on the context of the plan, however, that plan could be still non-executable (i.e., it remains suspended until its conflicting intentions terminate the execution of the intended means that are causing the conflict).

In practice, the BDI model presented in Sec. 2.1.1 is extended with the inclusion of the CS as a part of the plan structure, while the

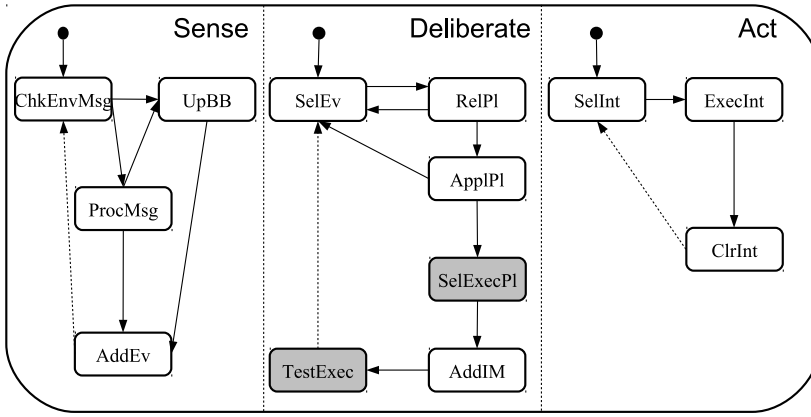

Figure 13 – New BDI agent reasoning cycle.

BDI agent reasoning cycle presented in Fig. 11 is extended as presented in Fig. 13. In the reasoning cycle, the step to select for an applicable plan (`SelAppl`) is modified to select for an *executable* plan (`SelExecPl`), and after the plan be added to an intention (`AddIM`), a verification is done in order to test if the intention remains executable or needs to be suspended (`TestExec`). The characteristics presented in this section and supported in our approach satisfy feature `H` (Sec. 3.4), which aims to detect and avoid the execution of conflicting intentions. While Sec. 4.3.1.1 presents our proposal for selecting an executable plan, Sec. 4.3.1.2 presents our proposal to determine what to do when intentions are suspended, resumed, or dropped.

4.3.1.1 Selection of an Executable Plan

When selecting a plan to achieve a goal, different plans can be considered applicable. While some of these applicable plans conflict with some intended means (i.e., they are not executable), others are non-conflicting (i.e., they are executable). The choice for a non-conflicting plan is a reasonable approach and its main benefit is that an agent can start working towards the goal as soon as the plan is selected. For simplicity, we consider that if there are non-conflicting plans among the applicable plans, the first non-conflicting plan is selected. If there is no executable plan, then three strategies can be conceived. (1) The current event is reintroduced in the events queue for later consideration.

```
Function selectExecutablePlan(P, i)
    forall p ∈ P do
        if conflict(p, i) = NULL then
            return p
    return P.first
```

Algorithm 15: Select an executable plan.

This solution implies that the computation of applicable plans for this event will be redone in next cycles until an executable plan exists. (2) The first applicable plan is immediately chosen for execution and the intention remains suspended until the conflicts are solved. It could be not optimal in the case where conflicts for other applicable plans are solved before this first applicable plan, however this strategy already lets an agent to know how the intention will proceed with its execution and thus the agent can consider it for reasoning about its future decisions. (3) All the applicable plans are stored in a temporary set and every time that an intention terminates to execute an intended means, a new attempt to select an executable plan from that set is performed. If the CS of an applicable plan become satisfied, the plan is selected and the intention can be resumed as soon as possible. One could argue that this approach could lead the agent to select a plan that is actually not applicable anymore. We claim that it is not an issue because all plans in the set were applicable when the event happened (i.e., their contexts were satisfied) and they should be in the set until one of them be effectively selected. In general, such situation happens even without postponing the selection of the executable plan. Due to the dynamics of the agent execution, the context of an applicable plan could become not satisfied as soon as it is chosen. For this thesis, we opted for strategy (2). The algorithm to select an executable plan for intention i is shown in Algorithm 15, where P is a list of applicable plans.[2] Algorithm 15 implements the step `SelExecPl` in Fig. 13.

After selecting an executable plan, the step `addIM` of the reasoning cycle is executed and the selected plan p is finally added on the top of the stack of plans of intention i (Algorithm 16).

[2]The computational complexity of the algorithm is $O(ACM)$, where A is the number of applicable plans, and C and M are defined in *footnote 1*.

Procedure *addIM(p, i)*
  i.push(p)

Algorithm 16: Add the selected plan on the top of the stack of an intention.

Intended Means	Suspended intentions set
$i_1[im_1]$	$[i_2, i_3]$
$i_1[im_2]$	$\varnothing$
$i_1[im_3]$	$\varnothing$
$i_1[im_{...}]$	$\varnothing$
$i_1[im_N]$	$\varnothing$
$i_2[im_1]$	$\varnothing$
$i_3[im_1]$	$\varnothing$
...	$\varnothing$
$i_N[im_1]$	$\varnothing$

Table 11 – Suspended intentions queues, where i_2 and i_3 are suspended due to a conflict with $i_1[im_1]$.

4.3.1.2 Suspending, Resuming, and Terminating Intentions

Besides the operations to suspend and resume intentions, satisfying feature `E` (Sec. 3.4), intentions can be suspended and resumed due to the execution of the conflicting intentions mechanisms. An intention is suspended when the algorithm to select an executable plan cannot find any non-conflicting plan. Thus, the first conflicting plan is selected and the intention is immediately suspended. The suspended intentions due to a conflict are stored in a queue of suspended intentions (SI), which is an attribute of each intended means cim. This queue stores all intentions that were suspended due to a conflict with the intended means cim. Table 11 presents an example of suspended intentions, where i_2 and i_3 are suspended due to a conflict with $i_1[im_1]$. The suspended intentions remain in the queue until the conflicting intention (e.g., i_1) terminates the execution of the intended means that caused their suspension (e.g., im_1).

The suspension of an intention is performed in the step `TestExec` of the reasoning cycle. If the plan on the top of the stack of intention i is conflicting, the intention i is suspended. Algorithm 17 presents the

```
Procedure testExecutableIntention(i)
    cim ← conflict(i.top, i)
    if cim ≠ NULL then
        suspend(i, cim)
```

Algorithm 17: Test an executable intention.

```
Procedure suspend(i, cim)
    I ← I \ {i}
    SI ← SI ∪ {i}
    if cim ≠ NULL then
        cim.SI ← cim.SI ∪ {i}
```

Algorithm 18: Suspend an intention.

algorithm to test if an intention i can proceed or not with its execution.[3]

One important point when suspending an intention is that suspending an intention i does not necessarily imply that intentions that conflict with i can be executed. For example, when an intention is suspended, its suspension may happen at the moment that its execution is between two operations that update the same belief, whose belief may be also updated by the conflicting intentions. Thus, the algorithm to suspend an intention i (Algorithm 18) simply removes i from the intended means set I, adds i in a set of suspended queues SI (a general set that stores all suspended intentions, not only the ones suspended due to a conflict), and if i was suspended due to a conflict, then adds i in the suspended queue of the intended means cim that caused its suspension.[4]

An intention i is resumed when all its conflicting intentions terminate the execution of the intended means that caused the suspension of i. When an intention terminates the execution of an intended means tim, the suspended intentions due to a conflict with tim can be revised in the clear intentions step (`ClrInt`) in order to check if some of them can be resumed (Algorithm 19). For each suspended intention si in the suspended intentions queue of tim, the plan on the top of its stack ($si.top$) is verified to check if its CS is satisfied. The other plans in the stack of si do not need any further verification, once they still keep their

[3]The computational complexity of the algorithm is $O(CM)$, where C and M are defined in *footnote 1*.

[4]The computational complexity of the algorithm is $O(1)$.

```
Procedure terminate(tim)
    I ← I \ {tim}
    forall si ∈ tim.SI do
        cim ← conflict(si.top, si)
        if cim = NULL then
            resume(si, tim)
        else
            suspend(si, cim)
```

Algorithm 19: Terminate an intended means.

```
Procedure resume(si, cim)
    if cim ≠ NULL or conflict(si.top, si) = NULL then
        if cim ≠ NULL then
            cim.SI ← cim.SI \ {si}
        SI ← SI \ {si}
        I ← I ∪ {si}
```

Algorithm 20: Resume an intention.

CS satisfied.[5] As the conflict mechanism only resumes the intentions that were suspended due to a conflict, the intentions suspended by the execution of a suspend operation need to be explicitly resumed by the resume operation. If the CS of *si* is satisfied, then *si* can be resumed, otherwise, *si* is re-suspended due to a new conflict, this time with *cim* ($suspend(si, cim)$).

The algorithm to resume an intention *si* (Algorithm 20) simply removes *si* from the suspended intentions queue of the intended means *cim* that caused its suspension, removes *si* from the set of suspended intentions SI, and adds *si* in the intended means set I. If the resume operation is called when an intended means terminate (when $cim \neq null$), no further verification is necessary to resume the intention once the resume operation is called only when the CS of the suspended intention is satisfied, otherwise, if an agent decides to resume an intention by its other reasons, the function to test conflict is called to check if the intention can really be resumed.[6]

[5]The computational complexity of the algorithm is $O(SCM)$, where S is the number of suspended intentions and C and M are defined in *footnote 1.*

[6]The computational complexity of the algorithm is $O(CM)$, where C and M are defined in *footnote 1.*

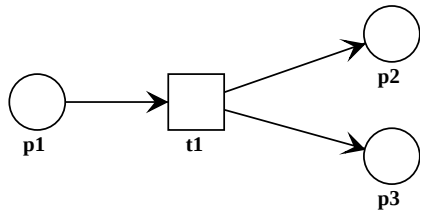

Figure 14 – Fork.

4.3.2 Fork and Join

Another important feature is to avoid that the execution of intentions are blocked due to waiting something to happen, such as a reply message from another agent, when the communication is synchronous[7], and to improve the concurrency when independent course of deeds can be identified in the plan body. Thus, inspired by the feature of fork and join already adopted in some approaches, we also propose a fork and join mechanism to allow that several lines of execution be active for the same intention, which means that an intention can fork and its branches can be executed concurrently. Fork and join are useful in several situations where independent course of deeds of an intention can be identified and executed concurrently and independently. Our proposal for the fork and join satisfies feature `D` (Sec. 3.4).

The operation to fork an intention divides an intention in different branches. We adopt Petri Nets in order to visually represent the specification of plans that use fork and join operations. Each *place* (circles) represents a plan, while *transitions* (squares) are fired when plans terminate. Figure 14 presents an example of Petri Net where a plan $p1$ is forked in two branches ($p2$ and $p3$) that can be executed concurrently.

Two main join operators are proposed: the *join-and* and *join-xor*. They are applied in different situations. The *join-and* means that all branches created by the fork need to terminate to proceed with the execution. A situation where the join-and can be used is when an agent needs to get two information (e.g., temperature of the oven and level of water in a tank) from two different agents and such information are required to proceed with the execution of an intention. The agent can fork the intention in two branches, each one to ask an agent about

[7] A communication is considered synchronous when the agent needs to wait for the reply of the message to proceed with the execution, otherwise the communication is asynchronous.

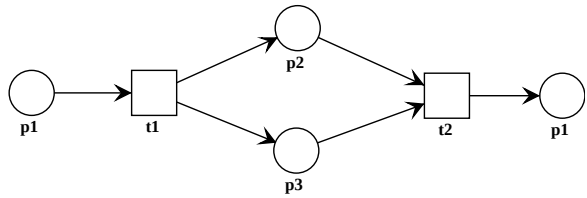

Figure 15 – Fork;Join-And.

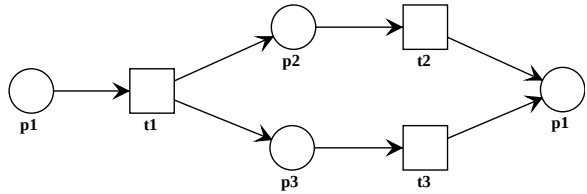

Figure 16 – Fork;Join-Xor.

one information. The agent does not need to wait for the answer from an agent to finally ask the other agent. It can get both information independently on the order that they arrive. Figure 15 presents an example of Petri Net where the join-and is used. In the example, $p1$ is forked in two branches (to execute $p2$ and $p3$ concurrently) and in order to fire the transition $t2$ and continue with the execution of $p1$, both $p2$ and $p3$ need to terminate.

The *join-xor* means that at least one branch created by the fork needs to terminate to proceed with the execution. As soon as one branch terminates, the remaining ones can be aborted. A situation where the join-xor can be used is when an agent needs an information related to the temperature of certain oven to proceed with the execution of an intention. In order to get the temperature, the agent may ask an agent about the temperature or go to the oven and check the temperature by its own. In this case, the intention can be forked in two branches. In the former, the agent can ask another agent about the temperature and waits for the answer. In the latter, the agent can go until the oven and check the temperature. The agent can execute both concurrently and the first one to successfully get the temperature is enough to proceed with the execution of the intention. Figure 16 presents an example of Petri Net where the join-xor is used. In the example, $p1$ is forked in two branches (to execute $p2$ and $p3$ concurrently) and in order to continue with the execution of $p1$, it is enough that either $p2$ or $p3$ terminate.

Both join operations can be combined in order to specify more

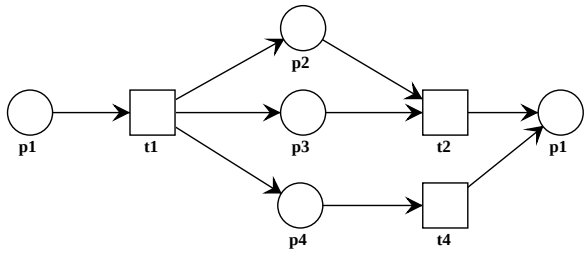

Figure 17 – Fork;Join-And/Xor.

complex situations. Figure 17 presents an example of Petri Net where both the join-and and join-xor are used. $p1$ is forked in three branches (to execute $p2$, $p3$, and $p4$ concurrently). $p1$ only continues its execution when either $p2$ and $p3$ terminate their executions or when $p4$ terminates its execution.

4.4 CONCLUSION

In this chapter, we presented a proposal for a model and architecture for BDI agents and MAS considering concurrency features identified in Sec. 3.4. Thus, the result is a combination of different agent models and architectures, from traditional BDI models until parallel BDI models. Although the scope of the thesis is agents following the BDI model, our BDI agent and MAS model and architecture and their concurrency features can be reused in platforms that adopt other agent models. For example, the features related to the MAS level can be mapped onto other agent models. This is possible because, in the MAS level, agents are executed without considering their internal structure (i.e., the agent internal components, algorithms, and other elements). Likewise, the agent level features used in our agent model and architecture can be reused, e.g., to manage the execution of the agent reasoning cycle, where different stages can be identified and their execution can be carried-on concurrently or sequentially, and multiple times (by means of the number of cycles).

The BDI agent and MAS architecture were integrated into the Jason platform, resulting in our extension named Jason(P). Details of the integration are presented in Appendix C. Besides the concurrent features already supported by Jason, as pointed in Table 4, our extensions enabled support for other important concurrent features. A summary of the new integrated features supported in Jason(P) is presented below:

- Support for executing the reasoning cycle asynchronously;
- Support for agents to use more than one thread;
- Support for different ways to execute the reasoning cycle synchronously, such as by executing the three reasoning cycle stages in different moments;
- Support for specifying the number of times that the reasoning cycle or its stages are executed everytime that a thread selects an agent for execution;
- Support for executing *several* deeds from different intentions in the same reasoning cycle execution;
- Support for processing *several* events in the same reasoning cycle execution;
- Support for defining conflicting plans, thus providing a more flexible mean to handle conflicting intentions, besides defining plans as atomic;
- Support for join-fork in intentions.

Feature	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Jason	✓				✓	✓						✓	✓	
Jason(P)	✓			✓	✓	✓		✓	✓			✓	✓	✓

Table 12 – Supported features.

Strictly considering the state-of-the-art and their supported concurrency features presented in Sec. 3.4, we make a comparison of the supported features of Jason and Jason(P) (Table 12). The symbol ✓ indicates that the feature is satisfied. The black ✓ indicates a feature already supported by Jason and the blue ✓ indicates a new feature integrated on Jason(P). Of course, the concurrency features supported by Jason(P) are not limited to those presented in the state-of-the-art. For example, the specification of the number of cycles and its different parameters (e.g., specify the number of cycles for each reasoning cycle stage) are not supported by any of the works analyzed in the state-of-the-art. In addition, the capability to configure the execution of a MAS is not supported by most of works from the state-of-the-art. Jason(P) allows to choose between executing the reasoning cycle synchronously or

asynchronously, while other works adopt either the synchronous execution or asynchronous execution in their execution platforms. Likewise, except by Jason, other works support either the execution of each agent by their own UE or the use of a pool to execute all agents from the MAS. Indeed, the aim of most works analyzed in Chapter 3 is different than ours.

In the next chapter, we evaluate our proposal by means of a theoretical analysis and performance of experiments, thus identifying the benefits or drawbacks of the integrated features.

5 EVALUATION

While the use of some features and configurations have more obvious benefits or drawbacks, other features and configurations need a practical analysis by means of experiments to identify their benefits or drawbacks. For example, the benefits of a more fine-grained configuration for conflicting intentions over to simply defining plans as atomic are quite easy to identify. Instead, the benefits of the asynchronous execution of the reasoning cycle over the synchronous execution is not that easy to identify. Thus, the focus of this chapter is on answering some questions:

Question 1: Is the use of a thread pool to execute all agents of the MAS always better (in terms of response time or MAS execution time) than giving a single thread to each agent?

Question 2: Is the use of a single thread to execute each agent always better (in terms of response time or MAS execution time) than using a thread pool to execute all agents of the MAS?

Question 3: Is the asynchronous execution always better (in terms of response time or MAS execution time) than the synchronous execution?

Question 4: Does changing the number of cycles for executing the reasoning cycle or its stages improve the execution of an agent?

Question 5: Does the number of cycles harm the fairness when executing an MAS?

Question 6: Which are benefits of the fork and join feature in the development phase?

We perform the experiments using a computer with low computational power and also a more powerful computer, as the aim of our proposal is to exploit the benefits of concurrency independent on the infrastructure of the hardware where the application is executed. In addition, once the focus of the experiments is on to investigate the importance of supporting certain concurrency features and configurations, we do not compare the modified version of the Jason platform with other platforms. The implementation on the platform can strongly interfere on the results.

The first part of the evaluation aims to answer **Question 1**, **Question 2**, and **Question 3** (Sec. 5.1). The second part of the evaluation aims to answer **Question 4** and **Question 5** (Sec. 5.2). The third part of the evaluation aims to answer **Question 6** (Sec. 5.3). The conclusions of this chapter are presented in Sec. 5.4.

5.1 THREAD CONFIGURATIONS AND REASONING CYCLE EXECUTION MODEL

The aim of this section is to investigate the benefits and drawbacks of some features to distribute the threads among the agents and how to execute the reasoning cycle, thus answering **Question 1**, **Question 2**, and **Question 3**. For answering these questions, we performed the evaluation according to five characteristics of an MAS:

- *computation load*. While a light computation load means that an agent has a soft task to perform (e.g., to factor a 3 digit number), an heavy computation load means that an agent has a hard task to perform (e.g., to factor a 100 digit number);
- *intention load*. While an agent with few intentions has a light intention load, an agent with many intentions has a heavy intention load;
- *perception load*. An agent that receives few percepts from the environment has a light perception load, while an agent that receives many percepts from the environment has a heavy perception load;
- *communication load*. While few message exchanges mean a light communication load, a high number of message exchanges means a heavy communication load;
- *MAS population*. A low populated MAS means that the MAS is composed of few agents and a high populated MAS means that the MAS is composed of many agents.

We evaluate three different concurrency configurations (*C1*, *C2*, *C3*). The two first configurations are already supported in the traditional Jason execution platform, while the support for the third configuration is implemented in the Jason(P) execution platform. In *C1*, agents execute a *synchronous* reasoning cycle and each agent has its own thread. In *C2*, agents also execute a *synchronous* reasoning cycle,

however all agents share the same thread pool. In *C3*, agents execute an *asynchronous* reasoning cycle where each component is executed by its own thread. Each thread in *C3* executes the same components of all agents (i.e., there is only one thread to execute the sense of all agents). The comparison of *C1* with *C2* allows to answer **Question 1** and **Question 2**, and the comparison of *C1* and *C2* with *C3* allows to answer **Question 3**.

The evaluation is done by means of experiments, which consist on the implementation of very simple and small scenarios, each one focused on some of the aforementioned MAS characteristics. We start the experiments by considering some scenarios where the *computation load* (Sec. 5.1.1) and the *intention load* (Sec. 5.1.2) are evaluated. Then, agents are stressed according to the *perception load* (Sec. 5.1.3). Finally, agents are evaluated considering different *communication loads* (Sec. 5.1.4). The *MAS population* is evaluated through changing the number of agents in some scenarios. The experiments for this evaluation were performed on a computer Intel(R) Core(TM) 2 Duo @ 2.0GHz (2 CPU cores), 4 GB DDR2, running Linux version 3.6.3-1.fc17.x86_64 and Java version 1.7.0_17. [1] The results and source codes of the experiments can be found at `https://sourceforge.net/p/mrzatelli/code/HEAD/tree/trunk/2015/Experiment3/`.

5.1.1 Computation Load

In this experiment, two simple applications are implemented aiming to give a heavy computation for the agents. In the first one, agents need to print the multiplication table from 1 to k without creating any sub-goal, thus, once instantiated the intention, the only stage of the reasoning cycle that should be executed is the act stage. The implementation is done by means of a nestled loop (Fig. 20).

In order to evaluate the computation load, we fixed the number of agents in *one* agent and vary k from 100 to 3000. The results depicted in Fig. 18 (left) show that adopting a synchronous reasoning cycle with one thread per agent (*C1*) has the fastest response time[2], while adopting a synchronous reasoning cycle with a single thread pool (*C2*) has the worst response time. These results are explained due to the overhead caused by using thread pools (*C2* and *C3*) to execute a

[1] Experiment parameters are dimensioned according to the available computer hardware.

[2] The response time is the elapse time for an agent to complete the execution of an intention.

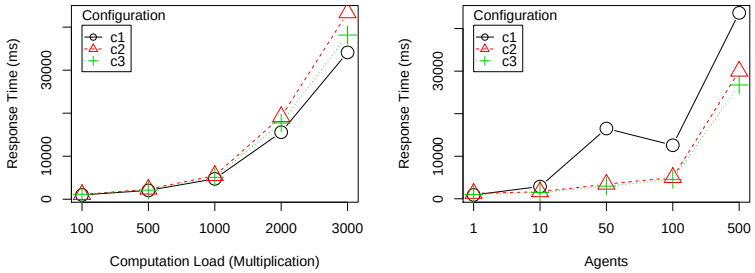

Figure 18 – Multiplication table: computation load (left) and MAS population (right).

```
1. +?fib(0,0).
2. +?fib(1,1).
3. +?fib(K,X) <-
4.     ?fib(K-1,A);
5.     ?fib(K-2,B);
6.     X = A+B.
```

Figure 19 – Fibonacci Numbers.

single agent. Moreover, the asynchronous reasoning cycle (*C3*) showed a faster response time than *C2*. The reason for this behavior is that while the full reasoning cycle is executed in *C2*, the only reasoning cycle stage that remains active in *C3* is the act stage.

In order to evaluate the MAS population, we vary the number of agents from 1 to 500 and fix $k = 100$. The results depicted in Fig. 18 (right) show that adopting an asynchronous reasoning cycle (*C3*) has the fastest response time, while adopting a synchronous reasoning cycle with one thread per agent (*C1*) has the worst response time. These results are explained due to the context-switch overhead caused by the high number of threads in *C1*. Moreover, as also stated before, *C3* has a faster response time than *C2* due to only executing the act stage.

In the second application, agents need to compute the first k Fibonacci numbers (Fig. 19). The implementation of the plan to compute the first k Fibonacci numbers follows a recursive approach, where each recursive call is a *sub-goal* (lines 4 and 5), which forces the execution of the deliberate stage of the reasoning cycle because the agent needs to select a plan to handle the adoption of the sub-goal.

In order to evaluate the computation load, we fix the number of

```
1. +!work(K) <-
2.     for (.range(X,1,K)) {
3.         for (.range(Y,1,K)) {
4.             .print(X * Y);
5.         };
6.     }.
```

Figure 20 – Multiplication Table.

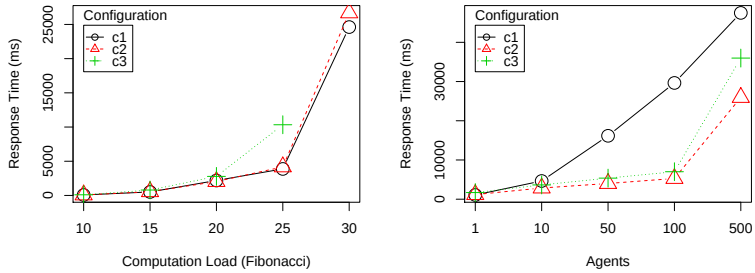

Figure 21 – Fibonacci: computation load (left) and MAS population (right).

agents in *one* agent and vary k from 10 to 30. The results depicted in Fig. 21 (left)[3] show that as soon as the computation load increases, adopting a synchronous reasoning cycle with one thread per agent (*C1*) has the fastest response time, while adopting an asynchronous execution for the reasoning cycle (*C3*) has the worst response time. These results are explained due to the overhead caused by using thread pools (*C2* and *C3*) to execute a single agent. Moreover, the concurrency control access mechanism necessary in the asynchronous reasoning cycle (*C3*) demonstrated to have a very high overhead in this scenario, where both deliberate and act stages needs to be executed constantly.

The MAS population is evaluated in the same way as in the multiplication table scenario, however we fix $k = 17$. The results depicted in Fig. 21 (right) show that adopting a synchronous reasoning cycle with a single thread pool (*C2*) has the fastest response time, while adopting a synchronous reasoning cycle with one thread per agent (*C1*) has the worst response time. Moreover, adopting an asynchronous reasoning cy-

[3]The response time for $fib(30)$ adopting the asynchronous reasoning cycle (*C3*) was 204117ms, however we omitted from the graphic to let it readable to compare the other two configurations.

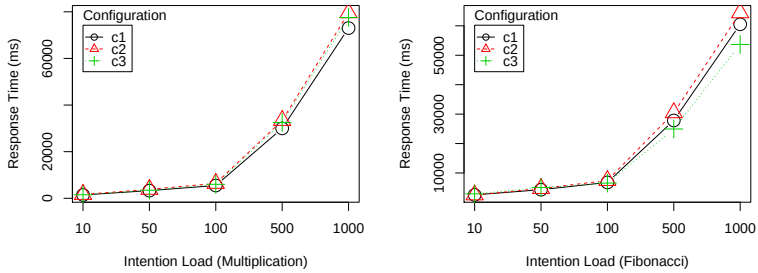

Figure 22 – Multiplication table (left) and Fibonacci (right): intention load.

cle (*C3*) has a worse response time than adopting *C2*. While the worst response time for *C1* is explained due to the high context-switch overhead caused by the high number of threads, the worse response time for *C3* compared to *C2* is caused by the overhead to handle the concurrent execution of the deliberate and act stages.

5.1.2 Intention Load

The intention load is evaluated based on the Fibonacci and the multiplication table scenarios by means of varying the number of intentions. Only *one* agent is used in the execution, but instead of only computing a single Fibonacci or printing a single multiplication table, the agent has from 10 to 1000 intentions (to compute Fibonacci numbers or to print the multiplication tables) being executed concurrently. In the Fibonacci scenario, we fixed $k = 17$, while in the multiplication table scenario, we fixed $k = 100$. The results are depicted in Fig. 22. While adopting a synchronous reasoning cycle with one thread per agent (*C1*) showed the fastest response time in the multiplication table scenario, the fastest response time in the Fibonacci scenario happens when an asynchronous reasoning cycle (*C3*) is adopted. This result is explained because in the Fibonacci scenario, while one thread is deliberating about the sub-goals that are being produced by the act stage, another thread is executing intentions. Such advantage of the asynchronous reasoning cycle does not appear in the multiplication scenario because the only active reasoning cycle stage is the act stage. The overhead caused by the thread pool in the act stage of the asynchronous reasoning cycle is

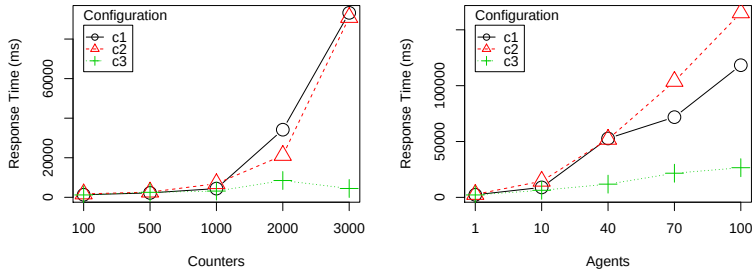

Figure 23 – Counting: perception load (left) and MAS population (right).

higher than the overhead of executing the sense and deliberate stages in the synchronous reasoning cycle with one thread per agent. Finally, the synchronous reasoning cycle with a single thread pool (*C2*) has the worst response time in both scenarios because of both the overhead of the thread pool and the overhead of executing the full reasoning cycle.

5.1.3 Perception Load

In this third experiment, we extend the multiplication table scenario (Fig. 20) to measure the agent reactivity, which consists on evaluating how long an agent takes to print the multiplication table from 1 to 100, while the environment is producing new percepts constantly. The environment basically consists on a set of counters that are updated everytime that an agent performs an `inc` action, thus producing percepts for agents that are observing the environment. All produced percepts are perceived by an agent in a *single shot*, which means that the state of the environment is given by the current value of all counters in a certain moment (i.e., all counters need to have the same value since they are updated in the same operation). Thus, the belief update only finishes when all percepts have been processed. While one agent is responsible for performing the `inc` action, others simply observe the environment. The execution finishes when all agents finish to print the multiplication table.

In order to evaluate the perception load, we fix the number of agents in *one* agent and vary the number of counters from 100 to 3000. The results depicted in Fig. 23 (left) show that adopting an asyn-

chronous reasoning cycle has the fastest response time, while adopting a synchronous reasoning cycle, with one thread per agent (*C1*) or with a single thread pool (*C2*), have the worst response times. This result is important to highlight one benefit of adopting an asynchronous reasoning cycle. In the asynchronous execution, the agent can execute the act stage more than once before the sense or deliberate stages enter in a critical section (e.g., the deliberate stage starts to execute the belief update function). Thus, the act stage can be executed concurrently until it gets blocked because the concurrent access to some critical section. Moreover, the main reason for the worst response times for *C1* and *C2* is due to the execution of only *one* deed in each reasoning cycle. Thus, everytime that the cycle restarts, the agent needs to sense the environment and update the belief base, which has a huge computational cost as soon as the number of counters increases.

In order to see the impact of the MAS population, we vary the number of agents from 1 to 100. The results depicted in Fig. 23 (right) show that adopting an asynchronous reasoning cycle (*C3*) has the fastest response time, while adopting a synchronous reasoning cycle with a single thread pool (*C2*) has the worst response time. The fastest response time for *C3* is again explained because the act stage can be executed concurrently while it does not get blocked due to the concurrent access to a shared structure with the sense or the deliberate stages.

5.1.4 Communication Load

The communication load is evaluated by means of a token-ring scenario. We implement a variation of the token-ring presented in (CARDOSO et al., 2013). The ring is made by means of linking each agent to another agent in a circular form. Each agent needs to pass the received tokens to its neighboring agent and each token needs to pass by each agent only once. The number of tokens in the MAS vary from 1 to t, in order to change the number of message exchanges. The initial configuration of the tokens is given by the formula: $a = i\ n/t$, where a is the current agent that will receive the token, i is the identifier of the current token that will be given for the agent, n is the number of agents in the ring (fixed in 500), and t is the number of tokens that need to be given for the agents. The execution finishes when all tokens have been passed by all agents in the ring.

The results depicted in Fig. 24 show that adopting a synchronous reasoning cycle with a single thread pool (*C2*) has the fastest execution

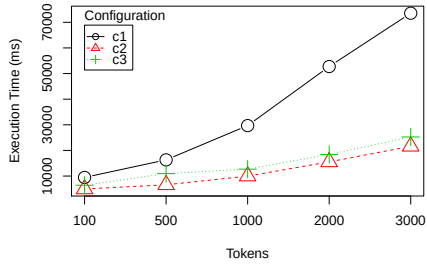

Figure 24 – Token ring: communication load.

time, while adopting a synchronous reasoning cycle with one thread per agent (*C1*) has the worst execution time. The faster execution time for *C2* compared to *C3* is explained because all stages of the reasoning cycle need to be executed for each received message. The agent receives messages (they are processed in the sense stage), selects a plan to forward the token (it is performed in the deliberate stage), and executes the intention to forward the token (it is performed in the act stage). In this scenario, all these activities have more or less the same computational cost. Thus, the asynchronous execution does not provide any advantage, because the overheads caused by the use of multiple thread pools (one for each reasoning cycle stage) and the concurrency control access mechanisms have a high impact on the execution. Finally, the reason for *C1* has the worst execution time is again due to the context-switch overhead caused by the high number of threads.

5.2 THE NUMBER OF CYCLES

The focus of this section is on investigating the impact of the number of times that the agent reasoning cycle or its stages are executed when the agent is selected by a thread from a pool, thus answering **Question 4** and **Question 5**. For simplicity, we are going to call *number of cycles* to refer to both the number of times that the agent reasoning cycle is executed and the number of times that each stage of the reasoning cycle is executed. The variable n is used in the sequence to define the *maximum* value for the number of cycles, as the actual number that will be executed depends on the agents activities. For example, if an agent has only four events to deliberate in a certain

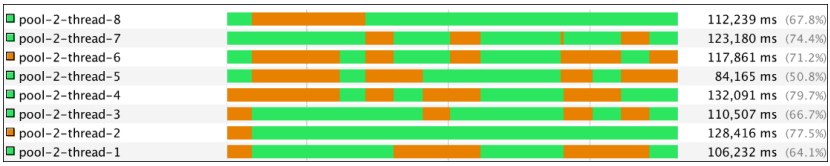

Figure 25 – Adopting thread pools to execute agents ($n = 1$).

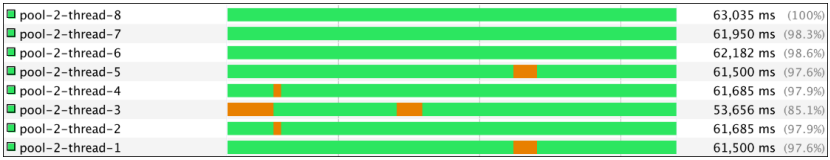

Figure 26 – Adopting thread pools to execute agents ($n = 15$).

moment, it will only execute the deliberate stage four times before to enqueue the agent again in the deliberate queue even if $n = 15$. In addition, the variable a is used in the sequence to refer to the number of agents in the MAS.

In order to test the efficiency of using thread pools with $n = 1$, a simple and preliminary experiment, considering the Fibonacci scenario (Fig. 19), was performed to run a MAS composed of 10,000 agents on an 8 CPU cores computer. The results of how the execution proceeds is depicted in Fig. 25. The green color means that the threads are executing and the orange color means that threads are blocked waiting to access the queue of agents. We observed that threads got blocked several times during the execution. The threads are executing from 50.8% until 79.7% of the time, considering the interval of time that the MAS was running. The reason for this behavior are the jobs that a thread need to perform, which are quite light (e.g., execute only *one* reasoning cycle). The combination of these light jobs and more computer cores implied in a high contention for the threads to select agents from the thread pool queue, resulting in several threads being blocked at the same time. This is not only an issue strictly related to the Jason execution platform, but to any kind of application that adopts thread pools to execute light jobs in multi-core computers.

As soon as we replicate the experiment considering $n = 15$, we observed that most threads were executing more than 97% of the time (Fig. 26). The computer cores are clearly being exploited more efficiently than when $n = 1$. In the remaining of this section we investigate what are the main implications of this result in the MAS context, when

executing an application.

The scenarios of the experiments in this section are the same as in Sec. 5.1. The scenarios have different demands for each stage of the reasoning cycle. This is especially useful when the execution is asynchronous. Thus, while only one stage is stressed in a first scenario, all the three stages are equally stressed in the last scenario. The chosen scenarios are the multiplication table (Sec. 5.2.1), Fibonacci (Sec. 5.2.2), and token-ring (Sec. 5.2.3).

The experiments consider eight different configurations. In the first four configurations, the agent reasoning cycle is executed *synchronously* and all agents are executed by the threads from the same thread pool. In the last four configurations, the reasoning cycle is executed *asynchronously* and each stage is executed by the threads from a different thread pool, thus three different thread pools are used to execute all the agents in the MAS. The difference among the configurations inside these two groups is the value of n, which can be between 1 and 15. The experiments performed in this section were done on a computer Intel(R) Xeon(R) CPU X7350 2.93GHz (8 CPU cores), 16 GB, and Java (openjdk) version 1.8.0_66.

5.2.1 Multiplication Table

In this first scenario, the aim is to stress only the act stage. Agents need to print the multiplication table from 1 to k without creating any sub-goal, thus, once instantiated the intention to print the multiplication table, the only stage that should be executed is the act stage. The implementation is done by means of a nestled loop (Fig. 20). a varies from 100 to $10,000$ and the results consist in the response time that each agent takes to print the multiplication table from 1 to k, where k is fixed in 200 for this experiment. Each agent only instantiates *one* intention during its execution.

The results depicted in Fig. 27 show a huge improvement when n is greater than 1. The benefits can be observed in both synchronous and asynchronous executions. The improvement is justified because of the high overhead caused by threads competing to select agents from the queues to execute their reasoning cycle (or each stage) when $n = 1$. The execution time of a single cycle is short enough that several threads can be trying to access the queue at the same time, which results in some threads being waiting to select an agent.

In general, changing n from 1 to 15 improved the response time

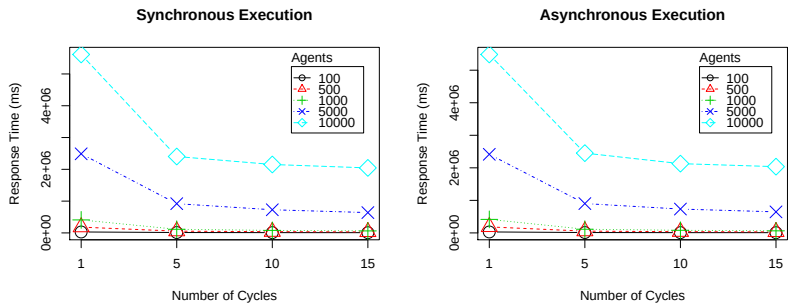

Figure 27 – Multiplication Table: impact of number of cycles.

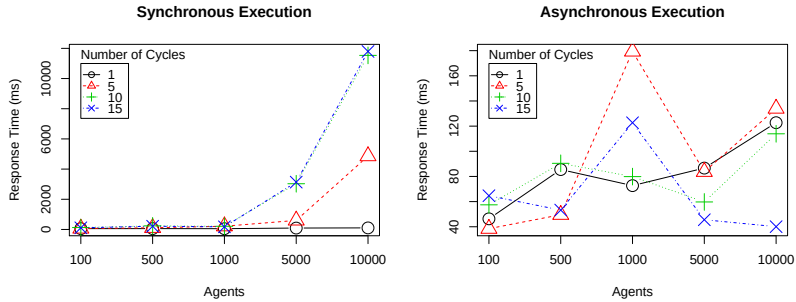

Figure 28 – Multiplication Table: impact of number of agents (standard deviation).

between 63.57% and 84.61% for the synchronous execution, and between 62.87% to 84.63% when the execution is asynchronous, which can be observed in Table 13. This table presents the values that were used to plot the graphic from Fig. 13 and also brings other important results, such as the increasing gain when n varies from 1 to 15. For example, for $a = 100$ and synchronous execution, the response times are 29,754.21ms (for $n = 1$), 12,175.14ms (for $n = 5$), 9,173.78ms (for $n = 10$), and 7,675.08ms (for $n = 15$). The same pattern repeats for the asynchronous execution and for all values of a used in this scenario.

Another indicator that can be analyzed is the standard deviation (Fig. 28), in order to measure how different is the individual response time compared to the mean. It is desirable that agents have a close response time, keeping the *fairness* in the MAS execution. While changing

Cycles/Agents	100	500	1,000	5,000	10,000
Synchronous Execution					
1	29754.21	182068.28	408869.14	2487819.40	5612606.00
5	12175.14	54925.18	114363.59	919623.40	2403035.00
10	9173.78	38933.62	79460.37	728171.50	2152043.00
15	7675.08	31569.64	62909.93	639339.30	2044527.00
Perc. Difference (Between 1 and 15)	**74.20%**	**82.66%**	**84.61%**	**74.30%**	**63.57%**
Asynchronous Execution					
1	29952.00	184701.44	415722.58	2415177.70	5483537.00
5	11915.26	54834.71	113379.28	901470.10	2451763.00
10	8819.44	38163.80	80715.91	733326.80	2128682.00
15	7699.33	31239.56	63885.07	646776.20	2035854.00
Perc. Difference (Between 1 and 15)	**74.29%**	**83.09%**	**84.63%**	**73.22%**	**62.87%**

Table 13 – Response time multiplication table.

n in the asynchronous execution does not have any noticeable impact on the standard deviation, increasing n in the synchronous execution shows a higher (and increasing) impact on the standard deviation. However, such impact is also not so significant if compared to the mean. For example, considering a standard deviation of 12,000ms (for $a = 10,000$ and $n = 15$), it only represents 0.59% of the mean (2,044,527.00ms).

5.2.2 Fibonacci

In this second scenario, the aim is to stress the deliberate and act stages. To do that, agents are implemented to compute the first k Fibonacci numbers (Fig. 19). a varies from 100 to $10,000$ and the results consist in the response time that each agent takes to compute the first k Fibonacci numbers, where k is fixed in 20 for this experiment. Each agent only instantiates *one* intention during the whole execution.

The results depicted in Fig. 29 show an improvement of the response time when n is greater than 1 in both execution forms. As in the Multiplication Table scenario, these results are explained because threads do not *clash* so much when accessing the queues and the overhead by enqueueing and dequeueing the agents from the queues is reduced. While the improvements of defining n greater than 1 considering the synchronous execution are similar in both scenarios (Fibonacci and Multiplication Table), the improvement in the Fibonacci scenario

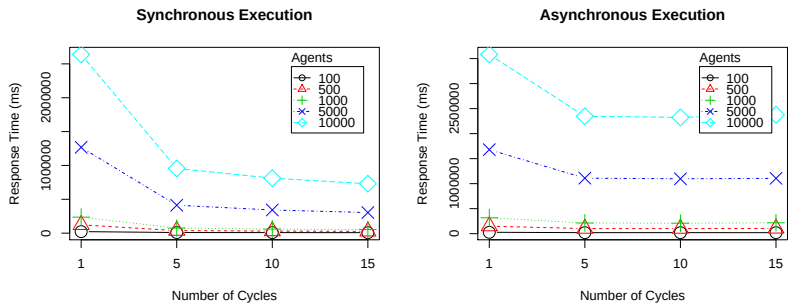

Figure 29 – Fibonacci: impact of number of cycles.

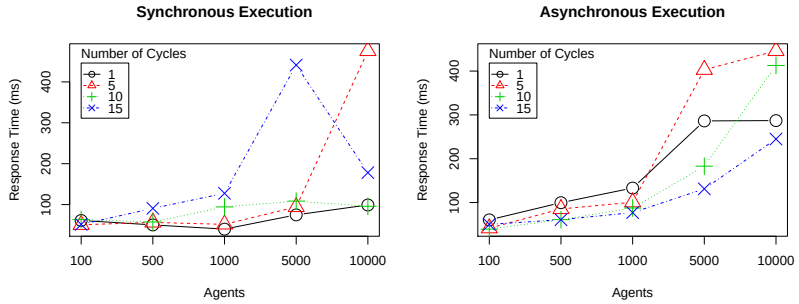

Figure 30 – Fibonacci: impact of number of agents (standard deviation).

considering the asynchronous execution is not as good as in the Multiplication Table scenario. This fact is due to the internal competition of threads from the deliberate and act components concurrently accessing common data structures like the intentions queue, which does not happen in the Multiplication Table scenario.

In general, changing n from 1 to 15 improved the response time between 68.16% and 78.41% for the synchronous execution, and between 27.73% and 34.16% when the execution is asynchronous, which can be observed in Table 14. This table presents the values that were used to plot the graphic from Fig. 29. Moreover, while adopting a synchronous execution and varying n from 1 to 15 shows an increasing improvement on the response time, the same cannot be said about adopting an asynchronous execution. For example, when adopting a synchronous execution and $a = 10,000$, the response times are 2,639,022.50ms (for $n = 1$),

956,510.80ms (for $n = 5$), 811,776.00ms (for $n = 10$), and 729,862.90ms (for $n = 15$). This pattern repeats for all values of a used in this scenario. However, when adopting an asynchronous execution and $a = 10,000$, the response times are 3,582,418.00ms (for $n = 1$), 2,346,757.00ms (for $n = 5$), 2,324,632.00ms (for $n = 10$), and 2,377,475.00ms (for $n = 15$), which means that the fastest response time appears when $n = 10$, instead of 15. Thus, there is not any clear impact on the response time of this scenario when $n = 5$, 10, or 15, differently than when $n = 1$ (the drawback of defining $n = 1$ is clear). This is caused because each agent has only *one* intention and the execution of the act stage is interrupted everytime that a sub-goal is adopted until the deliberate stage be executed to handle the adoption of the sub-goal. It means that the number of times that the act stage is actually executed is limited until a sub-goal is adopted, which according to the implementation in Fig. 19 is certainly lower than 10 times.

The analysis of the standard deviation (Fig. 30) for this scenario also shows that n in the asynchronous execution does not have any impact on the standard deviation, while the impact of n in the synchronous execution continues being irrelevant (this time being around 500ms, which means 0.16% of the mean (304,210.20ms), considering the worst case with $a = 5,000$ and $n = 15$).

5.2.3 Token Ring

In this third scenario, we also evaluate a variation of the token-ring benchmark presented in (CARDOSO et al., 2013). The aim of this scenario is to stress all the stages equally through the use of communication. An agent receives messages (they are processed in the sense stage), selects a plan to handle the message (it is performed in the deliberate stage), and executes the produced intention (it is performed in the act stage).

In order to change the number of message exchanges, the number of tokens vary from 100 to 50,000. At the beginning of the execution, the tokens are distributed among the agents following the formula $x = i\ a/t$, for each token id i (which is a number that varies from 1 to t), where x is the position of the agent in the ring that will receive the token with id i, a is the number of agents in the ring (fixed in 5,000), and t is the number of tokens that need to be given for the agents. The execution finishes when all tokens have been passed through all agents in the ring. In this scenario, we are interested on the overall *execution time* instead

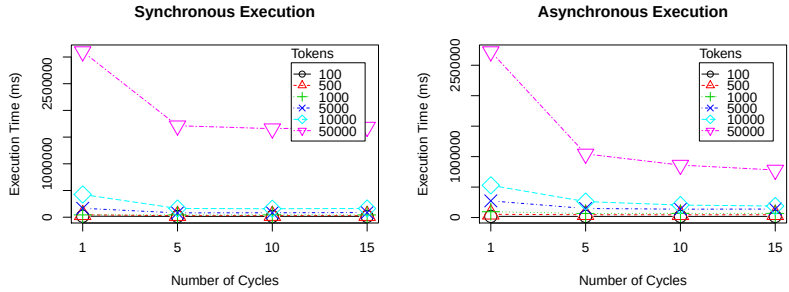

Figure 31 – Token Ring: impact of number of cycles.

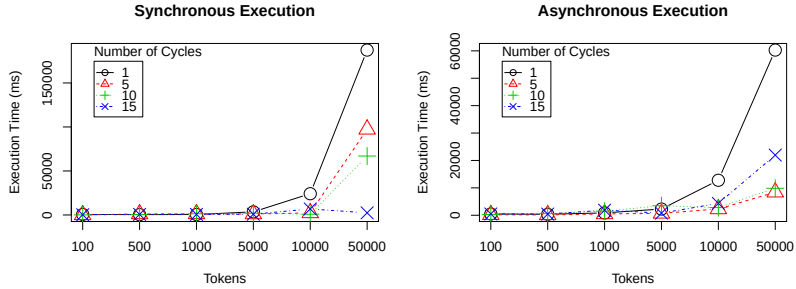

Figure 32 – Token Ring: impact of number of tokens (standard deviation).

of the individual response time. The response time does not make sense because an agent depends on another agent to perform its activities, which causes interferences. The execution time is given by the elapse time between the start and the termination of the MAS execution.

The results depicted in Fig. 31 show an improvement on the execution when n is greater than 1 considering a synchronous execution. A better improvement can be observed when the execution is asynchronous and n is also greater than 1.

In general, changing n from 1 to 15 improved the execution time between 1.40% to 61.68% for the synchronous execution, and between 2.04% to 71.35% when the execution is asynchronous, which can be observed in Table 15. This table presents the values that were used to plot the graphic from Fig. 31. It is also possible to observe that adopting the synchronous execution with $n = 5$, 10, or 15 does not seem to have

a huge impact on the execution time. While defining $n = 1$ has usually the worst execution times (clearly seen when $t = 5,000$ or greater), defining $n = 15$ does not necessarily result in the best execution times. The only scenario where $n = 15$ has the best execution time is when $t = 1,000$, however, it brings only 1.40% of improvement compared to when $n = 1$, which is not significant. Differently from adopting a synchronous execution, an increasing improvement can be noticed when $t = 10,000$ or $50,000$ and the execution is asynchronous. When $t = 5,000$ or lower, the benefits of $n = 5$, 10, or 15 are not relevant. The execution times are similar to one another, like seen in the synchronous execution.

Differently from the Multiplication Table and Fibonacci scenarios, the analysis of the standard deviation (Fig. 32) for this scenario considers the overall execution time. The standard deviation in both executions (synchronous and asynchronous) is higher when $n = 1$.

5.3 FORK AND JOIN

The aim of this section is to identify benefits or drawbacks of the fork and join feature, thus answering **Question 6**. As the improvement on the execution is obvious when using fork and join to exploit concurrency in the intention level, this section focus on evaluating the fork and join in the development phase of an agent. In the chosen scenario, an agent must decrement two different counters (`a` and `b`) in its belief base and then print a message when both counters reach zero. In both applications, the developer writes a plan to execute `!count(a, 10)` and `!count(b, 3)` concurrently and the message "*Finished!*" is printed as soon as both ones (`!count(a, 10)` and `!count(b, 3)`) terminate. Fig. 33 presents the code of the agent considering the version of Jason without the support to fork and join. The plan is written from lines 3 to 7. The developer needs to *explicitly* use the belief base for storing control information (the belief `finishedCounter(Id)`, in line 10) and checks it by means of the internal action `.wait()` to proceed with the execution as soon as the belief base contains both `finishedCounter(a)` and `finishedCounter(b)` (line 6). Fig. 34 presents how a developer would need to write the same plan using a specific fork and join construct. The plan is written from lines 3 to 7. The developer does not need to use the belief base to store control information, neither to explicitly use the internal action `.wait()` to wait for the termination of `!count(a, 10)` and `!count(b, 3)`. Instead, the developer simply uses

```
1. !start.
2.
3. +!start <-
4.    !!count(a, 10);
5.    !!count(b, 3);
6.    .wait(finishedCounter(a) & finishedCounter(b));
7.    .print("Finished!").
8.
9. +!count(Id, 0) <-
10.   +finishedCounter(Id).
11.
12. +!count(Id, X) <-
13.   .print(Id, " ", X);
14.   !count(Id, X-1).
```

Figure 33 – Example of how to implement fork and join without a specific construct.

```
1. !start.
2.
3. +!start <-
4.    {!count(a, 10)}
5.      |&|
6.    {!count(b, 3)};
7.    .print("Finished!").
8.
9. +!count(Id, 0).
10. +!count(Id, X) <-
11.   .print(Id, " ", X);
12.   !count(Id, X-1).
```

Figure 34 – Example of how to implement fork and join with a specific construct.

the fork and join operator **|&|** (lines 4, 5, and 6), which is clearly more abstract than using other ways.

5.4 CONCLUSION

In this chapter, we performed an evaluation of some features and configurations that can be adopted to run an MAS. Five main questions were identified and after performing the experiments to answer such questions, we conclude that the answer for **Question 1** is that the use of a thread pool is not always the best choice for executing an MAS. For example, having each agent being executed by its own thread showed better results when changing the computation load (Sec. 5.1.1) or intention load (Sec. 5.1.2) in a MAS composed of only one agent. However, we can also conclude that the answer for **Question 2** is that

the use of a single thread to execute each agent is not always the best choice for executing an MAS. Considering again the results presented in (Sec. 5.1.1) and (Sec. 5.1.2), the use of a thread pool is the best choice when the MAS population increases.

The answer for **Question 3** is that the asynchronous execution is not always better than the synchronous execution. Under certain circumstances, the results of the synchronous execution are better. For example, while the asynchronous execution got better results in scenarios where some stage of the reasoning cycles is stressed more than others (e.g., in the multiplication table scenario when evaluating different computation loads in Sec. 5.1.1), the synchronous execution got better results in scenarios where the stages of the reasoning cycle are stressed more or less similarly, such as in the token-ring scenario (Sec. 5.1.4).

The answer for **Question 4** is that the number of cycles for executing the reasoning cycle or its stages plays an important role for the agent execution, and choosing the number of cycles can bring improvements for the agent execution. Increasing the number of cycles brought a significant improvement (over 50%) on the response time and on the overall execution time in most of the cases, considering both the synchronous and the asynchronous executions. Finally, the answer for **Question 5** is that the number of cycles, considering the interval used in the experiments, did not have a significant impact on the fairness in the MAS. While the individual response time of the agents were quite similar for the multiplication table (Sec. 5.2.1) and Fibonacci scenarios (Sec. 5.2.2), the overall execution times for the token-ring scenario (Sec. 5.2.3) showed a higher standard deviation when the reasoning cycle (or its stages) is executed *once*.

As the answer for **Question 6**, we identify some benefits of using the fork and join feature, such as the separation of concerns, which means that the belief base is not used to store control information about the concurrent execution of plans. In addition, the execution control of the concurrent plans is managed by the execution platform without the need of the developer to explicitly program it in the agent code.

In the next chapter, we present a discussion around the choices adopted in the thesis and other obtained results.

Cycles/Agents	100	500	1,000	5,000	10,000
Synchronous Execution					
1	22795.32	122628.28	236751.51	1267787.30	2639022.50
5	9262.90	37526.11	74776.56	412214.50	956510.80
10	8310.88	30562.32	60200.44	342472.90	811776.00
15	7258.80	26475.31	52800.37	304210.20	729862.90
Perc. Difference (Between 1 and 15)	**68.16%**	**78.41%**	**77.70%**	**76.00%**	**72.34%**
Asynchronous Execution					
1	27830.78	150062.70	317497.40	1679745.00	3582418.00
5	20402.27	101891.20	213612.40	1109639.00	2346757.00
10	20539.13	101502.50	209065.50	1096760.00	2324632.00
15	20112.18	104625.70	217596.40	1105920.00	2377475.00
Perc. Difference (Between 1 and 15)	**27.73%**	**30.28%**	**31.47%**	**34.16%**	**33.63%**

Table 14 – Response time Fibonacci.

Cycles/Tokens	100	500	1,000	5,000	10,000	50,000
Synchronous Execution						
1	18485.33	44802.00	43000.00	166533.00	425754.30	3102042.00
5	14980.33	28079.33	43154.67	80701.00	163149.00	1713179.00
10	15444.67	28900.33	44241.67	82963.00	158737.70	1658086.00
15	15426.67	28716.00	42397.00	86679.00	163156.30	1688259.00
Perc. Difference (Between 1 and 15)	**16.55%**	**35.90%**	**1.40%**	**47.95%**	**61.68%**	**45.58%**
Asynchronous Execution						
1	19353.33	53431.33	95572.33	273690.70	528796.30	2724604.00
5	18742.67	45801.67	67287.67	148358.00	261844.70	1044252.70
10	18704.67	44703.00	68728.67	137454.30	204724.30	861358.30
15	18958.00	45492.33	68299.00	139403.00	188982.30	780594.70
Perc. Difference (Between 1 and 15)	**2.04%**	**14.85%**	**28.54%**	**49.07%**	**64.26%**	**71.35%**

Table 15 – Execution time token ring.

6 RESULTS AND DISCUSSION

Along the thesis we presented, analyzed, and evaluated several concurrency techniques that can be exploited to take benefits from the hardware infrastructure to execute an MAS. The main results of the thesis are presented by answering the research question: *which concurrency techniques and models can be exploited to develop execution platforms to better take advantage of multi-core architectures and related parallel hardware?* We noticed that each of the concurrent features analyzed in the thesis has its own importance and should be integrated in an execution platform. While a careful choice of a configuration to execute a MAS can bring important benefits for the execution, the choice for a random configuration (i.e., without consider the characteristics of the scenario of the application and the hardware infrastructure) could significantly harm the execution. In order to answer the research question, we highlight the main improvements that were identified along the thesis and which features allow to achieve such improvements. While Table 16 compiles the main features and how they affect the execution of an MAS, each of the remaining paragraphs gives a brief overview of the main improvements according to the features identified in the thesis.

The amount of time that agents are actually being executed can be improved (e.g., the overhead and the contention are reduced) by correctly choosing among three main features. The first feature is to execute each agent by its own UE in MAS composed of few agents (i.e., the number of agents is around the same number of PEs), so that the number of UEs is not causing a significant context-switch overhead. The second feature is the adoption of pools in MAS composed of a higher number of agents (the number of agents is significantly higher than the number of PEs). A pool reduces the overhead caused by context-switches, however it causes contention when UEs try to access the pool queue to select an agent, which results in a lower usage of the CPUs. The contention becomes more significant as soon as the number of PEs or the computational power of these PEs increases. The third feature is the number of cycles, which can be exploited to reduce the contention, making UEs to access the pool queue less times and in a higher time interval (i.e., the time interval between dequeue an agent and enqueue it again is higher). As a consequence, the number of clashes is reduced.

The fairness of the agent execution can be also improved by using pools. The scheduling of agents is controlled by the execution platform and the agents are executed in turns, which is not possible to do when

Feature	How it impacts on MAS execution
1. Number cycles	Improves the CPU usage, the overall performance, and prioritizes agents
1.1 Number cycles in each stage	Prioritizes stages of the reasoning cycle
1.2 Number act cycles	Improves the throughput
2. Asynchronous execution	Improves the response time when parts of the cycle are stressed more than others
3. Synchronous execution	Improves the response time when all parts of the cycle are stressed similarly
4. Pool	Improves the fairness and the overall performance in high populated MAS
5. Threaded agents	Improves the overall performance in low populated MAS
6. Interleaved execution of intentions	Improves the intention level concurrency and the fairness on execution of intentions
7. Conflicting intentions	Improves the intention level concurrency in intentions and the fairness on execution of intentions
8. Fork and join	Improves the intention level concurrency in intentions, minimizes idle times, and provides a more natural and easier way to synchronize the execution of intentions

Table 16 – Summary of features and how they impact on MAS execution.

each agent is executed by its own UE. In this latter case, the execution of the UEs is managed by the operational system and the execution platform does not have control over the scheduling of the UEs (e.g., some agents could start to execute before than others, while UEs are still being instantiated to execute the MAS).

The execution of some particular agents can be prioritized by means of the feature to specify the number of times that the reasoning cycle is executed every time that the UE selects an agent. A higher priority for an agent α can be given by specifying a higher number of times that its reasoning cycle is executed, thus α is executed for a longer time than other agents (i.e., we give more CPU time for agent α). Likewise, the execution of a particular stage of the reasoning cycle can be prioritized by means of the feature to specify the number of times

that a certain reasoning cycle stage is executed every time that the UE selects it.

The response time in highly dynamic environments, especially considering those that produce a lot of percepts, can be improved by means of the feature of asynchronous execution of the reasoning cycle. In this feature, each stage can be executed without waiting for another stage to finish. The sense stage can be executed concurrently with the deliberate and act stages and thus the agent can continue handling its percepts without compromise the execution of its intentions. In addition, while the response time is improved by adopting the asynchronous execution in scenarios where some stage of the reasoning cycle is stressed more than others, the synchronous execution improves the response time in scenarios where the stages are stressed more or less similarly.

Considering that the throughput of an agent is measured by the number of actions executed by the agent in a certain interval of time, it can be improved by a proper setup of the number of times in which the act stage is executed. A higher number of times for the execution of the act stage would let an agent to execute more actions before the agent restarts the reasoning cycle. This same feature allows several deeds from different intentions to be executed in the same reasoning cycle. Likewise, the feature to specify the number of times that each reasoning cycle stage is executed allows that several events be processed in the same reasoning cycle (when increasing the number of times that the deliberate stage is executed).

The internal concurrency level in an agent can be also improved by means of three main features. Firstly, intentions are executed concurrently by means of a round-robin scheduling mechanism that interleaves the execution of the intentions. Secondly, the mechanism for conflicting intentions improves the concurrency by allowing the concurrent execution of non-conflicting intentions. Thirdly, the fork and join mechanism improves the internal concurrency when executing an intention in the sense that different courses of deeds can be carried on concurrently. The first two ones also bring improvements for the fairness when agents are executing its intentions, that is, intentions with less work to do are expected to finish before than those intentions with a lot of work to do. Intentions with a lot of work need more cycles to finish.

The use of fork and join can also reduce the idle times (i.e., the time that the execution of an intention remains blocked due to some reason). Alternative courses of deeds can be specified when using fork with join-xor, so that an intention could avoid to get blocked in its execution (e.g., waiting for the reply of a synchronous message) by allowing

that as soon as one of the alternative courses of deeds terminate, the execution of the intention (i.e., its main course of deeds) proceeds.

The fork and join feature permits a more natural (and easier) way to synchronize the execution of intentions. While a typical approach to write a similar behavior is to write different plans and manually control the execution of the plans (e.g., by storing information in the belief base), the fork and join mechanism does it without any extra effort and without mixing control information with knowledge in the belief base. For example, the join-and can naturally waits for the completion of all concurrent plan bodies without other manual and more complex controls. Other examples of where to apply the fork and join mechanism are illustrated in Sec. 4.3.2.

Finally, the conflicts mechanism brings improvements in the programming point of view, such as flexibility to specify conflicts. On the one hand, the developer can explicitly specify the conflicts by informing the plan names or events (e.g., the adoption of goals). On the other hand, the developer can create conflict identifiers so that one does not need to care about other plans, how they work, their names, etc. The conflicts among plans are specified without explicitly referring to the specific plan names (or events), but in a more abstract way. The use of conflict identifiers brings benefits to maintain, reuse, and to better identify the reasons for the conflict, once the conflict identifiers can be created according to the reason for the conflict (e.g., a conflict identifier with the name of a critical resource can mean that the reason for the conflict is that resource). The improvement on the reuse can be seen when plans are exchanged among agents. The exchanged plans can be added in the plan library without the necessity for the developer to explicitly update the conflict sets to specify conflicts with the new added plans. Depending on how the conflict identifier was created (e.g., the name of a critical resource), the execution platform itself can discover which are the corresponding conflicting plans. All plans that use the same critical resource are expected to have the same conflict identifier in their conflict sets. The maintenance is improved due to similar reasons. For example, if a developer adds a new plan that uses a critical resource R, then the developer can write the plan and inform the conflict identifier related to resource R in its conflict set, without care about which other plans that use the same resource R.

It is important to highlight that the aforementioned improvements are only possible to be achieved when we consider an execution platform that implements the respective features and lets a MAS developer to configure the execution of a MAS according to the needs. For

example, while adopting a synchronous reasoning cycle with one UE per agent showed the fastest response time in the multiplication table scenario, the fastest response time in the Fibonacci scenario happens when an asynchronous reasoning cycle is adopted. This contrast of results between the Fibonacci and the multiplication table scenarios shows the importance of integrating both synchronous and asynchronous executions, and choose for the best one according to the characteristics of the target application scenario.

Not all features identified in the literature were integrated in our model and architecture for agents and MAS. For example, there is not support for executing intentions in different UEs. The main reason for not supporting this feature is due to harm the semantics of the execution. While scheduling the execution of the intentions by means of an algorithm, such as the round-robin, allows to define a precise semantics of execution as well as to clearly understand how the intentions are selected and executed, the execution of the intentions by independent UEs does not provide these facilities. The UEs are managed by the operational system and we can not easily control the amount of time that each intention is actually being executed or even in which order and how they are being selected. In addition, the execution of threaded intentions is clearly not scalable, resulting in an explosion of UEs as soon as the number of intentions in execution increase. This is not difficult to happen in MAS because agents usually have several active intentions at the same time.

Another feature not supported by the thesis is the priorities for intentions. Although the priorities can play an important role in the agent execution, a deeper investigation is necessary to be done in order to integrate priorities and conflicting intentions. The main question that rises is how to deal with a higher priority intention that was just instantiated while a conflicting lower priority intention is already in execution. While the interruption of the lower priority intention may drive the MAS to an inconsistent state (e.g., the interruption could happen while the belief base is being updated), the non-interruption of the lower priority intention may cause a delay for the higher priority intention to be executed.

Our approach does not support the idea of sub-agents, where an agent can be decomposed on sub-agents. Besides the idea of sub-agents be another interesting approach to exploit concurrency, we believe that the features already integrated in Jason(P) let us to implement the scenarios allowed by approaches that adopt the idea of sub-agents. For example, agents with many intentions can delegate new intentions to

other agents, which do not necessarily need to be a sub-agent.

Finally, although the agent communication is not directly handled by different UEs, we improved the communication when it is done synchronously, as already stated before. While the communication can be handled in one branch of the intention, another course of deeds can be executed in another branch, totally independent of the branch where the communication is being handled.

The originality in the thesis is to analyze, identify, and combine different techniques to improve the MAS execution by exploiting concurrency. The analysis of how to exploit concurrency in the MAS execution was never done before and such analysis gives a better understanding about the aspects where concurrency can be exploited to improve the execution of a MAS, and gives directions to fill the gap in the current state-of-the-art. The gap consists on a limitation of the current execution platforms to properly exploit concurrency in certain scenarios or considering different underlying hardware. In the thesis, we fill this gap by proposing a model and architecture for agents and MAS, and integrate it in an execution platform to bring more flexibility to configure how a MAS should be actually executed. For example, to configure how UEs should be distributed among the agents in the MAS, how to schedule the execution of the agents and their intentions, how to execute the reasoning cycle, etc.

As practical results of the thesis we can point (1) the integration of our BDI agent and MAS model and architecture and other features and mechanisms in the Jason platform; (2) the performed experiments that brought important results and statistics related to the adoption of certain concurrency configurations; (3) the characterization of the MAS execution that allowed us to identify five main characteristics that are useful to consider when evaluating an MAS (the computation load, intention load, perception load, communication load, and MAS population); and (4) the conception of the key scenarios used in the experiments that lets one to stress different characteristics and elements of the agent architecture, such as a specific stage of the reasoning cycle (e.g., the multiplication table scenario). The key scenarios can be used as a reference for a MAS developer to understand which configuration can be better to be adopted in a MAS application with similar characteristics.

The contributions of the thesis affects the use of MAS technologies and bring the MAS technology closer to scenarios found in industry, where different hardware infrastructures can be found in computers and other machines. A first implication for the MAS developers is that a MAS developer needs to have knowledge not only about the details and

requirements of the application, but also about the hardware infrastructure where the application will be executed. This can be seen by some MAS developers as a weakness, once the agent-oriented programming paradigm is meant to avoid a close contact between the developer and low-level issues, it actually abstracts this contact by some high-level abstractions, such as intentions instead of UEs. However, we still keep a good level of abstraction as much as we can. For example, the specification of conflicts by means of informing conflict identifiers, events, and plan names is far more abstract for agent-oriented programming than explicitly use synchronized blocks as in Java. In addition, a closer relation between the MAS developer and some lower-level issues can result on important improvements in the MAS execution, especially in critical applications.

7 FINAL CONSIDERATIONS

In this thesis, we analyzed different aspects that can be considered when developing execution platforms to better take advantage of multi-core computers and related parallel hardware. The analysis was made in the MAS level, which focus on how an execution platform manages the execution of several agents; the agent level, which is related to how the execution platform manages the execution of the internal elements of each agent individually; and in the intention level, which focus on how an execution platform manages the execution of intentions.

Based on the analysis and the current state-of-the-art, we conceived a model and architecture for BDI agents and MAS. The main characteristics of the proposed model and architecture are the support to a wider number of concurrency features, not strictly limited to the state-of-the-art, and flexibility to configure the execution of a MAS according to the needs of the application. The model and architecture integrate solutions for distributing UEs among the agents, for managing and scheduling execution of agents, for executing the reasoning cycle and its stages, and for managing and scheduling the execution of intentions. Agents can be executed by their own UE or by the UEs from a pool shared among all agents in the MAS. Their reasoning cycle can be executed synchronously or asynchronously, and the full cycle or each specific stage can be executed a certain number of times when the agent is selected. Intentions execute concurrently by means of an interleaving mechanism based on the round-robing scheduling mechanism, and constructs for forking the execution of intentions and for handling conflicts are provided.

In order to implement and evaluate the proposal, we extended the Jason platform to support a richer set of concurrency features. The evaluation was performed by means of a theoretical analysis of some features and configurations and also by means of experiments, which consisted in the development of small applications that covered key scenarios to investigate the benefits and drawbacks of different configurations used to run the MAS. The results of the evaluation highlighted the importance of developing execution platforms that lets a MAS developer to configure a MAS or adopt certain strategies to better exploit concurrency according to the requirements, demands, and characteristics intrinsic to each application. Benefits were identified in the overall MAS execution and in the execution of individual agents. We concluded that several concurrency techniques and models can be exploited to develop

execution platforms to better take advantage of multi-core architectures and related parallel hardware, as already discussed in Sec. 6. The use of pools improves scalability in the MAS without harm the execution. The asynchronous execution improves the execution in scenarios where some stages of the reasoning cycle are stressed more than the others. Setup of the number of cycles can improve the overall execution, once choosing the number of cycles can prevent the high overheads caused by contention when selecting agents from a pool queue. In addition, the number of cycles can be used to prioritize the execution of certain agents or stages of the reasoning cycle.

Although our option was to extend the Jason platform, the same extensions done in the Jason platform can be integrated in other platform, always considering that each platform has its own implementation, agent architecture, and agent reasoning cycle. For example, the 2APL and GOAL execution platforms can have their agent reasoning cycle revised in order to conceive a proper asynchronous version, as well as their agent architecture can be extended to support the use of different UEs to execute each agent component separately. In the remaining of this chapter, we highlight the publications resulted of this thesis (Sec. 7.1) and future works (Sec. 7.2).

7.1 PUBLICATIONS

This thesis resulted in some publications, where we proposed changes in the agent reasoning cycle and agent architecture, extended the Jason platform, and performed experiments. In our first publication, available in the proceedings of the 3rd International Workshop on Engineering Multi-Agent Systems (ZATELLI; RICCI; HÜBNER, 2015b), we developed an experimental execution platform to highlight effects caused by adopting different configurations to run an MAS. By means of the performed experiments, it was demonstrated the importance for an execution platform to provide richer options to run applications. In our second publication, available in the proceedings of the 13th European Conference on Multi-Agent Systems (ZATELLI; RICCI; HÜBNER, 2015a), the focus was on extending a concrete platform. In this case, Jason was adopted as a reference for BDI agents and an analysis was performed about its execution platform according to its concurrency features. The Jason agent reasoning cycle was revised and modified to support certain concurrency features as well as the platform and agent architecture were extended to support such new features. New exper-

iments were performed and reinforced the importance of considering a wider set of concurrency features when developing execution platforms. The experiments showed improvements on the overall MAS execution and also on the execution of individual agents, when adopted the most suitable configuration for running the applications. In our third publication, available in the proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (ZATELLI et al., 2016), the focus was on proposing a mechanism for handling conflicting intentions. We investigated several means to specify, detect, and handle conflicts; proposed a mechanism to deal with conflicts in BDI agents; and integrated it in a concrete agent platform. In addition, we informally analyzed the computational complexity of the relevant proposed algorithms.

7.2 FUTURE WORK

The present thesis can be extended in several ways. Each of the following sections briefly describes some idea of how to extend the thesis with other interesting features or lines of investigation.

7.2.1 Computational Complexity in Agent-Oriented Programming

Define metrics of efficiency in agents is a topic of increasing concern once the use of agents becomes more common (MAGARY, 2003; BIEN; LILLIS; COLLIER, 2010). Frequently, the metrics that are used to evaluate an agent are defined for each individual agent and based on temporal characteristics (e.g., execution time) or considering message exchanges (HMIDA; CHAARI; TAGINA, 2008). Some work already allows to perform *profiling* in agents, thus providing several information at run-time (HELSINGER et al., 2003; BIEN, 2008; NAGWANI, 2009). However, such metrics of efficiency and profiling are not enough to evaluate the computation complexity of an agent program, once many of them depend on the execution platform and the hardware infrastruture where the agent is executed and evaluated. This can be a limitation especially in an industrial environment, where many different machines may exist and they may work with the same agent program. Such machines also may eventually have a different hardware configuration than the machine where the agent was initially developed and tested. It is also

not possible to estimate the execution time without actually execute the MAS, which can be especially problematic when one think to perform experiments (i.e., one cannot predict how long the experiments may take). Thus, a study to define how the computational complexity of an agent program needs to be calculated is still necessary.

7.2.2 Explore Work Stealing Techniques

An alternative way to reduce the contention when executing a MAS using pools is to apply the technique of *work stealing* (BLUMOFE; LEISERSON, 1999; BLUMOFE; PAPADOPOULOS, 1998; AGRAWAL; HE; LEISERSON, 2007). Thus, instead of executing the reasoning cycle or its stages several times when a UE selects an agent for execution, the work stealing would allow to create several pools where each one has its own queue of agents to execute and UEs from one pool could steal agents from other pools' queues when their own queue gets empty. Exploring work stealing techniques is interesting in order to understand its benefits and drawbacks in the execution of applications. For example, a main desirable characteristic when executing agents is that the execution of agents is fair, which means that all agents should have the same chances and amount of time to use the CPU. In this way, the traditional work on stealing techniques need to be revised to allow that queues be balanced at run-time, once some agents can finish their executions earlier than others and the amount of agents in each queue can be different, possibly harming the fairness.

7.2.3 Integrate Environment, Interaction, and Organization

Considering a MAS composed of agents, environment, interaction, and organization (DEMAZEAU, 1995; HAMMER et al., 2006), or a subset of these components, it is interesting to exploit the concurrency not only on the agent component, but also to investigate how the concurrency can be exploited in the other components. The way in which these other components are executed certainly impact on the execution of the application and then their executions should be properly managed to take advantages of concurrency and work together with the agent component, once they actually share the same hardware infrastructure and its resources, like CPU and memory.

7.2.4 Distributing Agents

Our focus with the thesis is on a centralized approach, however, the extension of this thesis towards a distributed architecture is interesting. A distributed version can consider not only the distribution of the agents among different computers but also the distribution of the agent components. For example, the sense component and the act component could run in different computers. In addition, each intention could be executed in a different computer. Thus, we could have agents running in different computers and also a single agent spread in several computers that may have different computational powers. In this direction, new features to exploit concurrency in a distributed architecture should be proposed and integrated in the BDI agent and MAS model and architecture.

7.2.5 Updating Configurations at Run-time

Mechanisms for the agents to reason about their own configuration and change it at run-time is an interesting feature to investigate. Agents can have beliefs to let them to know how many active intentions they have, how many new percepts or messages they are receiving in certain moment, etc. Such beliefs can provide statistics to help the agents to adapt their own configuration as their needs and improve their performance at the moment. For example, if an agent is receiving many percepts in certain moment, probably it would be better to change some configuration to handle them in a more proper way. The agents should be autonomous to decide how to deal with the resources of the computer.

7.2.6 The Execution Platform as a Resource Manager

Based on the previous idea, the execution platform could also be developed with a resource manager feature. The execution platform could provide the amount of resources for the agents regarding to their needs and considering the resources available by the system at the moment. The advantage of such kind of flexible configuration at run-time is that the agents could continue running while changing their own configuration, otherwise it would be necessary to stop the agents, update their configuration, and launch them again. In some real scenarios it could be dangerous to stop the agents to update their configuration. Such

kind of feature is especially interesting when considering mobile agents, once they migrate from one computer to another and the infrastructure of hardware and available resources can be different.

7.2.7 Exploiting Other Kinds of Conflicts

In this thesis, our focus when handling conflicts was on negative interferences, however a point to be explored is on handling positive interferences (HORTY; POLLACK, 2001; THANGARAJAH; PADGHAM; WINIKOFF, 2003b; COX; DURFEE, 2003) as well as inter-agent interferences (CLEMENT; DURFEE, 1999a, 1999b; BOUTILIER; BRAFMAN, 1997; SUGAWARA et al., 2005). The main benefit of exploiting positive interferences is that agents can achieve the same goals with fewer actions because the execution of one action can benefit two or more goals if its execution is scheduled properly.

7.2.8 Real-Time Features

Finally, real-time is also an important topic to be explored in the agent paradigm (GEORGEFF; INGRAND, 1990; VINCENT et al., 2001; VIKHOREV; ALECHINA; LOGAN, 2010) and quite close to the aim of this thesis. Real-time features could be integrated in order to check which configurations could be used to accomplish the real-time constraints defined for an application.

REFERENCES

AGHA, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN 0-262-01092-5.

AGRAWAL, K.; HE, Y.; LEISERSON, C. E. Adaptive work stealing with parallelism feedback. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2007. (PPoPP '07), p. 112–120. ISBN 978-1-59593-602-8.

ALBEROLA, J. M. et al. A performance evaluation of three multiagent platforms. *Artif. Intell. Rev.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 34, n. 2, p. 145–176, ago. 2010. ISSN 0269-2821.

ALECHINA, N.; DASTANI, M.; LOGAN, B. Programming norm-aware agents. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2012. (AAMAS '12), p. 1057–1064. ISBN 0-9817381-2-5, 978-0-9817381-2-3.

ANDREWS, G. *Foundations of Multithreaded, Parallel, and Distributed Programming*. [S.l.]: Addison-Wesley, 1999.

ARMSTRONG, J. *Programming Erlang: Software for a Concurrent World*. [S.l.]: Pragmatic Bookshelf, 2007. ISBN 193435600X, 9781934356005.

BARAL, C.; GELFOND, M. Reasoning agents in dynamic domains. In: MINKER, J. (Ed.). *Logic-Based Artificial Intelligence*. [S.l.]: Springer US, 2000, (The Springer International Series in Engineering and Computer Science, v. 597). p. 257–279. ISBN 978-1-4613-5618-9.

BEHRENS, T. M. et al. *Putting APL Platforms to the Test: Agent Similarity and Execution Performance*. [S.l.], 2010.

BELLIFEMINE, F. et al. JADE - a Java agent development framework. In: BORDINI, R. H. et al. (Ed.). *Multi-Agent Programming*. [S.l.]: Springer, 2005, (Multiagent Systems, Artificial Societies, and Simulated Organizations, v. 15). p. 125–147. ISBN 0-387-24568-5.

BELLIFEMINE, F. L.; CAIRE, G.; GREENWOOD, D. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. [S.l.]: John Wiley & Sons, 2007. ISBN 0470057475.

BIEN, D. D. V. *AgentSpotter: a MAS Profiling System for Agent Factory*. Dissertação (Mestrado) — University College Dublin, 2008.

BIEN, D. D. V.; LILLIS, D.; COLLIER, R. W. Call graph profiling for multi agent systems. In: *Proceedings of the Second international conference on Languages, Methodologies, and Development Tools for Multi-Agent Systems*. Berlin, Heidelberg: Springer-Verlag, 2010. (LADS'09), p. 153–167.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM*, ACM, New York, NY, USA, v. 46, n. 5, p. 720–748, set. 1999. ISSN 0004-5411.

BLUMOFE, R. D.; PAPADOPOULOS, D. *The Performance of Work Stealing in Multiprogrammed Environments*. Austin, TX, USA, 1998.

BOER, F. de et al. A verification framework for agent programming with declarative goals. *Journal of Applied Logic*, v. 5, n. 2, p. 277 – 302, 2007. ISSN 1570-8683. Logic-Based Agent Verification.

BORDINI, R. H. et al. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, v. 30, n. 1, p. 33–44, 2006.

BORDINI, R. H. et al. *Multi-agent programming : languages, platforms and applications*. New York: Springer, 2005. (Multiagent systems, artificial societies, and simulated organizations., v. 15).

BORDINI, R. H. et al. *Multi-Agent Programming: Languages, Tools and Applications*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2009. ISBN 0387892982, 9780387892986.

BORDINI, R. H.; HÜBNER, J. F.; WOOLDRIDGE, M. *Programming multi-agent systems in AgentSpeak using Jason*. Liverpool: Wiley, 2007.

BOUTILIER, C.; BRAFMAN, R. I. Planning with concurrent interacting actions. In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.* [S.l.: s.n.], 1997. p. 720–726.

BRATMAN, M. E. *Intentions, Plans and Practical Reason*. [S.l.]: Harvard University Press, Cambridge, MA, USA, 1987.

BRATMAN, M. E.; ISRAEL, D. J.; POLLACK, M. E. Plans and resource-bounded practical reasoning. *Computational Intelligence*, v. 4, p. 349–355, 1988.

BURBECK, K.; GARPE, D.; NADJM-TEHRANI, S. Scale-up and performance studies of three agent platforms. In: *IPCCC 2004: IEEE International Conference on Performance, Computing, and Communications, Phoenix, AZ, USA*. [S.l.: s.n.], 2004. p. 857–863.

BURMEISTER, B. et al. BDI-agents for agile goal-oriented business processes. In: *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems: Industrial Track*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2008. (AAMAS '08), p. 37–44.

CAMPBELL, R. Deadlocks. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. [S.l.]: Springer US, 2011. p. 524–527. ISBN 978-0-387-09765-7.

CARDOSO, R. C.; HÜBNER, J. F.; BORDINI, R. H. Benchmarking communication in actor- and agent-based languages. In: COSSENTINO, M.; FALLAH-SEGHROUCHNI, A. E.; WINIKOFF, M. (Ed.). *EMAS@AAMAS*. [S.l.]: Springer, 2013. (Lecture Notes in Computer Science, v. 8245), p. 58–77. ISBN 978-3-642-45342-7.

CARDOSO, R. C. et al. Towards benchmarking actor- and agent-based programming languages. In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*. New York, NY, USA: ACM, 2013. (AGERE! '13), p. 115–126. ISBN 978-1-4503-2602-5.

CCITT Recommendation X.710. *Common Management Information Service Definition*. 1991. ISO/IEC 9595.

CISCHKE, C. Kabs: An extensible, parallel agent-based simulator. In: *Proceedings of the 2012 SpringSim Poster & Work-In-Progress Track*. San Diego, CA, USA: Society for Computer Simulation International, 2012. (SpringSim '12), p. 4:1–4:2.

CLARK, K.; MCCABE, F. Go! - a multi-paradigm programming language for implementing multi-threaded agents. *Annals of*

Mathematics and Artificial Intelligence, Kluwer Academic Publishers, v. 41, n. 2-4, p. 171–206, 2004. ISSN 1012-2443.

CLARK, K.; ROBINSON, P. Agents as Multi-threaded Logical Objects. In: *Computational Logic: From Logic Programming into the Future*. [S.l.]: Springer, 2002. p. 33–65.

CLARK, K. L.; MCCABE, F. G. Go! for multi-threaded deliberative agents. In: *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*. New York, NY, USA: ACM, 2003. (AAMAS '03), p. 964–965. ISBN 1-58113-683-8.

CLARK, K. L.; ROBINSON, P. J.; AMBOLDI, S. Z. Multi-threaded communicating agents in Qu-Prolog. In: TONI, F.; TORRONI, P. (Ed.). *Computational Logic in Multi-Agent Systems*. [S.l.]: Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 3900). p. 186–205. ISBN 978-3-540-33996-0.

CLARK, K. L.; ROBINSON, P. J.; HAGEN, R. Multi-threading and message communication in Qu-Prolog. *TPLP*, v. 1, n. 3, p. 283–301, 2001.

CLARK, K. L.; ROBINSON, P. J.; HAGEN, R. Multi-threading and message communication in Qu-Prolog. *CoRR*, cs.PL/0404052, 2004.

CLEMENT, B. J.; DURFEE, E. H. Identifying and resolving conflicts among agents with hierarchical plans. In: *AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities*. [S.l.: s.n.], 1999. (AAAI Technical Report WS-99-12).

CLEMENT, B. J.; DURFEE, E. H. Theory for coordinating concurrent hierarchical planning agents using summary information. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1999. (AAAI '99/IAAI '99), p. 495–502. ISBN 0-262-51106-1.

COHEN, P. R.; LEVESQUE, H. J. Intention = choice + commitment. In: *National Conference in AI*. [S.l.: s.n.], 1987.

COHEN, P. R.; LEVESQUE, H. J. Intention is choice with commitment. *Artif. Intell.*, Elsevier Science Publishers Ltd., Essex, UK, v. 42, n. 2-3, p. 213–261, mar. 1990. ISSN 0004-3702.

COLLIER, R. W.; RUSSELL, S.; LILLIS, D. Exploring AOP from an OOP Perspective. In: *Proceedings of the 5th International Workshop on Programming based on Actors, Agents & Decentralized Control (AGERE! @ SPLASH 2015)*. Pittsburgh, PA, USA: [s.n.], 2015.

COLLIER, R. W.; RUSSELL, S. E.; LILLIS, D. Reflecting on agent programming with AgentSpeak(L). In: CHEN, Q. et al. (Ed.). *PRIMA*. [S.l.]: Springer, 2015. (Lecture Notes in Computer Science, v. 9387), p. 351–366. ISBN 978-3-319-25523-1.

COSTA, A.; BITTENCOURT, G. UFSC-team: A cognitive multi-agent approach to the RoboCup'98 simulator league. In: ASADA, M.; KITANO, H. (Ed.). *RoboCup-98: Robot Soccer World Cup II*. [S.l.]: Springer Berlin Heidelberg, 1999, (Lecture Notes in Computer Science, v. 1604). p. 371–376. ISBN 978-3-540-66320-1.

COSTA, A. L.; BITTENCOURT, G. From a concurrent architecture to a concurrent autonomous agents architecture. In: VELOSO, M.; PAGELLO, E.; KITANO, H. (Ed.). *RoboCup-99: Robot Soccer World Cup III*. [S.l.]: Springer Berlin Heidelberg, 2000, (Lecture Notes in Computer Science, v. 1856). p. 274–285. ISBN 978-3-540-41043-0.

COSTA, M.; FEIJÓ, B. An architecture for concurrent reactive agents in real-time animation. In: *Brazilian Symposium on Computer Graphics and Image Processing*. [S.l.: s.n.], 1996.

COX, J. S.; DURFEE, E. H. Discovering and exploiting synergy between hierarchical planning agents. In: *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*. New York, NY, USA: ACM, 2003. (AAMAS '03), p. 281–288. ISBN 1-58113-683-8.

DASTANI, M. 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, Kluwer Academic Publishers, Hingham, MA, USA, v. 16, n. 3, p. 214–248, jun. 2008. ISSN 1387-2532.

DASTANI, M. et al. Programming agent deliberation: an approach illustrated using the 3APL language. In: *In AAMAS'03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. [S.l.]: ACM Press, 2003. p. 97–104.

DASTANI, M. et al. A programming language for cognitive agents goal directed 3APL. In: DASTANI, M.; DIX, J.; FALLAH-SEGHROUCHNI, A. E. (Ed.). *PROMAS*. [S.l.]: Springer, 2003.

(Lecture Notes in Computer Science, v. 3067), p. 111–130. ISBN 3-540-22180-8.

DASTANI, M.; RIEMSDIJK, M. B. van; MEYER, J.-J. C. Programming multi-agent systems in 3APL. In: BORDINI, R. H. et al. (Ed.). *Multi-Agent Programming*. [S.l.]: Springer, 2005, (Multiagent Systems, Artificial Societies, and Simulated Organizations, v. 15). p. 39–67. ISBN 0-387-24568-5.

DASTANI, M.; TORRE, L. van der. Programming BOID-plan agents deliberating about conflicts among defeasible mental attitudes and plans. In: *Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on*. [S.l.: s.n.], 2004. p. 706–713.

DELOACH, S. A. Specifying agent behavior as concurrent tasks. In: *Proceedings of the Fifth International Conference on Autonomous Agents*. New York, NY, USA: ACM, 2001. (AGENTS '01), p. 102–103. ISBN 1-58113-326-X.

DELZANNO, G.; MARTELLI, M. Proofs as computations in linear logic. *Theor. Comput. Sci.*, v. 258, n. 1-2, p. 269–297, 2001.

DEMAZEAU, Y. From interactions to collective behaviour in agent-based systems. In: *Proceedings of the 1st. European Conference on Cognitive Science*. Saint-Malo, France: [s.n.], 1995. p. 117–132.

DENNIS, L. A. et al. Model checking agent programming languages. *Automated Software Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 19, n. 1, p. 5–63, mar. 2012. ISSN 0928-8910.

DHAON, A.; COLLIER, R. W. Multiple inheritance in AgentSpeak(L)-style programming languages. In: *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*. New York, NY, USA: ACM, 2014. (AGERE! '14), p. 109–120. ISBN 978-1-4503-2189-1.

DIX, J.; ZHANG, Y. Impact: A multi-agent framework with declarative semantics. In: BORDINI, R. H. et al. (Ed.). *Multi-Agent Programming*. [S.l.]: Springer, 2005, (Multiagent Systems, Artificial Societies, and Simulated Organizations, v. 15). p. 69–94. ISBN 0-387-24568-5.

EITER, T.; SUBRAHMANIAN, V.; PICK, G. Heterogeneous active agents, i: Semantics. *Artificial Intelligence*, v. 108, n. 1-2, p. 179 – 255, 1999. ISSN 0004-3702.

EVERTSZ, R. et al. Implementing industrial multi-agent systems using jack. In: DASTANI, M.; DIX, J.; FALLAH-SEGHROUCHNI, A. E. (Ed.). *PROMAS*. [S.l.]: Springer, 2003. (Lecture Notes in Computer Science, v. 3067), p. 18–48. ISBN 3-540-22180-8.

EVERTSZ, R. et al. Agent oriented modelling of tactical decision making. In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2015. (AAMAS '15), p. 1051–1060. ISBN 978-1-4503-3413-6.

FALLAH-SEGHROUCHNI, A. E.; SUNA, A. Claim: A computational language for autonomous, intelligent and mobile agents. In: DASTANI, M.; DIX, J.; FALLAH-SEGHROUCHNI, A. E. (Ed.). *PROMAS*. [S.l.]: Springer, 2003. (Lecture Notes in Computer Science, v. 3067), p. 90–110. ISBN 3-540-22180-8.

FERNÁNDEZ-BAUSET, V. et al. Tuning Java to run interactive multiagent simulations over Jason. In: LI, J. (Ed.). *Australasian Conference on Artificial Intelligence*. [S.l.]: Springer, 2010. (Lecture Notes in Computer Science, v. 6464), p. 354–363. ISBN 978-3-642-17431-5.

FERNÁNDEZ, V. et al. Evaluating Jason for distributed crowd simulations. In: FILIPE, J.; FRED, A. L. N.; SHARP, B. (Ed.). *ICAART (2)*. [S.l.]: INSTICC Press, 2010. p. 206–211. ISBN 978-989-674-022-1.

FISCHER, K.; MÜLLER, J. P.; PISCHEL, M. A pragmatic BDI architecture. In: WOOLDRIDGE, M.; MüLLER, J. P.; TAMBE, M. (Ed.). *ATAL*. [S.l.]: Springer, 1995. (Lecture Notes in Computer Science, v. 1037), p. 203–218. ISBN 3-540-60805-2.

FISHER, M. A survey of concurrent METATEM - the language and its applications. In: GABBAY, D.; OHLBACH, H. (Ed.). *Temporal Logic*. [S.l.]: Springer Berlin Heidelberg, 1994, (Lecture Notes in Computer Science, v. 827). p. 480–505. ISBN 978-3-540-58241-0.

FISHER, M.; BARRINGER, H. Concurrent METATEM processes - a language for distributed AI. In: *In Proceedings of the European Simulation Multiconference*. [S.l.: s.n.], 1991.

FRANCESQUINI, E.; GOLDMAN, A.; MÉHAUT, J.-F. Improving the performance of actor model runtime environments on multicore and manycore platforms. In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*. New York, NY, USA: ACM, 2013. (AGERE! '13), p. 109–114. ISBN 978-1-4503-2602-5.

FRITZ, C.; BAIER, J. A.; MCILRAITH, S. A. ConGolog, sin trans: Compiling ConGolog into basic action theories for planning and beyond. In: BREWKA, G.; LANG, J. (Ed.). *KR*. [S.l.]: AAAI Press, 2008. p. 600–610. ISBN 978-1-57735-384-3.

GEORGEFF, M. P.; INGRAND, F. F. Decision-making in an embedded reasoning system. In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. (IJCAI'89), p. 972–978.

GEORGEFF, M. P.; INGRAND, F. F. *Managing Deliberation and Reasoning In Real-Time AI Systems*. 333 Ravenswood Ave., Menlo Park, CA 94025, Dec 1990.

GEORGEFF, M. P.; LANSKY, A. L. Reactive reasoning and planning. In: *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 2*. [S.l.]: AAAI Press, 1987. (AAAI'87), p. 677–682. ISBN 0-934613-42-7.

GIACOMO, G. et al. Indigolog: A high-level programming language for embedded reasoning agents. In: SEGHROUCHNI, A. E. F. et al. (Ed.). *Multi-Agent Programming:*. [S.l.]: Springer US, 2009. p. 31–72. ISBN 978-0-387-89298-6.

GIACOMO, G. de; LESPÉRANCE, Y.; LEVESQUE, H. J. ConGolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, Elsevier Science Publishers Ltd., Essex, UK, v. 121, n. 1-2, p. 109–169, ago. 2000. ISSN 0004-3702.

GOETZ, B. *Java theory and practice: Thread pools and work queues*. July 2002.

GOETZ, B. et al. *Java Concurrency in Practice*. [S.l.]: Addison-Wesley Longman, Amsterdam, 2006. ISBN 0321349601.

GONZALEZ, A.; ANGEL, R.; GONZALEZ, E. BDI concurrent architecture oriented to goal managment. In: *Computing Colombian Conference (8CCC), 2013 8th*. [S.l.: s.n.], 2013. p. 1–6.

GOUDA, M. G.; CHOW, C. H.; LAM, S. S. *Livelock Detection in Networks of Communicating Finite State Machines*. Austin, TX, USA, 1984.

HAMMER, F. et al. A multi-agent approach to social human behaviour in children's play. In: *Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology*. Washington, DC, USA: IEEE Computer Society, 2006. (IAT'06), p. 403–406.

HELSINGER, A. et al. Tools and techniques for performance measurement of large distributed multiagent systems. In: *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM, 2003. (AAMAS '03), p. 843–850. ISBN 1-58113-683-8.

HERLIHY, M. Transactional memories. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. [S.l.]: Springer US, 2011. p. 2079–2086. ISBN 978-0-387-09765-7.

HERLIHY, M.; MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 21, n. 2, p. 289–300, maio 1993. ISSN 0163-5964.

HERLIHY, M.; SHAVIT, N. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123705916, 9780123705914.

HERMENEGILDO, M. et al. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems (extended abstract). In: *IN PARALLELISM AND IMPLEMENTATION OF LOGIC AND CONSTRAINT LOGIC PROGRAMMING*. [S.l.]: Nova Science, 1995. p. 65–85.

HINDRIKS, K. V. Programming rational agents in GOAL. In: SEGHROUCHNI, A. E. F. et al. (Ed.). *Multi-Agent Programming:*. [S.l.]: Springer US, 2009. p. 119–157. ISBN 978-0-387-89298-6.

HINDRIKS, K. V. et al. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, Kluwer Academic Publishers, Hingham, MA, USA, v. 2, n. 4, p. 357–401, nov. 1999. ISSN 1387-2532.

HIRSCH, B.; KONNERTH, T.; HESSLER, A. Merging agents and services — the JIAC agent platform. In: BORDINI, R. H. et al. (Ed.). *Multi-Agent Programming: Languages, Tools and Applications*. [S.l.]: Springer, 2009. p. 159–185.

HMIDA, F. B.; CHAARI, W. L.; TAGINA, M. Performance evaluation of multiagent systems: communication criterion. In: *Proceedings of the 2nd KES International conference on Agent and multi-agent systems: technologies and applications*. Berlin, Heidelberg: Springer-Verlag, 2008. (KES-AMSTA'08), p. 773–782. ISBN 3-540-78581-7, 978-3-540-78581-1.

HORTY, J. F.; POLLACK, M. E. Evaluating new options in the context of existing plans. *Artif. Intell.*, v. 127, n. 2, p. 199–220, 2001.

INGRAND, F. F.; GEORGEFF, M. P.; RAO, A. S. An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and Their Applications*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 7, n. 6, p. 34–44, dez. 1992. ISSN 0885-9000.

KINNY, D. Vip: a visual programming language for plan execution systems. In: *AAMAS*. [S.l.]: ACM, 2002. p. 721–728.

KOSTIADIS, K.; HU, H. A multi-threaded approach to simulated soccer agents for the RoboCup competition. In: *RoboCup-99: Robot Soccer World Cup III*. London, UK, UK: Springer-Verlag, 2000. p. 366–377. ISBN 3-540-41043-0.

KOWALSKI, R.; SADRI, F. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, Kluwer Academic Publishers, v. 25, n. 3-4, p. 391–419, 1999. ISSN 1012-2443.

LAVENDER, R. G.; SCHMIDT, D. C. Active object: An object behavioral pattern for concurrent programming. In: *Pattern languages of program design 2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. p. 483–499. ISBN 0-201-895277.

LEE, S.-K. et al. Design and implementation of a multi-threaded tmn agent system. In: *Parallel Processing, 1999. Proceedings. 1999 International Workshops on*. [S.l.: s.n.], 1999. p. 332–337. ISSN 1530-2016.

LEITE, J. a. A.; ALFERES, J. J.; PEREIRA, L. M. MINERVA - a dynamic logic programming agent architecture. In: *Revised Papers from the 8th International Workshop on Intelligent Agents VIII*. London, UK, UK: Springer-Verlag, 2002. (ATAL '01), p. 141–157. ISBN 3-540-43858-0.

LING, Y.; MULLEN, T.; LIN, X. Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 34, n. 2, p. 42–55, abr. 2000. ISSN 0163-5980.

MAGARY, J. *Examining Performance Issues of Multiagent Systems*. [S.l.], 2003.

MASCARDI, V.; MARTELLI, M.; STERLING, L. Logic-based specification languages for intelligent software agents. *Theory Pract. Log. Program.*, Cambridge University Press, New York, NY, USA, v. 4, n. 4, p. 429–494, jul. 2004. ISSN 1471-0684.

MATTSON, T.; SANDERS, B.; MASSINGILL, B. *Patterns for Parallel Programming*. First. [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321228111.

MCCABE, F. G.; CLARK, K. L. APRIL: Agent PRocess Interaction Language. In: *Proceedings of the Workshop on Agent Theories, Architectures, and Languages on Intelligent Agents*. New York, NY, USA: Springer-Verlag New York, Inc., 1995. (ECAI-94), p. 324–340. ISBN 3-540-58855-8.

MILLER, M. S.; TRIBBLE, E. D.; SHAPIRO, J. Concurrency among strangers: Programming in e as plan coordination. In: *Proceedings of the 1st International Conference on Trustworthy Global Computing*. Berlin, Heidelberg: Springer-Verlag, 2005. (TGC'05), p. 195–229. ISBN 3-540-30007-4, 978-3-540-30007-6.

MULDOON, C. et al. Towards pervasive intelligence: Reflections on the evolution of the agent factory framework. In: SEGHROUCHNI, A. E. F. et al. (Ed.). *Multi-Agent Programming:*. [S.l.]: Springer US, 2009. p. 187–212. ISBN 978-0-387-89298-6.

MULET, L.; SUCH, J. M.; ALBEROLA, J. M. Performance evaluation of open-source multiagent platforms. In: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*. New York, NY, USA: ACM, 2006. (AAMAS '06), p. 1107–1109. ISBN 1-59593-303-4.

MUSCAR, A. Agent oriented programming: from revolution to evolution. In: JONES, A. V. (Ed.). *ICCSW*. [S.l.]: Imperial College London, 2011. (Department of Computing Technical Report, DTR11-9), p. 52–58.

MUSCAR, A. Agents for the 21st century: the Blueprint agent programming language. In: *Proc. of the 1st AAMAS Workshop on Engineering MultiAgent Systems*. [S.l.: s.n.], 2013. p. 49–64.

MUSCAR, A. *Exploring the Design Space of Agent-Oriented Programming Languages*. Tese (Doutorado) — University of Craiova, 2013.

MUSCAR, A.; BADICA, C. Investigating F# as a development tool for distributed multi-agent systems. In: *System Theory, Control, and Computing (ICSTCC), 2011 15th International Conference on*. [S.l.: s.n.], 2011. p. 1–6.

MUSCAR, A.; BADICA, C. Monadic foundations for promises in Jason. *ITC*, v. 43, n. 1, p. 65–72, 2014.

NAGWANI, N. K. Performance measurement analysis for multi-agent systems. In: *Proc. of IAMA*. [S.l.: s.n.], 2009.

NODA, I.; NAKASHIMA, H.; HANDA, K. Programming language Gaea and its application for multiagent systems. In: *Workshop on Multi-Agent Systems in Logic Programming*. [S.l.: s.n.], 1999. p. 9.

O'HARE, G. M. P. Agent Factory: An Environment for the Fabrication of Multi-Agent Systems. In: *Distributed Artificial Intelligence*. [S.l.: s.n.], 1996. p. 449–484.

PADUA, D. A. (Ed.). *Encyclopedia of Parallel Computing*. [S.l.]: Springer, 2011. ISBN 978-0-387-09765-7.

PEREIRA, L. M.; QUARESMA, P.; CENTRE, C. A. Modelling agent interaction in logic programming. In: *Science University of Tokyo*. [S.l.: s.n.], 1998. p. 150–156.

PÉREZ-CARRO, P. et al. Characterization of the Jason multiagent platform on multicore processors. *Sci. Program.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 22, n. 1, p. 21–35, jan. 2014. ISSN 1058-9244.

POKAHR, A.; BRAUBACH, L.; JANDER, K. Unifying agent and component concepts: Jadex active components. In: *Proceedings of the 8th German Conference on Multiagent System Technologies*. Berlin, Heidelberg: Springer-Verlag, 2010. (MATES'10), p. 100–112. ISBN 3-642-16177-4, 978-3-642-16177-3.

POKAHR, A.; BRAUBACH, L.; LAMERSDORF, W. A goal deliberation strategy for BDI agent systems. In: *Proceedings of the Third German Conference on Multiagent System Technologies*. Berlin, Heidelberg: Springer-Verlag, 2005. (MATES'05), p. 82–93. ISBN 3-540-28740-X, 978-3-540-28740-7.

POKAHR, A.; BRAUBACH, L.; LAMERSDORF, W. Jadex: A BDI reasoning engine. In: BORDINI, R. H. et al. (Ed.). *Multi-Agent Programming*. [S.l.]: Springer, 2005, (Multiagent Systems, Artificial Societies, and Simulated Organizations, v. 15). p. 149–174. ISBN 0-387-24568-5.

PRAUN, C. von. Race conditions. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. [S.l.]: Springer US, 2011. p. 1691–1697. ISBN 978-0-387-09765-7.

RAO, A. S. AgentSpeak(L): BDI agents speak out in a logical computable language. In: *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World : Agents Breaking Away: Agents Breaking Away*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996. (MAAMAW '96), p. 42–55. ISBN 3-540-60852-4.

RAO, A. S.; GEORGE, M. P. BDI agents: From theory to practice. In: *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*. [S.l.: s.n.], 1995. p. 312–319.

RAYNAL, M. *Concurrent Programming: Algorithms, Principles, and Foundations*. Berlin: Springer, 2013. ISBN 978-3-642-32026-2.

RICCI, A. et al. Environment programming in CArtAgO. In: *Multi-Agent Programming II: Languages, Platforms and Applications, Multiagent Systems, Artificial Societies, and Simulated Organizations*. US: Springer-Verlag, 2009. p. 259–288.

RICCI, A.; SANTI, A. Programming abstractions for integrating autonomous and reactive behaviors: An agent-oriented approach. In: *Proceedings of the 2Nd Edition on Programming Systems, Languages*

and Applications Based on Actors, Agents, and Decentralized Control Abstractions. New York, NY, USA: ACM, 2012. (AGERE! '12), p. 83–94. ISBN 978-1-4503-1630-9.

RICCI, A.; SANTI, A. Concurrent object-oriented programming with agent-oriented abstractions: The ALOO approach. In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*. New York, NY, USA: ACM, 2013. (AGERE! '13), p. 127–138. ISBN 978-1-4503-2602-5.

RICCI, A.; VIROLI, M. simpA: an agent-oriented approach for prototyping concurrent applications on top of Java. In: AMARAL, V. et al. (Ed.). *PPPJ*. [S.l.]: ACM, 2007. (ACM International Conference Proceeding Series, v. 272), p. 185–194. ISBN 978-1-59593-672-1.

RICCI, A.; VIROLI, M.; PIANCASTELLI, G. simpA: An agent-oriented approach for programming concurrent applications on top of Java. *Science of Computer Programming*, v. 76, n. 1, p. 37 – 62, 2011. ISSN 0167-6423.

RIEMSDIJK, M. B.; DASTANI, M.; MEYER, J.-J. C. Goals in conflict: Semantic foundations of goals in agent programming. *Autonomous Agents and Multi-Agent Systems*, Kluwer Academic Publishers, Hingham, MA, USA, v. 18, n. 3, p. 471–500, jun. 2009. ISSN 1387-2532.

RODRIGUEZ, S.; GAUD, N.; GALLAND, S. SARL: a general-purpose agent-oriented programming language. In: *The 2014 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. Warsaw, Poland: IEEE Computer Society Press, 2014.

RODRIGUEZ, S.; HILAIRE, V.; KOUKAM, A. Towards a methodological framework for holonic multi-agent systems. In: *Fourth International Workshop of Engineering Societies in the Agents World*. Imperial College London, UK (EU): [s.n.], 2003. p. 179–185.

RODRIGUEZ, S.; HILAIRE, V.; KOUKAM, A. A holonic approach to model and deploy large scale simulations. In: ANTUNES, L.; TAKADAMA, K. (Ed.). *Multi-Agent-Based Simulation VII, International Workshop, MABS 2006, Hakodate, Japan, May 8, 2006, Revised and Invited Papers*. [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 4442), p. 112–127. ISBN 978-3-540-76536-3.

RODRIGUEZ, S. A. *From analysis to design of holonic multi-agent systems: A framework, methodological guidelines and applications.* Tese (Doutorado) — Université de Technologie de Belfort-Montbéliard and Université de Franche-Compté, December 2005.

SADRI, F.; TONI, F. Computational logic and multi-agent systems: a roadmap. *Computational Logic, Special Issue on the Future Technological Roadmap of Compulog-Net*, 1999.

SAITO, M. M. et al. Parallel agent-based simulator for influenza pandemic. In: *Proceedings of the 10th International Conference on Advanced Agent Technology*. Berlin, Heidelberg: Springer-Verlag, 2012. (AAMAS'11), p. 361–370. ISBN 978-3-642-27215-8.

SANTI, A.; RICCI, A. A task framework on top of a concurrent oop language rooted on agent-oriented abstractions. In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*. New York, NY, USA: ACM, 2013. (AGERE! '13), p. 139–144. ISBN 978-1-4503-2602-5.

SANTORO, N. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. [S.l.]: Wiley-Interscience, 2006. ISBN 0471719978.

SARDINA, S. et al. On the semantics of deliberation in indigolog—from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, Kluwer Academic Publishers, Hingham, MA, USA, v. 41, n. 2-4, p. 259–299, ago. 2004. ISSN 1012-2443.

SCHMIDT, D. C. Evaluating architectures for multithreaded object request brokers. *Commun. ACM*, ACM, New York, NY, USA, v. 41, n. 10, p. 54–60, out. 1998. ISSN 0001-0782.

SCHROEDER, M.; WAGNER, G. Vivid agents: Theory, architecture, and applications. *Applied Artificial Intelligence*, v. 14, n. 7, p. 645–675, 2000.

SCOTT, M. L. Synchronization. In: PADUA, D. (Ed.). *Encyclopedia of Parallel Computing*. [S.l.]: Springer US, 2011. p. 1989–1996. ISBN 978-0-387-09765-7.

SHANAHAN, M. Reinventing shakey. In: MINKER, J. (Ed.). *Logic-Based Artificial Intelligence*. [S.l.]: Springer US, 2000, (The

Springer International Series in Engineering and Computer Science, v. 597). p. 233–253. ISBN 978-1-4613-5618-9.

SHAPIRO, S. et al. Revising conflicting intention sets in BDI agents. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2012. (AAMAS '12), p. 1081–1088. ISBN 0-9817381-2-5, 978-0-9817381-2-3.

SHOHAM, Y. AGENT0: A simple agent language and its interpreter. In: DEAN, T. L.; MCKEOWN, K. (Ed.). *AAAI*. [S.l.]: AAAI Press / The MIT Press, 1991. p. 704–709. ISBN 0-262-51059-6.

SUGAWARA, T. et al. Predicting possible conflicts in hierarchical planning for multi-agent systems. In: *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. New York, NY, USA: ACM, 2005. (AAMAS '05), p. 813–820. ISBN 1-59593-093-0.

SYER, M. D.; ADAMS, B.; HASSAN, A. E. Identifying performance deviations in thread pools. In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. [S.l.: s.n.], 2011. p. 83–92. ISSN 1063-6773.

SYME, D.; PETRICEK, T.; LOMOV, D. The F# asynchronous programming model. In: *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*. Berlin, Heidelberg: Springer-Verlag, 2011. (PADL'11), p. 175–189. ISBN 978-3-642-18377-5.

TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

TARAU, P. Jinni: Intelligent mobile agent programming at the intersection of Java and Prolog. In: *In Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*. [S.l.: s.n.], 1999. p. 109–123.

THANGARAJAH, J.; PADGHAM, L. Computationally effective reasoning about goal interactions. *J. Autom. Reason.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 47, n. 1, p. 17–56, jun. 2011. ISSN 0168-7433.

THANGARAJAH, J.; PADGHAM, L.; HARLAND, J. Representation and reasoning for goals in BDI agents. *Aust. Comput. Sci. Commun.*,

IEEE Computer Society Press, Los Alamitos, CA, USA, v. 24, n. 1, p. 259–265, jan. 2002.

THANGARAJAH, J.; PADGHAM, L.; WINIKOFF, M. Detecting & avoiding interference between goals in intelligent agents. In: GOTTLOB, G.; WALSH, T. (Ed.). *IJCAI*. [S.l.]: Morgan Kaufmann, 2003. p. 721–726.

THANGARAJAH, J.; PADGHAM, L.; WINIKOFF, M. Detecting & exploiting positive goal interaction in intelligent agents. In: *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*. New York, NY, USA: ACM, 2003. (AAMAS '03), p. 401–408. ISBN 1-58113-683-8.

THANGARAJAH, J. et al. Avoiding resource conflicts in intelligent agents. In: HARMELEN, F. van (Ed.). *ECAI*. [S.l.]: IOS Press, 2002. p. 18–22.

TSUTSUI, S.; FUJIMOTO, N. Parallel ant colony optimization algorithm on a multi-core processor. In: *Swarm Intelligence*. [S.l.]: Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6234). p. 488–495. ISBN 978-3-642-15460-7.

VIKHOREV, K. et al. An operational semantics for AgentSpeak(RT) (preliminary report). In: *Proceedings of the Ninth International Workshop on Declarative Agent Languages and Technologies (DALT 2011)*. Taipei, Taiwan: [s.n.], 2011.

VIKHOREV, K.; ALECHINA, N.; LOGAN, B. The ARTS real-time agent architecture. In: _____. *Languages, Methodologies, and Development Tools for Multi-Agent Systems: Second International Workshop, LADS 2009, Torino, Italy, September 7-9, 2009, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 1–15. ISBN 978-3-642-13338-1.

VIKHOREV, K.; ALECHINA, N.; LOGAN, B. Agent programming with priorities and deadlines. In: SONENBERG, L. et al. (Ed.). *AAMAS*. [S.l.]: IFAAMAS, 2011. p. 397–404. ISBN 978-0-9826571-5-7.

VINCENT, R. et al. Implementing Soft Real-Time Agent Control. *Proceedings of the 5th International Conference on Autonomous Agents*, ACM Press, Montreal, p. 355–362, June 2001.

VRBA, P. Java-based agent platform evaluation. In: MARíK, V.; MCFARLANE, D. C.; VALCKENAERS, P. (Ed.). *HoloMAS*. [S.l.]:

Springer, 2003. (Lecture Notes in Computer Science, v. 2744), p. 47–58. ISBN 3-540-40751-0.

WANG, X.; CAO, J.; WANG, J. A runtime goal conflict resolution model for agent systems. In: *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2012 IEEE/WIC/ACM International Conferences on*. [S.l.: s.n.], 2012. v. 2, p. 340–347.

WEERASOORIYA, D.; RAO, A.; RAMAMOHANARAO, K. Design of a concurrent agent-oriented language. In: WOOLDRIDGE, M.; JENNINGS, N. (Ed.). *Intelligent Agents*. [S.l.]: Springer Berlin Heidelberg, 1995, (Lecture Notes in Computer Science, v. 890). p. 386–401. ISBN 978-3-540-58855-9.

WIECZOREK, D.; ALBAYRAK, S. Open scalable agent architecture for telecommunication applications. In: ALBAYRAK, S.; GARIJO, F. (Ed.). *Intelligent Agents for Telecommunication Applications*. [S.l.]: Springer Berlin Heidelberg, 1998, (Lecture Notes in Computer Science, v. 1437). p. 233–249. ISBN 978-3-540-64720-1.

WINIKOFF, M. Jack$^{\text{tm}}$ intelligent agents: An industrial strength platform. In: *Multi-Agent Programming*. [S.l.: s.n.], 2005. p. 175–193.

WOOLDRIDGE, M. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, v. 10, n. 2, p. 115–152, Outubro 1995.

WOOLDRIDGE, M. *An introduction to multiagent systems*. Chichester: John Wiley & Sons Ltd, 2002.

ZATELLI, M. R. et al. Conflicting goals in agent-oriented programming. In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. New York, NY, USA: ACM, 2016. (AGERE 2016), p. 21–30. ISBN 978-1-4503-4639-9.

ZATELLI, M. R.; RICCI, A.; HÜBNER, J. F. A concurrent architecture for agent reasoning cycle execution in Jason. In: *Multi-Agent Systems and Agreement Technologies (EUMAS 2015/AT 2015)*. [S.l.]: Springer, 2015. (LNAI, v. 9571).

ZATELLI, M. R.; RICCI, A.; HÜBNER, J. F. Evaluating different concurrency configurations for executing multi-agent systems. In: BALDONI, M.; BARESI, L.; DASTANI, M. (Ed.). *Engineering Multi-Agent Systems: Third International Workshop, EMAS 2015*. Cham: Springer, 2015. (LNCS, v. 9318), p. 212–230.

ZHANG, H.; HUANG, S.-Y. A parallel BDI agent architecture. In: *Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*. [S.l.: s.n.], 2005. p. 157–160.

ZHANG, H.; HUANG, S. Y. Are parallel BDI agents really better? In: *Proceedings of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva Del Garda, Italy*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2006. p. 305–309. ISBN 1-58603-642-4.

ZHANG, H.; HUANG, S.-Y. A general framework for parallel BDI agents. In: *IAT*. [S.l.]: IEEE Computer Society, 2006. p. 103–112.

ZHANG, H.; HUANG, S.-Y. A general framework for parallel BDI agents in dynamic environments. *Web Intelligence and Agent Systems: An International Journal*, v. 6, p. 327–351, 2008.

ZHENG, G.-P.; HOU, Z.-Y.; YIN, X.-N. Research of the agent technology based on multi-thread in transformer substation communication. In: *Machine Learning and Cybernetics, 2006 International Conference on*. [S.l.: s.n.], 2006. p. 56–60.

APPENDIX A – Languages and Platforms References

AGENT0	(SHOHAM, 1991)
Jason	(BORDINI; HÜBNER; WOOLDRIDGE, 2007)
simpA	(RICCI; VIROLI, 2007; RICCI; VIROLI; PIANCASTELLI, 2011)
simpAL	(RICCI; SANTI, 2012)
ALOO	(RICCI; SANTI, 2013; SANTI; RICCI, 2013)
JADE	(BELLIFEMINE et al., 2005; BELLIFEMINE; CAIRE; GREENWOOD, 2007)
Jadex	(POKAHR; BRAUBACH; LAMERSDORF, 2005b; POKAHR; BRAUBACH; JANDER, 2010)
JACK	(EVERTSZ et al., 2003; WINIKOFF, 2005; EVERTSZ et al., 2015)
3APL	(HINDRIKS et al., 1999; DASTANI; RIEMSDIJK; MEYER, 2005; DASTANI et al., 2003b, 2003a)
2APL	(DASTANI, 2008)
N-2APL	(ALECHINA; DASTANI; LOGAN, 2012)
CLAIM	(FALLAH-SEGHROUCHNI; SUNA, 2003)
Qu-Prolog	(CLARK; ROBINSON; HAGEN, 2004; CLARK; ROBINSON; AMBOLDI, 2006; CLARK; ROBINSON; HAGEN, 2001)
QuP++	(CLARK; ROBINSON, 2002)
Go!	(CLARK; MCCABE, 2003, 2004)
MINERVA	(LEITE; ALFERES; PEREIRA, 2002)
ConGolog	(GIACOMO; LESPÉRANCE; LEVESQUE, 2000; FRITZ; BAIER; MCILRAITH, 2008)
IndiGolog	(SARDINA et al., 2004; GIACOMO et al., 2009)
ViP	(KINNY, 2002)
Ciao Prolog	(HERMENEGILDO et al., 1995)
GAEA	(NODA; NAKASHIMA; HANDA, 1999)
Jinni	(TARAU, 1999)
Blueprint	(MUSCAR, 2013b, 2013a)

IMPACT	(EITER; SUBRAHMANIAN; PICK, 1999; DIX; ZHANG, 2005)
METATEM	(FISHER; BARRINGER, 1991; FISHER, 1994)
$\boldsymbol{E_{hhf}}$	(DELZANNO; MARTELLI, 2001)
Agent Factory Framework	(O'HARE, 1996; MULDOON et al., 2009)
JIAC Agent Framework	(WIECZOREK; ALBAYRAK, 1998; HIRSCH; KONNERTH; HESSLER, 2009)
Vivid agents	(SCHROEDER; WAGNER, 2000)
GOAL	(HINDRIKS, 2009)
ASTRA	(DHAON; COLLIER, 2014; COLLIER; RUSSELL; LILLIS, 2015b, 2015a)
SARL	(RODRIGUEZ; GAUD; GALLAND, 2014)
AgentSpeak(L)	(WEERASOORIYA; RAO; RAMAMOHANARAO, 1995; RAO, 1996)
AgentSpeak(RT)	(VIKHOREV; ALECHINA; LOGAN, 2011; VIKHOREV et al., 2011)
APRIL	(MCCABE; CLARK, 1995)
INTERRAP	(FISCHER; MÜLLER; PISCHEL, 1995)

APPENDIX B – State-of-the-Art

This appendix details some of the related work (agent languages, execution platforms, applications, architectures) that, somehow, exploit concurrency in the context of MAS. The aim of this appendix is to present in more details some of the works introduced in Sec. 3. Considering that several works implement the features identified in Sec. 3, in this appendix we focus on those features and works that we consider that need a deeper analysis. For example, while works have different ideas about how to execute the internal elements of an agent concurrently, the use of pools or executing each agent by its own UE is more similar among the current works.

Firstly, we present works that use a parallel approach for the agent architecture, that is, those that execute internal elements of an agent concurrently (Sec. B.1). Secondly, we present some works that focus on providing concurrency features in order to improve the MAS execution (Sec. B.2). Thirdly, we present works that implement real applications and that depend on certain concurrency features to accomplish some requirements of execution (Sec. B.3).

B.1 PARALLEL ARCHITECTURES

The main idea of this section is to present approaches that separate the elements of the agent reasoning cycle in different components that can run concurrently. The main reason for a concurrent execution of the internal components of an agent is that sometimes the reasoning cycle can take more time to execute and the reactivity may not be assured. As a consequence, agents are not able to handle the environmental changes promptly, or to complete a task because of a very dynamic environment needs constantly changes in the belief base, resulting in less rational and reactive agents (ZHANG; HUANG, 2005, 2006a, 2006b, 2008). Sec. B.1.1 presents the work of Zhang and Huang, while Sec. B.1.2 presents the work of Kostiadis and Hu, and Sec. B.1.3 presents the work of Costa and Bittencourt. Finally, Sec. B.1.4 briefly presents other works that also use a parallel architecture for the agent reasoning cycle.

B.1.1 Zhang and Huang

The approach proposed in (ZHANG; HUANG, 2005, 2006a, 2006b, 2008) is based on how the human brain seems to work. The human brain performs many things at the same time, like planning, walking,

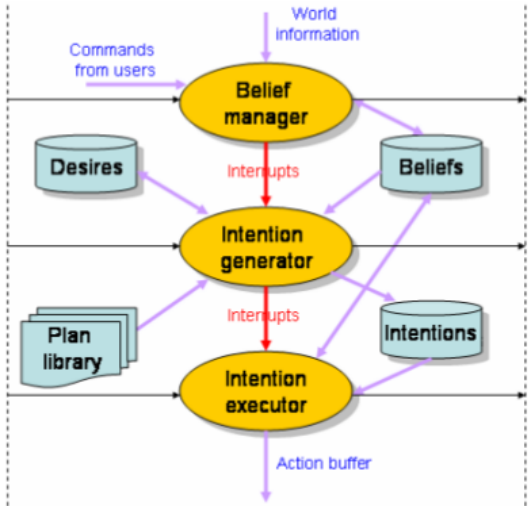

Figure 35 – Parallel BDI architecture (ZHANG; HUANG, 2008).

watching for traffic, etc. In order to achieve a similar behavior, the authors propose an agent architecture to simulate the parallel behavior of the internal components of an agent and each component has its own UEs. The number of UEs for each component depends of how many elements each component will have inside. Moreover, individual intentions can also run in separated UEs, the authors define that only one of these UEs can be active in each moment.

Zhang and Huang also define some desired characteristics for the agents are: (1) to monitor the environment all the time and react promptly to emergencies; (2) to reconsider and re-schedule goals, intentions, and actions in reaction to unexpected or new information; (3) to perform multiple actions at once; (4) to perceive, deliberate, and act simultaneously; and (5) to prioritize the deliberations and intention executions.

The architecture of Zhang and Huang is depicted in Figure 35. An agent consists of the three components: belief manager, intention generator, and intention executor. These components represent the three steps in the reasoning cycle of an agent: sense, deliberate, and act, respectively, and are executed concurrently by means of their own UEs. For example, the belief manager may be sorting new information about the traffic, the intention generator may be planning for the next step of the journey, and the intention executor may be controlling the current

movements of the car towards the destination. In addition, the belief manager can have several environment monitors, which means that the agents can have one UE for each source of information about the environment. Thus, agents have the ability to handle new beliefs, responding quickly to changes, and execute intentions, all these activities concurrently.

The three proposed components retrieve and update data in four structures: *beliefs*, *desires*, *intentions*, and *plan library*. It is possible to define priorities in order to decide what will be executed in each moment. If the priority of a new intention is higher than the current intention, the current intention is suspended and the new intention will be executed. The intentions are separated in three sets: *inactive*, *pending*, and *executing*. These sets can support the scheduling and reconsideration of intentions.

The coordination among the components is done my means of interruptions. For example, if new beliefs are produced, an interruption will be produced to notify the intention generator. The interruptions can have their own priority. For example, if a new information received by the agent has not a higher priority than what the agent is currently doing, the new information will wait for a future processing. Thus, the interrupt mechanism can ensure that emergencies can be handle immediately, while the agent is still able to careful deliberate when it is required.

In order to evaluate their approach, the authors perform experiments comparing their parallel architecture against sequential architectures. As a metric for the evaluation, the authors use the response time, which is defined as the time between the arrival of some event and the end of the execution of the plan chosen to handle the event. The authors conclude that, in the performed experiments, the parallel architecture has the fastest response time, around 3 times faster than sequential architectures.

B.1.2 Kostiadis and Hu

In (KOSTIADIS; HU, 2000), the authors propose a very similar agent architecture to (ZHANG; HUANG, 2005, 2006a, 2006b, 2008). Their aim is to provide a multi-thread architecture for agents for the RoboCup competition. At regular 150ms time intervals the RoboCup server broadcasts visual information to all clients, according on their position. In addition, the auditory sensory data can be received at completely ran-

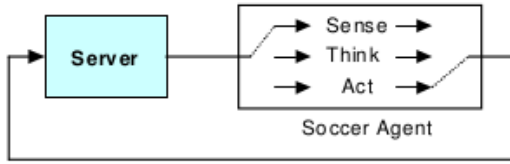

Figure 36 – Multi-threaded model (KOSTIADIS; HU, 2000).

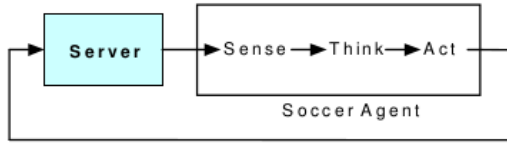

Figure 37 – Single-threaded model (KOSTIADIS; HU, 2000).

dom intervals. The agents need to react to the perceptions within an interval of 100ms, therefore they need to act as soon as possible.

In their architecture, the sense, deliberate, and act are executed concurrently by means of different UEs, as depicted in Figure 36. While the *sense* UE is responsible for waiting data from the server, the *act* UE is responsible for timing (guarantee that the agent can answer within 100ms) and sending actions to the server, and the *deliberate* UE is responsible for process the information and decide the next actions. Thus, several different computations can be performed while waiting for the perceptions from the server.

Finally, the experiments performed in (KOSTIADIS; HU, 2000) also shows improvements in the efficiency, reactivity, and scalability when comparing the parallel architecture presented in Figure 36 with the sequential architecture presented in Figure 37. The parallel architecture showed especially advantages when it is necessary to handle I/O operations and the agents need to respond more precisely within a limited time.

B.1.3 Costa and Bittencourt

Costa and Bittencourt (COSTA; BITTENCOURT, 1999, 2000) also propose a concurrent agent architecture (Figure 38) to implement an agent team for the RoboCup. Their architecture consists of three concurrent processes that encapsulate different inference engines. Their architecture makes decisions in three different levels: reactive, instinc-

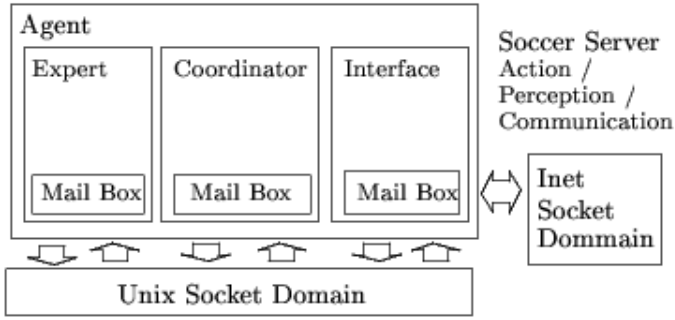

Figure 38 – Parallel agent architecture (COSTA; BITTENCOURT, 2000).

tive, and cognitive.

The first parallel architecture (COSTA; BITTENCOURT, 1999) was based on three processes: *interface*, *coordinator*, and *expert*. The process *interface* was designed to handle perception and action. The process *coordinator* was responsible for the agent communication and for starting and conducting the cooperation processes. The process *expert* was responsible for planning and decision making. It has a knowledge-based system encapsulated with information provided by means of messages and perceptions. Each of these three processes has a mailbox to exchange messages by means of sockets and behave as independent processes.

The decision making in the first parallel architecture was centralized and it has presented some problems with the synchronization between agent and environment. The response time was also too high, making the agents fail to respect the real-time constraints imposed by the competition. In order to solve these issues, another agent architecture was proposed in (COSTA; BITTENCOURT, 2000), and instead of using a centralized decision making, it was used an architecture based on three decision levels: reactive, instinctive, and cognitive, implemented in a concurrent way. The new architecture kept the same three processes: *interface*, *coordinator*, and *expert*, however each of these processes encapsulates a different inference engine, each one responsible for one of the three decision levels. Moreover, each process is composed of two UEs. While one UE is responsible for handling the interruptions caused because of the arrival of a new message, the other UE is responsible for other activities.

The reactive level inference engine is implemented in the *interface* process and it is responsible for the real-time response. The instinctive

level inference engine is implemented in the *coordinator* process and it is responsible for updating the symbolic variables used by the cognitive level and for choosing the adequate behaviors. In order to do that, the instinctive level has an expert system. Finally, the cognitive level inference engine is implemented in the *expert* process and it is responsible for determining the local and global goals of the agent. This level is also implemented as an expert system, which is much more complex than the instinctive level. The aim of this expert system is to update the rules in the instinctive level.

As a result, the authors state that the agents following the proposed architecture can react to environment stimulus, make plans, establish goals, and perform complex agent cooperation strategies concurrently. Moreover, the proposed architecture made it possible to respect real-time constraints, as desired by the competition.

B.1.4 Others

The parallel BDI architecture proposed in (GONZALEZ; ANGEL; GONZALEZ, 2013) is inspired in the human practical reasoning in which some mental processes are performed in parallel. The architecture includes three main components: the belief cycle, the desire goal management process, and the mean ends intention-based manager. The belief manager supports the beliefs of the agent and it is responsible for managing the cycle of beliefs, which is composed of four states: emergency, update, inference, and dead. The desire management is responsible for dealing with desires and converting them in intentions. Finally, the mean ends manager is responsible for handling the intentions and make them execute. Each of these three components is controlled by one meta-agent, each of them running in different UEs. Vivid agents (SCHROEDER; WAGNER, 2000) uses a similar approach and agents have the perception-reaction cycle running concurrently with the planning activity, each one with its UE. Thus, the aim of Vivid agents is to improve the agent reactivity. Moreover, in (KOWALSKI; SADRI, 1999), the deliberate stage can be interrupted to accept inputs or to generate outputs.

Besides executing the internal components of an agent explicitly adopting UEs, other approaches use sub-agents (COSTA; FEIJÓ, 1996; RODRIGUEZ, 2005; RODRIGUEZ; HILAIRE; KOUKAM, 2003, 2007) as a way to exploit intra-agent level concurrency, which means that an agent can be composed of several sub-agents so that internal activities of an

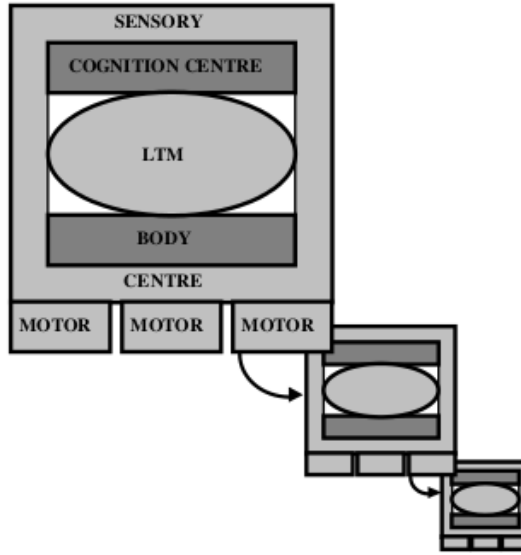

Figure 39 – The agent structure (COSTA; FEIJÓ, 1996).

agent can be performed, possibly concurrently, by its sub-agents. Sub-agents can be responsible for the reactive response, planning, belief revision, goal management, learning, dialogue management, preference evaluation, and diagnosis functionalities.

An example of this kind of agent architecture is presented in (COSTA; FEIJÓ, 1996). The authors propose an architecture for real-time behavior animation based on parallel interactions between reactive agents. Every visible object, from decoration artifacts to living characters, is an agent. Each agent is driven by motors, which are also agents with the same structure and run concurrently (Fig. 39). Agents have a sensory center, which has the basic function to handle messages and perceptions. An agent is activated by a message sent by its parent-agent and the tasks that an agent receive are usually distributed to the agent motors. Examples of languages that support the notion of agents composed of sub-agents are SARL, CLAIM, and MINERVA.

B.2 MAS LANGUAGES, PLATFORMS, AND CONCURRENCY

This section has the aim to briefly present some works that provide other concurrency features in order to exploit concurrency in the

MAS execution. Sec. B.2.1 presents the works developed by Clark and colleagues. Sec. B.2.2 presents the works developed by Ricci and colleagues. Sec. B.2.3 presents the works developed by Bordini and Hübner. Sec. B.2.4 briefly presents some tuning performed by Muscar and colleagues in agent languages. Finally, Sec. B.2.5 presents the works developed by Pokahr and colleagues. Along the presentation of all these works, other related work may be cited, especially because several agent languages provide similar features.

B.2.1 Clark et al.

Clack and his colleagues have proposed a couple of languages and extensions, always considering interesting aspects of concurrency. Although their languages are not only aimed to be used by MAS developers to program agents (i.e., they are multi-paradigm languages), the concurrency features adopted by the languages bring inspiration for the agent paradigm. In Qu-Prolog, UEs have an unique name and behave as communicating processes, which have a single message buffer of unread messages. A Qu-Prolog agent can be implemented as one or more cooperating UEs. UEs can communicate by means of UE-to-UE (e.g., thread-to-thread) store and forward communication system, or by a publish/subscribe mechanism. The number of UEs dedicated to communication depends on the number of conversations that are running in the MAS. If an agent is exchanging messages with two other agents, that agent will own two conversational UEs. Each conversational UE accesses and updates a shared belief store. These conversational UEs can also be suspended to wait a reply or if they want to read some message and the buffer is empty, resuming again when some message arrives. Moreover, UEs can be designed to wait for any other kind of change before continuing with their executions. When it is necessary to execute something, like a query, it is also possible to fork another UE to execute the query. The UE terminates when the query is accomplished.

In QuP++, the authors extend Qu-Prolog by means of object-oriented concepts. They also define agents as distributed objects and propose a distributed object-oriented logic programming language in which each object is a collection of UEs. The different behaviors (i.e., intentions in the BDI agents) of the agents execute concurrently, as separated UEs of an active object. QuP++ allows inter-UE communication between Qu-Prolog UEs running in any part of the internet. The agent interface UE can concurrently respond to queries from other agents and

the interface UE can fork a new UE for each received query. Moreover, each new UE can also be a new active object. This last feature is quite similar to what exists in some actor languages (AGHA, 1986), such as Erland (ARMSTRONG, 2007).

In (CLARK; MCCABE, 2003, 2004), the authors propose a multi-paradigm language called Go!. As in Qu-Prolog and QuP++, each agent in Go! is composed of several UEs. UEs execute action procedures, calling functions, and querying relations. UEs can communicate and coordinate their activities by means of asynchronous UE-to-UE messages, in the case of UEs that belong to different agents. Each UE can directly communicate with UEs in other agents and if UEs belong to the same agent, they can also communicate by means of a shared object. The updates of these objects are atomic and UEs also can suspend themselves until certain information be available in the shared object. Go! preserves other aspects of Qu-Prolog and QuP++. For example, UEs can spawn new UEs and these new UEs have their own message buffer and identity. Finally, Go! is a flexible language and does not directly support any specific agent architecture or methodology. The desired agent architecture can be developed by using library modules.

B.2.2 Ricci et al.

Ricci and his colleagues also propose a couple of languages and extensions, always considering aspects to exploit concurrency. In (RICCI; VIROLI, 2007; RICCI; VIROLI; PIANCASTELLI, 2011), the authors propose simpA, an extension for the Java language, which uses agent-oriented abstractions to organize the applications in terms of agents and artifacts. The aim is to introduce higher-level abstractions to help building concurrent programs in the same way as object-oriented abstractions help building large programs with many components.

Agent activities, which can be mapped to the concept of intention in the BDI model, in simpA can be either atomic (not composed of sub-activities) or structured, composed of some kinds of sub-activities in an hierarchical way. simpA introduces the concept of *agenda* to specify the activities and sub-activities. If some structured activity is executed, the sub-activities are also executed as soon as their pre-conditions hold. Multiple sub-activities can be executed concurrently. For example, two sub-activities $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ can be executed concurrently after activity $\boldsymbol{\gamma}$, and an activity $\boldsymbol{\delta}$ is only executed after the completion of both sub-activities $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.

simpA avoids some usual concurrency problems. For example, on the agent side, no race conditions can occur in updating or inspecting the *agenda* even considering that agents can have multiple concurrent activities. The agent internal actions are atomic. Agents also have timeout mechanisms to avoid waiting for certain events forever.

In (RICCI; SANTI, 2012) the authors deal especially with reactivity issues by proposing the simpAL language, an extension of simpA. simpAL integrates reactive and autonomous behavior. Differently from actors and objects that are only activated when they need to process some message that arrive, an agent has an autonomous behavior that starts processing as soon as it is created. Two main families can be identified integrating object with agent: (i) the reactive components (these approaches follow the reactivity principle, as actors); and (ii) autonomous active objects (the active entity may compute before some message arrive).

simpAL implements a classic pool-based strategy to improve concurrency. The concurrency is totally logical and the developer never suspends or acts upon UEs directly. There is not one UE for each agent, but all agents (and also artifacts) are executed by a pool. The size of the pool depends on the number of PEs available, which means that if the underlying hardware provides five PEs, there will have five UEs in the pool.

In (RICCI; SANTI, 2013; SANTI; RICCI, 2013), the authors propose ALOO, an extension of simpAL, which is conceived to address the problem of integrating active entities (actors, active objects, processes) with plain passive objects. ALOO conceives a MAS as an organization of task-driven autonomous agents that work cooperatively inside a shared environment composed of a set of passive objects that the agents can use, observe, and create in a concurrent and safe (race-free) manner. Tasks are a key concept in ALOO and are represented by objects. An agent is created with the reference to the object that represents the task to do. A task is considered cooperative when multiple agents are created with the same task object. In addition, ALOO provides some predefined tasks based on some relevant concurrent programming patterns and agent-oriented organizational schemes. Some of them are the producer-consumer, master-worker, pipeline, fork and join join, map-reduce, and contract-net. When executing an action, an agent can decide if it waits or not for the completion of the action. A key point in ALOO is that if an agent is waiting for the completion of an action, it still owns its UE of control, which permits the agent to react to events. However, internal actions of the agents are executed atomically and can

not be interrupted.

Plans in ALOO encapsulate the strategy to accomplish some specific kinds of task. A plan can decompose a task in sub-tasks and instantiate new tasks to run concurrently. Concurrency in ALOO agents means that for every plan in execution during each execution cycle, actions are selected and executed. In addition, agents do not interact directly, but they interact by acting on shared objects. The direct communication can be simulated by using objects such as mailboxes or channels.

B.2.3 Bordini and Hübner

Jason is an interpreter for an extended version of AgentSpeak(L) and implements the operational semantics, also providing a platform for MAS development. Related to concurrency, Jason has features for both, MAS level and intra-agent level concurrency. In the MAS level, Jason is configured to use one UE for each agent, which means that each agent has its own UE. The MAS developer can also explicitly configure a pool and the number of UEs that will be used to execute all agents in the MAS. Each UE from the pool selects *one* agent from the pool queue and executes *one* reasoning cycle.

In the intra-agent level, intentions run concurrently even without a dedicated UE for each one. Its interpreter manages the intention execution and executes *one* deed of *one* selected intention in each reasoning cycle execution, as detailed in Sec. 2.1.2. Another way to manage the intentions execution in Jason is to use the already provided internal actions, where one can specify when intentions need to be dropped, resumed, or suspended, or even check the current state of an intention (e.g., if the intention is suspended or not).

Finally, Jason lets the MAS developer to customize some aspects of the agents by means of extending some default classes in its implementation. For example, the MAS developer can customize the interaction with the environment (act and sense) and other agents (send and receive messages). Several aspects related to BDI can be customized, such as the way that agents select the messages, events, and intentions that will be handled in the current reasoning cycle. One could implement a priority mechanism to manage the selection of intentions from the intentions queue. Some languages that already support the definition of priorities for plans or let the agents to focus on some specific activities are ConGolog, IndiGolog, AgentSpeak(RT), N-2APL, GOAL,

and JIAC.

B.2.4 Muscar et al.

Muscar and his colleagues deal with several aspects of concurrency in agents. First of all, in (MUSCAR, 2011; MUSCAR; BADICA, 2011), the authors characterize how agents are usually executed and analyze two main strategies to manage the agents execution: (1) each individual agent can be executed using its own UE; (2) a pool can be used to execute all agents of the MAS. An experiment is performed to compare both strategies. The scenario adopted for the experiment is a distributed version of the Depth-First Traversal algorithm presented in (SANTORO, 2006) and the results demonstrated advantages of adopting a pool in the scenario of the experiment.

In (MUSCAR, 2013b, 2013a; MUSCAR; BADICA, 2014), the authors propose Blueprint, an agent language that is inspired in the asynchronous programming model adopted in F# (SYME; PETRICEK; LOMOV, 2011). Besides the adoption of a pool to execute the agents and allows the concurrent execution of the agents intentions, the authors adopt the concept of *promises* to express concurrent flows in the execution of intentions in a more natural way. The main problem addressed by their approach is when agents have different intentions executing concurrently and these intentions are required to be accomplished to let the agents to make progress with their executions (e.g., a join). In most agent languages, the only way to synchronize the execution of these different intentions is to use the belief base. The main disadvantage of using the belief base to synchronize the execution of intentions is to mix knowledge representation with operational aspects (i.e., condition variables). Thus, the authors propose the use of promises as a way to address this issue.

Promises are objects that represent the (yet unknown) results of an ongoing computation which is executing concurrently with other computations in the system. Promises can have callbacks attached which will be called when the value of the promise becomes available. The aim of promises is to make it possible to easily compose asynchronous computations, which offers advantages for real world scenarios where agents need to use resources that imply latencies (e.g., web services). Promises is one mechanism to let the developer to implement fork and join situations in plan in a more natural way.

B.2.5 Pokahr et al.

Jadex, an extension of JADE, uses a PRS based reasoning cycle for their agents. Plans are executed step-by-step however the length of the plan step depends on the context, and not only on the plan itself. Jadex offers different internal architectures to build the agents. They call these architectures as *kernels*. The *BDI-kernel* supports the development of complex reasoning agents that follow the belief-desire-intention model. The *micro-kernel* is a kernel that provides a simple programming style and supports the execution of a large number of agents (e.g., to build agents like insects). The *task-kernel* is a middle-term between the other agent kernels in terms of programming constructs and memory consumption, which is more suitable for agents performing a fixed set of tasks. In order to execute workflows modeled in the business process modeling notation (BPMN) it can be used the *BPMN-kernel*. In addition, they also provide a kernel called *GPMN-kernel* to interpret the goal process modeling notation, which is a unification of BDI agents and BPMN process concepts (BURMEISTER et al., 2008).

In (POKAHR; BRAUBACH; JANDER, 2010), the authors propose a new approach called active components. Active components are also autonomous, like agents, and owns their own UEs. They are also manageable entities, which exhibit clear interfaces making their functionality explicit, like software components. Three different paradigms for complex distributed systems are unified: active objects (LAVENDER; SCHMIDT, 1996), agents, and components. An active component is determined by its internal architecture while the structure may include a hierarchical decomposition into sub-components. Any component can contain an arbitrary number of child components and each child component is concurrent to all other entities. Finally, Jadex provide several methods in its class `Plan` to perform operations on intentions, such as suspend, resume, wait for conditions, or get its current state. Similar operations are also provided by JADE in its `Behaviour` class. In addition, by means of JADE it is also possible for the developer to decide when to adopt a specific UE to execute some behavior. JADE provides the *ThreadedBehaviourFactory* class that can wrap a normal JADE behavior into a threaded behavior wrapper (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

B.3 CONCURRENCY IN MAS APPLICATIONS

In (LEE et al., 1999; ZHENG; HOU; YIN, 2006), the authors present some real applications of multi-threading and agent technologies where there is a need to execute concurrent intentions. The execution of concurrent intentions impacts on the performance and reliability of some applications. In (LEE et al., 1999), the authors deal with a problem in the context of telecommunications management networks and propose a multi-threaded process for an agent system which can perform Common Management Information Service operations (CMIS) (CCITT Recommendation X.710, 1991) efficiently. Whenever an event (e.g., CMIS request, event report, timer interrupt) occurs, a new UE is created to handle it. The main aim is to execute CMIS operations concurrently. In order to do that, an coordinator UE is provided and it is responsible for creating other UEs to handle each CMIS operation separately. Thus, the agent never gets blocked and it is an important feature when an agent receives a critical request, otherwise the agent could not respond to other requests while already performing something.

In (ZHENG; HOU; YIN, 2006), the authors focus the use of agents and multi-thread technology on transformer substation automatic systems. Their aim is to strengthen the intelligence, real-time, safety, and extensibility of the system. The most important issues in their work are the communication and the several activities that need to be performed concurrently. For example, the system has to collect the data information constantly and deal with different operations according to the data. The agents need to monitor several ports, collect the data, and handle data in real-time. Most of these activities are handled by different UEs, improving the real-time responses and the reliability of the system. Several agent languages provide a way to launch each intention in dedicated UEs, such as ConGolog, IndiGolog, JACK, Ciao Prolog, GAEA, Jinni. Jinni also provides a way to spread the UEs in several computers, which is useful in scenarios where many intentions need to be performed at the same time. Other languages run each intentions using dedicated UEs automatically, such as $\boldsymbol{E_{hhf}}$.

In (CISCHKE, 2012; SAITO et al., 2012; TSUTSUI; FUJIMOTO, 2010; FERNÁNDEZ et al., 2010; FERNÁNDEZ-BAUSET et al., 2010; PÉREZ-CARRO et al., 2014), the authors work with simulations that demand a large amount of agents. In (CISCHKE, 2012), the authors state that the most common agent-based simulators are written using a multi-thread approach, and the use of a large number of agents is not possible, making impossible to take advantage of parallel computers. The authors pro-

pose KABS, a simulator that instead of providing one or more UEs for each agent, each UE is responsible for its own agents and its own portion of the environment. Thus, each UE is responsible for a group of agents and part of the environment. Likewise, in (SAITO et al., 2012), an agent-based simulator for the influenza spread is provided, which will require a large amount of agents (order of millions). The experiments performed by the authors verify the speedup of the system by using more UEs in relation to a single UE for the application. The result was the saturation of the MAS after using a certain number of UEs, which means that no speedup was being provided by increasing more the number of UEs. In (FERNÁNDEZ et al., 2010; FERNÁNDEZ-BAUSET et al., 2010; PÉREZ-CARRO et al., 2014), the authors are also interested in exploiting other aspects of the Java language in order to get better scalability without a big performance degradation. Thus, the authors propose some tunings in the heap size and in the garbage collection of Java Virtual Machine.

In (TSUTSUI; FUJIMOTO, 2010), the authors parallelize the execution in ant colonies. Parallelization is performed at the intra-agent level. Operations for each agent are performed concurrently in one colony and a set of operations for an agent is assigned to a specific UE. Since the number of agents (`m`) can be large, the proposed approach generates `n` UEs to be shared among the `m` agents, where `n` corresponds to the number of PEs.

APPENDIX C – Integration with Jason

As our proposal in Appendix 4 is inspired in the Jason platform, its integration in the Jason platform was quite direct. This appendix only highlights the most important aspects of the integration and the added constructs and changes in the language. Sec. C.1 presents the extensions in Jason considering the MAS and agent levels. We highlight the main changes in the reasoning cycle and how to configure a MAS execution. Sec. C.2 presents the extensions related to the intention level. We present the main features integrated in the Jason platform to exploit concurrency in the intentions execution.

C.1 EXTENSIONS IN THE MAS AND AGENT LEVELS

Jason(P) supports two new options to execute the reasoning cycle: the asynchronous execution (Algorithm 4) and an extension of the synchronous execution to execute only one stage of the reasoning cycle every time that a thread selects an agent, allowing that the three stages are executed in different moments (Algorithm 3). New parameters were introduced to configure how the reasoning cycle should be executed. These parameters permit to specify the number of times that the reasoning cycle and its stages are executed by a thread before the execution moves to another agent or another stage. The default job adopted in Jason for executing the reasoning cycle (Algorithm 21), is extended to Algorithm 2. Likewise, the execution of the sense (Algorithm 7), deliberate (Algorithm 11), and act (Algorithm 13) stages are explicitly executed separately and contain the parameter of the number of cycles for all possible configurations of the reasoning cycle. The parameters for the synchronous reasoning cycle are detailed in Sec. C.1.1 while the parameters for the asynchronous reasoning cycle are detailed in Sec. C.1.2. Finally, Sec. C.1.3 presents parameters to better adjust the reasoning cycle according to the needs of each individual agent.

sense()
deliberate()
act()

Algorithm 21: Traditional execution of the reasoning cycle in Jason.

C.1.1 Configuring a Synchronous Reasoning Cycle

Considering the centralized infrastructure of the application, by default, each Jason(P) agent has one thread, which executes the reasoning cycle synchronously and cyclically, as specified by the Algorithm 2, where each stage of the reasoning cycle is executed once and the number of times in which the sequence sense-deliberate-act is executed is infinity (i.e., until the agent termination). One can change the number of times that a stage executes by changing the values of the parameters:

```
Centralised(threaded, <NUMBER_CYCLES_SENSE>,
                      <NUMBER_CYCLES_DELIBERATE>,
                      <NUMBER_CYCLES_ACT>)
```

The keyword `threaded` informs the platform to create one thread for each agent in the MAS; `<NUMBER-CYCLES-SENSE>` is the maximum number of times that the sense stage is executed before the deliberate stage starts its execution; `<NUMBER-CYCLES-DELIBERATE>` is the maximum number of times that the deliberate stage is executed before the act stage starts its execution; and `<NUMBER-CYCLES-ACT>` is the maximum number of times that the act stage is executed before the sense stage starts its execution. For example, `Centralised(threaded,1,1,5)` means that the sense and deliberate stages will be executed only once, while the act stage will be executed at most 5 times. If `9999` is informed for the act stage, then, at least one action of each intention will be executed in the act stage. We adopted `9999` as a key value because it is quite high, and its usage in its literal meaning (i.e., to execute the act stage 9999 times before to sense and deliberate) would certainly harm the reactivity of an agent to react to changes in the environment, messages from other agents, etc, once the act stage would take a quite long time to be executed. This is clearly not a behavior expected for the execution of an agent.

A second way to execute the synchronous execution is to use thread pools. Two different combination of parameters can be used to configure this option:

```
Centralised(pool, <NUMBER_THREADS>,
                  [NUMBER_REASONING_CYCLES])
```

and

```
Centralised(pool, <NUMBER-THREADS>,
                  <NUMBER-CYCLES-SENSE>,
                  <NUMBER-CYCLES-DELIBERATE>,
                  <NUMBER-CYCLES-ACT>,
                  [NUMBER-REASONING-CYCLES])
```

The keyword `pool` informs the execution platform to create one thread pool with `<NUMBER-THREADS>` threads. `[NUMBER-REASONING-CYCLES]` is the maximum number of times that the sequence sense-deliberate-act is executed. In both combination of parameters, the reasoning cycle is executed like in Algorithm 2. For example, `Centralised(pool,4,1,1,5,10)`, means that a thread pool with 4 threads will be created, the sense and deliberate stages will be executed just once, the act stage will be executed at most 5 times, and the sense-deliberate-act sequence will be repeated 10 times.

In order to configure the reasoning cycle to execute synchronously and execute only one stage every time that a thread selects an agent, such as in Algorithm 3, the parameters are almost the same as before. The changes are in the first parameter, which need to be defined as `synch_scheduled`, and the parameter `[NUMBER-REASONING-CYCLES]` is not supported. For example, `Centralised(synch_scheduled,4,1,1,5)` means that a thread pool with 4 threads will be created, the sense and deliberate stages will be executed once, while the act stage will be executed at most 5 times, and only one of these stages are executed when a thread selects an agent.

C.1.2 Configuring an Asynchronous Reasoning Cycle

The asynchronous reasoning cycle can be configured to use a single thread pool to execute all the stages or to use one dedicated thread pool to execute each stage. The jobs that are executed by the threads in the asynchronous reasoning cycle are the same as presented in Algorithm 4. Two different combination of parameters can be used to execute the reasoning cycle asynchronously with a single thread pool to execute all the stages:

```
Centralised(asynch_shared, <NUMBER-THREADS>,
                           [NUMBER-REASONING-CYCLES])
```

and

```
Centralised(asynch_shared, <NUMBER-THREADS>,
                           <NUMBER-CYCLES-SENSE>,
                           <NUMBER-CYCLES-DELIBERATE>,
                           <NUMBER-CYCLES-ACT>)
```

The keyword `asynch_shared` makes the execution platform to create a single thread pool with `<NUMBER-THREADS>` threads. For example, `Centralised(asynch_shared,4,15,15,20)` means that a thread pool with 4 threads will be created, the sense and deliberate stages will execute at most 15 times, and the act stage will be executed at most 20 times.

Likewise, two different combination of parameters can be used to execute the reasoning cycle asynchronously with a different thread pool to execute each stage:

```
Centralised(asynch, <NUMBER-THREADS-SENSE>,
                    <NUMBER-THREADS-DELIBERATE>,
                    <NUMBER-THREADS-ACT>,
                    [NUMBER-CYCLES])
```

and

```
Centralised(asynch, <NUMBER-THREADS-SENSE>,
                    <NUMBER-THREADS-DELIBERATE>,
                    <NUMBER-THREADS-ACT>,
                    <NUMBER-CYCLES-SENSE>,
                    <NUMBER-CYCLES-DELIBERATE>,
                    <NUMBER-CYCLES-ACT>)
```

The keyword `asynch` makes the execution platform to create three thread pools, one for each stage. `<NUMBER-THREADS-SENSE>` is the number of threads for the thread pool to execute the sense stage. `<NUMBER-THREADS-DELIBERATE>` is the number of threads for the thread pool to execute the deliberate stage. `<NUMBER-THREADS-ACT>` is the number of threads for the thread pool to execute the act stage. For example, while `Centralised(asynch,4,4,4,15)` means that three thread pools with 4 threads will be created and each stage will be executed at most 15 times, `Centralised(asynch,4,4,4,15,15,20)` means that three thread pools with 4 threads will be created, the sense and deliberate stages will execute at most 15 times, and the act stage will be executed at most 20 times.

```
ana [cycles_sense = 2,
     cycles_deliberate = 2,
     cycles_act = 10];
```

Figure 40 – Configuring the agent `ana`.

```
bob [cycles = 10];
```

Figure 41 – Configuring the agent `bob`.

C.1.3 Configuring Agents Individually

Besides a global configuration for the MAS, the number of cycles for each agent can be specified individually. The parameters for the agents are presented by means of the two examples. In the first example (Fig. 40), the parameters of the agent `ana` mean that the number of cycles for the sense and deliberate stages is 2, while the number of cycles for the act stage is 10.

In the second example (Fig. 41), the parameters of the agent `bob` mean that the number of cycles for the sequence sense-deliberate-act is 10. Of course, some architectures of agents do not allow the parameter `cycles`, such as the asynchronous execution once the sequence sense-deliberate-act is not supported.

C.2 EXTENSIONS IN THE INTENTION LEVEL

This section presents the new features supported by the Jason(P) platform related to the intentions execution. While Sec. C.2.1 presents how conflicting intentions can be specified, Sec. C.2.2 presents how to specify fork and join in plans.

C.2.1 Specifying Conflicts in Jason(P)

Broadly speaking, the support of conflicting intentions was integrated in the Jason(P) platform by considering the algorithms proposed in Sec. 4.3.1. The syntax of Jason was not modified and we just added semantics for new keywords. Plans can have an annotation called `conflict`, where the developer informs all conflicts related to the plan. We propose three options to inform the conflicts, which can be combined in order to give more flexibility for the developer. The developer can

```
@p1[conflict(["@p2", "@p3"])]
+!g1 <- ...

@p2[conflict(["@p1"])]
+!g2 <- ...

@p3[conflict(["@p1"])]
+!g3 <- ...

@p4[atomic]
+!g4 <- ...
```

Figure 42 – Conflicts specified in Jason(P).

inform (1) the specific plan names that conflict, (2) the events or goals that conflict with a plan, or (3) a common identifier of the conflict (e.g., all plans that use the resource $\boldsymbol{R}$ will have informed the same conflict identifier, such as $\boldsymbol{control\_R}$). In practice (i.e., at run-time), options (2) and (3) are reduced to option (1), once the events, goals, and conflict identifiers can be replaced by the corresponding plan names. Finally, when a plan conflicts with all other plans, then a plan can be specified as *atomic* by using the keyword `atomic` or the expression `conflict(_)`.

Assuming that the plan names in Jason(P) start with `@`. Events start with the symbols `+` or `-`. The symbol `+` is used when goals are adopted (e.g., `+!g` means the adoption of the goal `!g`) or new beliefs are added (e.g., `+b` means the addition of the belief `b`). The symbol `-` is used when goals are dropped (e.g., `-!g` means the removal of the goal `!g`) or beliefs are removed (e.g., `-b` means the removal of the belief `b`).

Figure 42 presents examples of skeletons of plans in Jason(P)and their conflicts. While `@p1`, `@p2`, `@p3`, `@p4` are the names of the plans, the lists next to them allow the specification of their conflicts. For example, `@p1[conflict(["@p2", "@p3"])]` means that `@p1` conflicts with `@p2` and `@p3` (and vice versa). We can also inform the conflicts with events, such as `@p5[conflict(["+!g6", "+b"])]`, which means that `@p5` conflicts with all plans to achieve the goal `!g6` and all plans that handle the belief inclusion of `b`.

Besides the specification of the conflicts by explicitly indicating the events or plan names in the CS of the plans, conflicts can be specified by a conflict identifier. For example, a robot can have some critical resources (e.g., wheels) that is used by different plans. If some plan is using the wheels, a conflict with the identifier $\boldsymbol{wheel\_control}$ can be created. Thus, if $\boldsymbol{p1}$ and $\boldsymbol{p2}$ are conflict plans, they can be written without referring to each other, and they could be even added in the plan library in different moments. Plans do not need to refer to the

```
@p1[conflict([conflict_1, conflict_2])]
+!g1 <- ...

@p2[conflict([conflict_1])]
+!g2 <- ...

@p3[conflict([conflict_2])]
+!g3 <- ...

@p4[atomic]
+!g4 <- ...
```

Figure 43 – Conflicts specified by means of a conflict identifier.

```
@p1[conflict([@p4])]
+!g1 <- !g3 ...

@p2[conflict([@p3])]
+!g2 <- !g4 ...

@p3
+!g3 <- ...

@p4
+!g4 <- ...
```

Figure 44 – Deadlock among conflicting plans.

existence of other plans or goals explicitly as well as the developer does not need to care about the existence of other plans. When a new plan is added to the plan library, the developer simply specifies which are the conflict identifiers related to the plan. The examples presented in Fig. 42 can be rewritten using to specify conflicts (Fig. 43).

At run-time, the conflict identifiers are mapped to the corresponding plan names. Thus, while `conflict_1` is replaced by `@p2` in the CS of ***p1*** and replaced by `@p1` in the CS of ***p2***, `conflict_2` is replaced by `@p3` in the CS of ***p1*** and replaced by `@p1` in the CS of ***p3***. In order to specify a conflict of a plan with itself, we created a reserved conflict identifier named `self`, so that the developer can let clear that two instances of the same plan can never be executed concurrently.[1]

One important issue in our approach to specify conflicts is the possibility to have deadlocks at run-time depending on how the conflicts are specified. An example of conflicts that may lead to a deadlock is presented in Code 44. If `@p1` and `@p2` are in execution, as soon as `@p1` adopts the goal `!g3`, it gets suspended because `@p2`, which conflicts with `@p3`, would not allow `@p3` to be executed. Likewise, as soon as `@p2`

[1]The conflict of a plan with itself can be also specified by adding its own plan name in its CS (e.g., `@p1[conflict(["@p1"])]`).

adopts the goal `!g4`, it also gets suspended because `@p1`, which conflicts with `@p4`, would not allow `@p4` to be executed, even if `@p1` is already suspended. An extensive discussion can be done around this issue, however we do not address this on this thesis. The deadlock problematic and suggestions to solve deadlocks in agent programming are also presented in (THANGARAJAH; PADGHAM; WINIKOFF, 2003a; THANGARAJAH et al., 2002; THANGARAJAH; PADGHAM; WINIKOFF, 2003b).

C.2.2 Specifying Fork and Join in Plans

Besides the usual sequence operator `;`, the Jason language was extended to support the use of the fork and join operators in the plan bodies: `|&|` and `|||`. The former is a fork with a join-and and the latter is a fork with a join-xor. The semantics of these operators adopted in Jason(P) is the same as presented in Sec. 4.3.2.

Figure 43 presents an example of fork with join-and, where both `!g3` and `!g4` need to be achieved before `a1` is executed. Figure 46 presents an example of fork with join-xor, where the achievement of either `!g3` or `!g4` is enough to allow `a1` be executed.

Regarding the precedence, the operator `|&|` has precedence over `|||` which has precedence over `;`. In a plan like in Fig. 47, the execution is as follows: execute `x`; concurrently execute `a;b`, `c;d`, and `e;f`; and execute `y` when either `a`, `b`, `c`, and `d` have finished, or when both `e` and `f` have finished.

```
+!g1 <- ...; !g2; ...
+!g2 <- ...; (!g3 |&| !g4); a1; ...
```

Figure 45 – Fork with join-and.

```
+!g1 <- ...; !g2; ...
+!g2 <- ...; (!g3 ||| !g4); a1; ...
```

Figure 46 – Fork with join-xor.

```
x; (a;b) |&| (c;d) ||| (e;f); y
```

Figure 47 – Precedence of fork and join.