Department of Computer Science, University of Liverpool, UK

Contact: Richard Stocker (R.S.Stocker@liverpool.ac.uk)

# Towards the Formal Verification of Human-Agent-Robot Teamwork

A PhD Thesis
By
Richard Stocker

*Supervisors:*
Prof. Michael Fisher
Dr. Louise Dennis
Dr. Clare Dixon

*Reviewers:*
Dr. Wamberto Vasconcelos
Dr. Davide Grossi

June 6, 2013

Work funded in the UK through EPSRC grant EP/F033567
Verifying Interoperability Requirements in Pervasive Systems.

# Declaration

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere else for any other award. Where other sources of information have been used, they have been acknowledged.

| | |
|---|---|
| Signature: | ....................................... |
| Date: | .................... |

# Acknowledgements

I would like to express my deepest appreciation to Prof. Michael Fisher, Dr. Clare Dixon, and Dr. Louise Dennis for their supervision of this thesis. Michael's ability to explain the more difficult concepts in easy to read diagrams and his foresight of the advantages and disadvantages of the possible paths we could take made the project a lot less difficult to achieve. Louise's technical knowhow of how to implement the tool we needed was of invaluable help. I also thank Louise for her uncanny ability in discussions to act as an interpreter between myself and Michael when I could not understand a question he asked or he could not understand a poorly explained answer I had given. Clare's assistance for the theoretical aspects of the project was of immense help, providing guidance on how to structure the semantic rules. Clares advice on a multitude of areas, from technical and theoretical aspects to simple formatting of documents greatly improved every aspect of the thesis. Without the guidance and help of Michael, Clare and Louise this thesis would not have been possible.

I would like to thank Prof. Maarten Sierhuis for his assistance with the project. His understandings and explanations of the Brahms framework made it possible for us to create a formal operational semantics for Brahms. Also his guidance on applications of Brahms helped us to create the case studies for the thesis, most notably the digital nurse scenario.

Finally I would like to thank Dr. Neha Rungta and Dr. Franco Raimondi for their assistance with this thesis. Their willingness to use the operational semantics we created and to involve me in the development of their model checking process for Brahms validated the work we have done and gave me the confidence I needed to complete this thesis. Also, to include me as a co-author for their AAMAS paper was a great honour which I greatly thank them for.

# Contributions to Learning in Computer Science

This thesis:

- presents the first formal semantics for Brahms; Part II, Chapter 6, published in [79]
- presents the first formal verification tool specifically for human-agent teamwork; Part II, Chapter 7, published in [78]
- presents the first formal verification tool for the Brahms simulation framework; Part II, Chapter 7, published in [78]
- presents our aided construction of another formal verification tool for the Brahms simulation framework; Part II, Chapter 12, published in [71]

# Thesis Statement/Abstract

The formal analysis of computational processes is by now a well-established field. However, in practical scenarios, the problem of how we can formally verify interactions with humans still remains. This thesis is concerned with addressing this problem through the use of the Brahms language. Our overall goal is to provide formal verification techniques for human-agent teamwork, particularly astronaut-robot teamwork on future space missions and human-robot interactions in health-care scenarios modelled in Brahms.

# Thesis Structure

This thesis is structured into 4 Parts and 13 Chapters as follows:

## Part I

Introduces the thesis and provides information on the tools and techniques used.

**Chapter 1**

introduces the thesis providing justification for undertaking the project.

**Chapter 2**

explains to the reader our concept of what an agent is, how it is used and the representation of agents in teams with both humans and other agents.

**Chapter 3**

introduces the agent simulation tool we use in this thesis (Brahms), explaining its core functions and describes how it compares to other agent programming languages.

**Chapter 4**

explains our meaning of formal verification, introducing different types of formal verification, logics and agent verification.

**Chapter 5**

describes our analysis of how to conduct this project, from selecting the tools we used to the methods we employed.

## Part II

Explains the development of the tool we created to formally verify human-agent teamwork. Introducing our formal operational semantics for Brahms, the implementation of our model checking tool and another tool to formally verify Brahms using our operational semantics.

**Chapter 6**

describes the formal operational semantics produced for this thesis.

**Chapter 7**

describes the process of translating Brahms code into *PROMELA* for Spin verification.

**Chapter 8**

describes NASAs Brahms verification tool developed using the formal operational semantics produced for this thesis.

## Part III

Details the case studies produced to test the correctness, effectiveness and performance of our tool.

**Chapter 9**

describes the home helper scenario, where a robotic helper assists a person with dementia.

**Chapter 10**

describes the digital nurse scenario, where doctors and nurses have digital assistants to help them treat patients.

**Chapter 11**

brings together all the conclusions formed from the analysis of the case studies.

## Part IV

Contains our evaluation and conclusions of the tool produced. Here we also describe other methods used to evaluate the performance of the tool and their results.

**Chapter 12**

evaluates the performance of the verification tool on simple scalable tasks.

**Chapter 13**

brings together all the conclusion made during this thesis.

# Contents

# Part I

# Introduction and Background

# Chapter 1

# Introduction

Computational devices often need to interact with humans. These devices can range from mobile phones or domestic appliances, all the way to fully autonomous robots. In many cases all that the users care about is that the device works well most of the time. However, in mission critical scenarios we clearly require a more formal, and consequently much deeper, analysis. For example, as various space agencies plan missions to the Moon and Mars which involve robots and astronauts collaborating, then we surely need some form of *formal verification* for astronaut-robot teamwork. This is needed at least for astronaut safety (e.g., “the astronaut will never be out of contact with the base”) but also for mission targets (e.g., “three robots and two astronauts can together build the shelter within one hour”). As autonomous devices are increasingly being developed for, and deployed in, both domestic and industrial scenarios, there is an increasing requirement for humans to at least interact with, and often work cooperatively with, such devices. While the autonomous devices in use at present are just simple sensors or embedded hardware, a much wider range of systems are being developed. These consist not only of devices performing solo tasks, such as the automated vacuum cleaners we see already, but are likely to include robots working cooperatively with humans. Examples are robot ‘helpers’ to assist the elderly and incapacitated in their homes [57, 65], manufacturing robots to help humans to make complex artefacts [52], and robots tasked with ensuring that humans working in dangerous areas remain safe. All these are being developed, many will be with us in the next 5 years, and all involve varying degrees of cooperation and teamwork.

What are the *challenges* facing such analysis? The first is: how can we accurately describe human, and indeed robot, behaviour? Even when we have described such behaviours, how can we exhaustively assess the possible interactions between the humans and robots? While some work has been carried out on the safety analysis of low-level human-robot interactions [62], a detailed analysis of the *high-level* behaviours within such systems has not yet been achieved.

In this thesis we are concerned with the general problem of matching a set of requirements (which could concern safety, capabilities, or interactions) against scenarios involving humans, robots, and software agents. Within this, we use important work on high-level modelling of human-robot-agent teamwork that has already been carried out using the Brahms framework [73]. Brahms is a simulation/modelling language in which complex human-agent work patterns can be described. Importantly for the purpose of this thesis, Brahms is based on the concept of *rational agents* and the system continues to be successfully used within NASA for the sophisticated modelling of astronaut-robot planetary exploration teams [21, 75, 74]. For information on the Brahms framework see

Chapter 3. Thus, for this thesis an assumption is made that the key interactions and behaviours of any human-robot-agent scenario have been captured within a Brahms model. Also an assumption is made that a set of informal requirements have been constructed.

However, how can we go about verifying requirements against Brahms models? How can we possibly verify human behaviour? And how can we analyse teamwork? In this thesis we aim to take a step forward by developing techniques to solve these problems.

The above examples discuss robots deployed in both domestic and safety-critical industrial situations where human safety can be compromised. Thus, it is vital to carry out as much analysis as is possible not only to maximize the safety of the humans involved, but to ascertain whether the humans and robots together 'can', 'should', or 'will' achieve the goals required of the team activity. In [11] a formal approach to the problem of human-agent (and therefore astronaut-robot) analysis has been proposed, suggesting the model-checking of Brahms models [73, 42]. Thus, it seems natural to want to formally verify Brahms models [11].

This thesis takes a first step in verifying human-agent teamwork using the Brahms framework. This first step is not concerned with being the most efficient with the widest range of functionality, but simply a prototype for others to learn from and even a platform for others to develop on. Also by specifying that we are working towards achieving this goal we are acknowledging the difficulty and complexity of this task, specifically in verifying human behaviour. Since human behaviour can be unpredictable and irrational there is no way to guarantee a model will ever represent a human in any given situation. However, by taking typical actions of a human - actions typical for that scenario - we can make an attempt to verify the protocols of the scenario given that the humans act in a rational way. Therefore, in this thesis we present our process and achievements while working towards achieving the verification of human-robot-agent teamwork. We present the reader with a description of our chosen simulation tool Brahms, along with a formal operational semantics [79] describing its behaviour and methodologies. Using this formal operational semantics we describe our translation process from Brahms input code to Java data structures and then to the input language *PROMELA* for verification via the Spin model checker. We evaluate the use and correctness of our tool using two case studies; a robotic home helper scenario and a digital nurse scenario in a hospital. An evaluation of our tools performance is conducted to demonstrate how it performs on simple scalable tasks, providing graphs and figures to show any strength or weaknesses and also the scalability of our tool.

This thesis provides contributions to the field of Computer Science through our formal operational semantics of Brahms, published in [79] and the translation from Brahms to Java data structures and then to *PROMELA* for verification via the Spin model checker [78]. Our operational semantics is the first of its kind for Brahms, which provides a foundation for the formal verification of Brahms. The operational semantics has not only proven a foundation for our verification tool but also for the verification tool of Franco Raimondi and Neha Rungta in [71]. The translation of Brahms models into Spin has given way to the first verification system for Brahms and for human-agent-robot teamwork.

# Chapter 2

# Software Agents

In this thesis we are concerned with analysing the interactions of humans, software agents and robots while operating together in a team. In this chapter we provide the reader with an overview of the concept of an agent. We discuss what it means to be an agent, how agents are used and programmed, how agents are grouped together in teams, how our view point of agents and humans differ, and the architectures used to model them.

## 2.1 Agents

Agents are a relatively new addition to computer science originating around 1980 but only coming into recognition during the mid-1990s [89]. There are differing definitions of an agent but in this thesis we consider agents to be entities which act autonomously to achieve self-interested goals. The large number of applications where agents can be applied results in different requirements of how they should act, such as robotic agents [35] and agents for intrusion detection [3] having very different requirements. Agents can also be viewed differently from different perspectives, such as artificial intelligence, databases, distributed computing and programming languages [43]. Agent learning is an example of a trait which may or may not be required; learning can improve the functionality of an agent but can cause it to develop unexpected behaviours [89]. The agent modelling tool used in this thesis, Brahms, specifically omits agent learning. This minimises unexpected behaviour and reduces the state space for analysing the agents' actions, see Chapter 4.2.

Brahms allows the user to explicitly describe the environment and the properties it has. For this reason it is important to consider how an environment can be represented. Environments can be [89]:

1. Accessible: if an environment is accessible then all the accurate, up to date and complete details about the state of the environment is available to the agent.

2. Deterministic: deterministic environments have no uncertainty about how an agent's actions will alter the state of the environment. Every time actions are performed they will have the same effect.

3. Static/Dynamic: A static environment is an environment where no changes occur unless an action is performed by an agent, whereas dynamic environment variables can change without the involvement of an agent.

4. Discrete/Continuous: A discrete environment has a limited number of actions, perceptions and effects that can occur, whereas in continuous are unlimited.

Brahms allows for all the possible environments listed above. The environments we will consider in this thesis will utilise most of these environment types, except for the continuous and deterministic environment types. Verification considers every set of possible actions so we are required to have a limited number of possible actions for the verification to complete. Determinism in an environment would mean verification is not required, since the outcome can be pre-determined before the simulation is run. Static environments are considered relevant to this thesis, however only the performance testing section utilises static environments. All the case studies use dynamic environments where emergencies occur and we verify the agent's actions and reactions. In such dynamic environments we expect our agents to be "reactive", "pro-active" and "social" [89]. By being "pro-active" the agents will operate on their own initiative and exhibit goal-directed behaviour, they will then be "reactive" to changes in these dynamic environments, and since they will be operating in teams they will need to be "social" and interact with each other to complete tasks efficiently.

## 2.2 Rational Agents

To be able to analyse the behaviour of an agent, we need to be able to explain and understand it. In this thesis we use the concept of a *rational agent* so that our agents are able to make their own choices and carry out actions in a rational and explainable way. Brahms represents rational agents by using concepts from the Beliefs, Desires, Intentions architecture (BDI) [85], see Chapter 2.4. This architecture allows the representation of the information, motivation and deliberative actions of the agents, enabling us to see the reasoning behind each action the agent makes. Rational agents are regarded as having their own agenda, which may concur or conflict with other agent's agendas [88]. They are situated in environments which may also have additional agents each with their own agendas. This makes rational agents desirable for teamwork; where both reactive and social behaviour is required to achieve the team's goals.

### 2.2.1 Applications

Rational agents are becoming ever more common in modern life especially in situations too dangerous, too difficult or too complex to employ humans [68]. These situations are often safety critical and require decisions to be made by autonomous entities that are independent, safe and explainable, i.e., rational. NASA's Deep Space One probe launched October 24th 1998 [59] is one example. It was the first NASA spacecraft to feature an on-board planner. This on-board planner formed an autonomy architecture which enabled high-level command, robust fault responses and opportunistic responses to serendipitous events. By employing rational agents in this spacecraft NASA was able to reduce mission costs and increase mission quality [58]. Before this automation NASA required a ground crew of up to 300 staff to continually monitor progress and make every decision [88]. This is one application of where rational agents can be employed, but there exist much simpler commonplace areas such as in e-commerce, buying/selling goods [88]; or expert systems, as an aid for general practitioners or nurses [44]. All these examples have some form of safety/business critical aspect, i.e., lives or large quantities of money are at risk. There is always doubt when employing an agent or a robot to control such situations. This is where formal verification becomes useful, by mathematically analysing the correctness of the system we can be more confident that the system will behave as expected.

### 2.2.2 Humans as Agents

In this thesis we take the view point that humans can be represented as complicated agents. We believe that humans in professional scenarios such as astronauts, nurses, people in search and rescue, etc. will act in rational and explainable ways, i.e., like a rational agent. This leads us to believe that if we can model human aspects into our agents then we can model both our humans and agents in the same way, except the agents will be more restricted than the humans. Such human aspects we would need to be able to represent are: taking breaks, making errors, forgetting things, taking varied amounts of time to perform tasks, etc.

## 2.3 Multi-Agent Systems

In this section we describe multi-agent systems to give an insight into the difficulties of handling multiple agents in an environment and the formation of teams. Multi-agent systems are very diverse systems which can be as simple as a single computer system with multiple software agents to a large distributed system where each agent is a computer system itself. A multi-agent system can also be a group of robots operating in a mutual environment, where the agent is the software which acts as the “mind” of the robot. It is possible that a single multi-agent system can contain agents designed by different individuals with differing goals. Agents can “team-up” with others to form coalitions, compete against each other for their preferred outcome, or perhaps act mutually exclusively to each other. Situations such as these mean that multi-agent systems can sometimes be represented as games where agents must act in a way which best suits either their own interests or the systems’. When agents work together multiple factors have to be taken into account: how to break down the problem, how to produce an overall solution from the sub-problems, how to maximize efficiency and how to avoid destructive clashes of activities [89]. When we have multiple agents working together or against each other in an environment a lot of uncertainties develop, such as: which agents will form coalitions, how an agent will react to anothers actions, will the agents coordinate their tasks, will any deadlocks or race conditions occur, will the agents choose the most optimal set of actions to complete their task, and will the agent react in a timely fashion to the actions of another. These uncertainties are additional to those present in a single agent system, e.g., will an agent successfully complete its task. These uncertainties can leave doubts on whether multi-agent systems should be implemented in a safety critical scenario. Therefore, tests and analyses, such as formal verification, need to be performed to ensure the safety of a multi-agent system before it can be employed within such scenarios.

### 2.3.1 Humans and Agents Working as a Team

Sierhuis et al. [75] note a growing interest in requirements for human-agent interaction, stating that: “many new space efforts are specifically motivated by the need to support close human-agent interaction”. Sierhuis et al. [75] and Bradshaw et al. [16, 14] mainly centre their discussions about human-agent teamwork around space related situations but there are many other applications such as: pervasive systems operating with humans in health care and at home [4]; simulation based training [81]; evacuation during an emergency [80]; and gamebots and players in video games [47]. In such scenarios humans and agents need to work together to form a team in order to effectively complete their tasks, but there are many challenges in achieving this. In [15] Bradshaw et al. focus on the problem of coordination in human-agent teamwork. Bradshaw et al's [15] aim is to give human-agent teamwork a richness of interaction and characterize natural and effective teamwork among groups of people. Techniques to over come the challenges are: the notions of joint activity [17] and joint intentions [24], and team plans [20].

Bradshaw et al. [17] started researching the notion of *joint activity* by studying how humans succeed and fail when taking part in an activity which requires a high degree of interdependence between participants. From this study Bradshaw et al. decided to focus on the issue of coordination and identifying the difficulty in representing and reasoning about humans in comparison to agents. The difficulties involved with coordinating humans and agents are one of the limitations of agents: only certain aspects of the world can be represented to the agents and the agents have a limited ability to sense or infer information from a human environment. Bradshaw et al. [17] identify three basic requirements for effective coordination:

- Inter-predictability
  - By being able to predict what others will do, you will be able to predict what you will be able to do yourself.
- Common Ground
  - Everyone has the same beliefs and assumptions about the activity and everyone knows that everyone has these beliefs and assumptions.
- Directability
  - Directability is being able to evaluate and modify others as the conditions and priorities of the activity change.

Cohen and Levesque [24] introduce a notion for agent collaboration they call "*joint intention*". The aim of a *joint intention* is to perform a collective action while in a shared mental state, meaning the agents will have shared beliefs and intentions while performing the activity. This notion of *joint intention* is based upon the beliefs-desires-intentions paradigm, meaning that when a collection of agents share the same beliefs and desires they will involve each other when creating their intentions so that they become *joint intentions*. This makes communication of paramount importance, by asking when the agents should communicate we are able to decide how a *joint intention* is formed.

Cavedon et al. [20] describe a notion called "*Team Plans*" for implementing team behaviour for autonomous agents. These team plans are used to implement team strategies. The team plan is created so that it specifies what subsets of abilities the agents require to take part in this team, the agents are then able to select and commit to tasks as if they were committing to single-agent plans. Once an agent joins the team it is assigned roles; a set of tasks an agent will execute to complete a part of the task. Even though each agent has its own roles within the task it still has the goal of achieving the whole plan. Team members that are committed to a *team plan* that fails can either try to resolve any failed sub-tasks or determine if the failed task is unsolvable or infeasible.

### 2.3.2 What is Teamwork?

Teamwork itself is a popular research topic, with researchers such as Dyer [31] and Salas [72], differing opinions on what teamwork is and how it is formed. A general description of teamwork could be that it involves 2 or more individuals working together to achieve a common goal. However this may or may not encapsulate everything that we associate with teamwork; would we consider 2 individuals sweeping the floors of 2 different branches of the same company as working in a team? After all they are both working for the same company and trying to achieve the same goal (i.e., maintain the company's level of hygiene) but they may never see or speak to each other. Some would argue that this is still teamwork, albeit very minimal teamwork. This level of teamwork is similar to what can be found in a generic multi-agent system, where agents have no *joint intentions*, *joint actions* or *team goals*. In this thesis we are taking the opinion that teamwork can be considered to be on a sliding scale; one extreme being a multi-agent system where agents own individual goals are closely linked (such as in the sweeping floors example), and the opposite end of the spectrum where the agents are involved in *joint actions* and with *joint intentions*. From this perspective we are only interested in a level of teamwork which is somewhere in the middle of the two extremes on this sliding scale between *joint intentions*, etc. and simply related activities. This is due to the way Brahms handles *joint intentions* and *joint activities*, see Chapter 3.2. The teamwork we are interested in involves communications and interactions to achieve goals of mutual interest, such as robotic helpers; where the robots' goals are to help the humans, and the humans' goals are to complete a task. For example the human will request an object or request a task to be done and the robot will comply. Another example would be a robot looking after a human, so the human has their own personal interests and the robot has the safety and well-being of the human as its interest. These levels of teamwork are not what could be considered to be very deep. For example a team of humans and robots playing a game of football together would involve much deeper levels of teamwork with constantly changing *joint intentions* and activities between team members with the common goals to score 'goals' and prevent 'goals' being conceded.

When verifying properties of teamwork we need to take into account the teamwork aspect of the scenario, so we need to look at how to measure a team's performance. Hexmoor and Beavers [38] suggest four properties for testing the effectiveness of teamwork:

1. Efficiency, use of resources by the team to achieve the goal, such as time or effort
2. Influence, how members affect the performance of others
3. Dependence, how the ability of a member to complete its task is affected by the performance of other members
4. Redundancy, the duplication of effort by distinct members

Consideration also needs to be made about the organisation of the teamwork, such as; roles, delegation of tasks, and the obligations of each team member. McCallum, Vasconcelos and Norman [56] highlight the importance of verifying roles, delegations and obligations, and produced a formal verification framework for organisational multi-agent systems. However, this framework concentrates on organisations of agents, with no direct consideration for human team members.

## 2.4 Beliefs-Desires-Intentions

In this section we describe the Beliefs-desires-intentions (BDI) architecture [69], which the agent modelling simulation framework we use (Brahms) is based on. In the BDI architecture agents have a set of beliefs (beliefs), a set of goals (desires) and a set of pre-compiled plans to achieve these goals based upon their beliefs (intentions). Generating plans according to the desires the agent wishes to achieve is known as means-ends-reasoning. These plans take into account the beliefs the agents have about their environment and what effect they believe their actions will have. BDI models are used to determine the behaviour of the agent and also to optimise the performance when in a resource-bound scenario.

BDI architectures use Beliefs, Desires and Intentions to represent an agent's mental model of information, motivation and deliberation. Beliefs represent what the agent believes to be true, such as the location of an object and the distance to that location. Desires are what the agent aims to achieve, these are usually represented as a desired set of beliefs, e.g., a desire to fill a glass with water would be achieved when the agent believes the glass is full. Intentions are how the agent aims to achieve its desires based on its current beliefs; effectively a list of actions or a plan to achieve its goal.

Figure 2.1 shows an architecture for resource-bounded agents demonstrating how BDI models will determine the actions of the agents. The diagram shows how perceptions form the agents' beliefs about its environment, reasoning about these beliefs is performed and how to discover what implications they have. Means-end reasoning decides which plans are best used to complete the tasks. Desires are shown in this model in order to influence the option filtering process and also influence the deliberation process to produce intentions. Actions are decided once the options have been filtered, the list of intentions has been generated and the intentions are structured into plans. The model also shows that previously generated plans influence future plans yet to be created.

The BDI architecture does have its criticisms [69]: classical decision theorists and planning researchers question the necessity of all three mental attitudes while sociology and distributed artificial intelligence researchers question the adequacy of using only three.

Brahms only takes inspiration from the BDI architecture, it does not follow all of its ideologies. Brahms uses the notion of beliefs for its agents but does not directly implement desires or intentions in the form of plans. Brahms operates over its beliefs using an activity based subsumption architecture (a structured layering of simple behaviours with a single goal per layer) of workframes (used to represent an agent's work process) which can be assumed to resemble plans. Guard conditions on these workframes match to the agent's beliefs, therefore these can effectively be considered as intentions. Figure 2.2 shows how Brahms relates to the BDI architecture, showing that the Brahms engine operates on and has an effect on all the beliefs, desires, intentions and plans of the agents. To explain this further we need to consider how thoughtframes and workframes represent the desires, intentions and plans. The thoughtframes and workframes guard conditions operate on the beliefs of the agent, thereby creating desires for the agent,

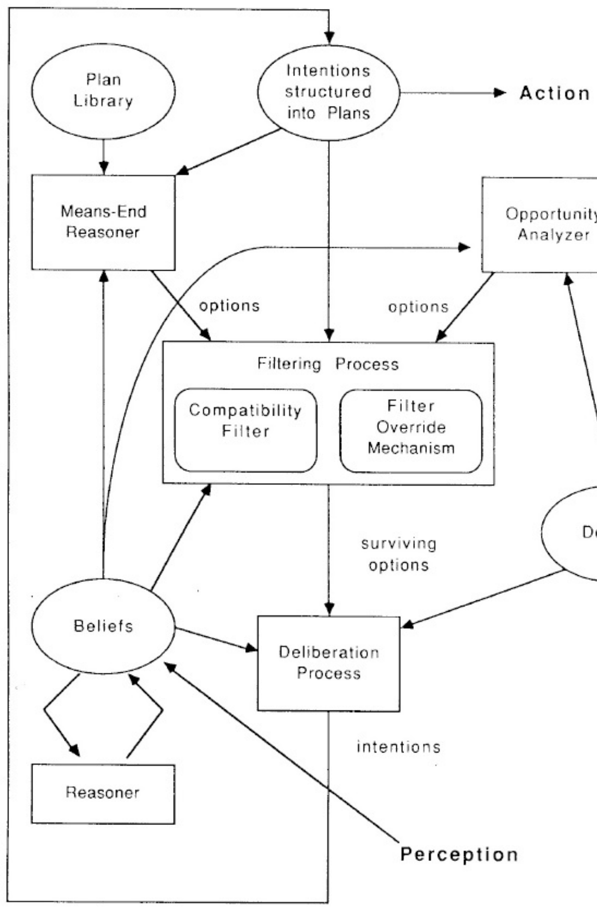


Figure 2.1: An architecture for resource-bounded agents [18]

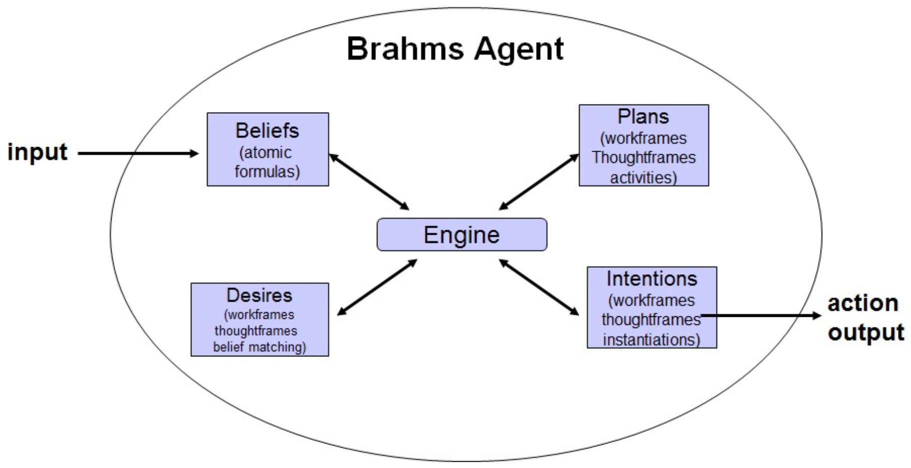

Brahms Agent
input
Beliefs (atomic formulas)
Plans (workframes Thoughtframes activities)
Engine
Desires (workframes thoughtframes belief matching)
Intentions (workframes thoughtframes instantiations)
action output

Figure 2.2: Relating Brahms to the BDI architecture

i.e., if the agents beliefs match the guard condition of a workframe then that agent has a desire to execute this workframe. Plans are formed using the workframes and thoughtframes order by their priorities. Workframes will be executed in the descending order of their priority forming a plan of workframes, each workframe will complete a sub-task of activities with the aim of achieving the overall goal. The intentions of the agent are then the currently active workframes and thoughtframes which can be selected for execution.

The emphasis on the BDI agent architecture in this chapter is due to the likeness that Brahms has to this architecture. We would also like to point out that there are other alternative architectures for implementing agents. An example of such architectures are deliberative, planning, *Intelligent Resourse-bounded Machine Architecture* (IRMA), and reactive [87]. A deliberate agent has an explicitly represented, symbolic model of its environment. Decisions, such as the actions it will perform, are then made through some logical reasoning. An example of a planning agent architecture is STRIPS which takes a symbolic description of the world, a desired goal state, and a set of action descriptions. Pre- and post-conditions are used to execute actions in a specific order to achieve the goal. The IRMA architecture uses a symbolic data structures with a plan library and explicit representations for the beliefs, desires and intentions of the agent. A reasoner is then used to reason about the world and determine which plans may achieve the agent's intentions. The IRMA architecture also has an opportunity analyser to monitor the environment to respond to events. A reactive agent architecture is one which does not use a central symbolic model of the world and does not use complex symbolic reasoning; the agents simply react to events that occur [87].

## 2.5 Chapter Summary

This chapter introduced the notion of an agent, what it is and how it can be used in the real world. It also introduced the idea of rational agents, agents that can act in a rational explainable way so that we can understand the actions of the agent. It also introduced the idea of multi-agent systems, where there are multiple agents (of varying types from software to robotic) who interact in an environment, sometimes competing against each other and other times assisting each other to achieve a shared goal. Issues of multi-agent systems were also discussed along with how they can help in real world scenarios. This chapter also brought forward the idea of teamwork, in respect to human-human and human-agent; detailing the difficulties and advantages of having humans and agents working together in a team. Finally the idea of the beliefs desires intentions paradigm was discussed, the paradigm which the Brahms simulation framework is based upon.

# Chapter 3

# Brahms

In this chapter we will discuss Brahms (Business Redesign Agent-based Holistic Modeling System), a multi-agent modelling, simulation and development environment devised by Sierhuis [73] and subsequently developed at NASA Ames Research Center. We will describe the core aspects of Brahms and what they are for, along with some example code to illustrate how they are used. We also describe applications of Brahms, showing how and when it has been used. A comparison between Brahms and other agent programming languages is presented along with an explanation of why Brahms was chosen over these other languages.

## 3.1 Basic Anatomy of Brahms

### 3.1.1 Geography

In Brahms the model of the agent's world is described using the geography model. Here the world is organised hierarchically, where an area can be conceptual (an *areaDef*, e.g., house, restaurant) or a physical location (*area*, e.g., 10 Downing Street). These *area* and *areaDef* are used to form the hierarchy where: an *area* can be an *instanceof* an *areaDef*; an *areaDef* can *extend* another *areaDef*; and an *area* can be *partof* another *area*. The distance between two areas is described using a *path*, undefined paths are transitively calculated from the defined paths. Agents are assigned an initial location in the geography in their code. The following code shows a Brahms description of a city called Berkeley which has a university, a restaurant, a bank and a hall within the university. It also describes a path from the hall to the restaurant and the bank which infers a path from the bank to the restaurant.

```
area AtmGeography instanceof World { }
areadef University extends BaseAreaDef { }
areadef UniversityHall extends Building { }
areadef BankBranch extends Building { }
areadef Restaurant extends Building { }
area Berkeley instanceof City partof AtmGeography { }
area UCB instanceof University partof Berkeley { }
area SouthHall instanceof UniversityHall partof UCB { }
area Telegraph_Av_2405 instanceof Restaurant partof Berkeley { }
area Bancroft_Av_77 instanceof BankBranch partof Berkeley { }

path StH_to_from_RB {
     area1: SouthHall;
```

```
    area2: Telegraph_Av_2405;
    distance: 400;
}
path StH_to_from_WF {
    area1: SouthHall;
    area2: Bancroft_Av_77;
    distance: 200;
}
```

### 3.1.2 Agents and Objects

Agents and objects are the core components of every Brahms simulation with agents modelling intelligent entities and objects modelling inanimate objects and sensors, etc. Objects have the same capabilities of agents except they react to external factors (facts, explained in section 3.1.3) and agents react based on their internal beliefs. Brahms provides an option for objects to switch from reacting to external factors to internal beliefs, but for our purposes we consider objects to react to external factors.

### 3.1.3 Attributes, Relations, Beliefs and Facts

Agents and objects can have their own personal attributes, relations and beliefs. Attributes are characteristics of the agent such as their height, weight, power levels, etc. Attributes can be of type Boolean, String, Double or Integer. Relations are a form of attribute where the attribute's 'type' is set of agents or objects (these sets are groups and classes, explained in Section 3.1.8). The relation then allows for a connection between agents and objects, e.g., *public Account hasAccount* would state that the current agent could have a relationship called 'hasAccount' with a set of agents (or objects) labelled 'Account'. Beliefs and facts are all tied to attributes and relations, every belief and fact has to contain either an attribute or a relation, e.g., an agent could have the belief *AgentA hasAccount AccountA* which would represent an 'AgentA' owning an account called 'AccountA'. Beliefs and facts differ in that facts represent the real value the attribute or relation has and the beliefs represent what the agent believes it to be. Below is example code for declaring an agent with attributes, etc. note that the initial location on initialisation is defined using 'location:' and beliefs use the keyword 'current' to refer to the agent being defined by this code, e.g., 'current.howHungry = 0' represents the current agent's belief that its attribute 'howHungry' has value 0 where 'Bob.howHungry = 0' represents this agent's belief on Bob's attribute 'howHungry'.

```
agent Alex {
/*Assign the agent's initial location*/
   location: SouthHall;
   attributes:
      public double howHungry;
      public int perceivedtime;
   relations:
      public Account hasAccount;
   initial_beliefs:
      (current.howHungry = 0);
      (Bob.howHungry = 0);
      (current.perceivedtime = 0);
      (current hasAccount Alex_Account);
   initial_facts:
      (current.howHungry = 0);
      (current.perceivedtime = 0);
      (current hasAccount Alex_Account);
```

### 3.1.4 Workframes and Thoughtframes

Workframes and thoughtframes represent the work and thought process in Brahms. A workframe contains a sequence of activities and belief updates which the agent/object will perform, whereas a thoughtframe only contains sequences of belief updates; a thoughtframe is simply a restricted workframe which is unable to process activities. Workframes can detect (using *detectables*) changes in the environment, update agent's beliefs accordingly and then decide whether or not to continue executing. Essentially, workframes represent the work processes involved in completing a task and thoughtframes represent the reasoning process upon the current beliefs, e.g.,

- Start workframe to go to the shops
- Rain causes a detectable to fire
- Agent now believes it is raining
- Workframe is suspended
- Thoughtframe is executed to decide that the agent needs a coat

### 3.1.5 Executing plans: Activities and Concludes

Agents are able to perform activities and concludes (belief/fact updates), these are executed via workframes and thoughtframes (which can update beliefs only, no fact updates or activities can be performed) which decide when they should be performed. *Primitive* activities, *move* activities and *communication* activities are the three main types of activities.

**Primitive activities** are conceptual activities in the sense that they only spend simulation time, while the assigned name infers what the agent was doing, e.g., *primitive* activity 'dig_hole' has a duration of 400 time units and no belief or fact updates are made. When assessing the simulation the name infers that the agent was digging a hole. To confirm a hole was dug the workframe would require a conclude to update the beliefs and the facts that a hole now exists. Brahms is a simulation framework, not an execution framework, so activities such as *primitive* activities are used to subtract simulation time while an event is occurring.

**Move activities** are performed to change an agent's location. Like *primitive* activities they are assigned a duration, however this duration is calculated from the Brahms geography model. When a *move* activity is performed multiple things occur: the simulation time is spent; the agent's location is changed; all other agent's in the previous location have their beliefs deleted about the agent's location and the agents in the new location recognise this agent has joined them.

**Communication** Activities are used for passing messages between agents, these communications are assigned a duration. Once this duration is over the beliefs of the other agents are updated corresponding to this communication, an agent however can only communicate beliefs it already has.

Below is an example of a workframe for eating using a simple *primitive* activity and a belief update. Note that a repeat variable has been set to true so that the workframe can be performed multiple times. The 'when' condition states the requirement for the workframe to activate, i.e., when the agent believes its hunger is greater than 10. The 'do' statement declares which activities and belief updates will be made; 'eat()' represents the activity to eat and conclude defines a belief update, i.e., that the agent's hunger decreases by 3. Note that the belief update has 'bc:100' and 'fc:0', which represents a 'belief certainty' of 100% and 'fact certainty' of 0%, i.e., update the belief with a 100% probability and the fact with a 0% probability.

```
workframe wf_eat {
   repeat: true;
   when(knownval(current.howHungry > 10))
   do{
      eat();
      conclude((current.howHungry = current.howHungry - 3.00)
         , bc:100, fc:0);
   }
}
```

### 3.1.6 Detectables

Detectables are contained within workframes and can only be executed if their workframe is currently active. They can detect changes in facts and can: abort, continue, complete or impasse the workframe. When a detectable is executed it imports the fact it "detected" into the agent's belief base and then it either:

1. *abort* - deletes all elements from the workframe's stack;
2. *continue* - carries on regardless;
3. *complete* - deletes only activities from the workframe's stack; or
4. *impasse* - suspends the workframe until the detectable's guard is no longer satisfied

Below is an example of a workframe containing a detectable. The detectable states to 'impasse' (suspend) the workframe when it detects the agent's thirst is greater than 10. The 'when(whenever)' means that the detectable can be activated at any time, a point of time in the simulation could be inserted here instead. The 'dc:100' represents a 'detect certainty' of 100%, meaning the detectable will always fire when it is active.

```
workframe wf_eat {
   repeat: true;
   detectables:
      detectable thirsty{
         when(whenever)
         detect((current.howThirsty > 10), dc:100)
         then impasse;
      }
   when(knownval(current.howHungry > 10))
   do{
      eat();
      conclude((current.howHungry = current.howHungry - 3.00)
         , bc:100, fc:0);
   }
}
```

### 3.1.7 Variables

Variables provide a method of quantification within Brahms. So, if there are multiple objects or agents which can match the specifications in a guard condition then the variable can either perform: *forone* - select one; *foreach* - work on all, one after another; or *collectall* - work on all at once. Below is an example of a workframe containing a variable to identify an object in the class of objects called ‘Cash’. The guard condition identifies which member of ‘Cash’ is to be selected, i.e., an object that the current agent has the relationship ‘hasCash’ with. The selected object is identified by the assigned name ‘cs’. A belief update in the workframe then changes the agent’s belief about the object’s attribute ‘amount’.

```
workframe wf_eat {
   repeat: true;
   variables:
      forone(Cash) cs;
   when(knownval(current.howHungry > 10) and
      knownval(current hasCash cs))
   do{
      eat();
      conclude((current.howHungry = current.howHungry - 3.00)
         , bc:100, fc:0);
      conclude((cs.amount = cs.amount - 10.00)
         , bc:100, fc:0);
   }
}
```

### 3.1.8 Groups and Classes

Groups in Brahms form the hierarchical structure of agents, where agents can be members of groups and groups can also be members of other groups. Groups form a template for an agent, so if an agent is a member of a group then it will inherit all the beliefs, workframes and thoughtframes declared in the group. The following code shows a Brahms description of a group called student, any agent who is a member of student will inherit all the attributes, relations, beliefs, etc. Classes are identical to groups except they form a hierarchical structure of objects, not agents. Below is an example of a group with an agent inheriting from the group. To make the code more clear the attributes, etc. have been replaced by a comment, but notice that the agents’ codes

```
workframe ::=   workframe workframe-name
                {
                { display : ID.literal-string ; }
                { type : factframe | dataframe ; }
                { repeat : ID.truth-value ; }
                { priority : ID.unsigned ; }
                { variable-decl }
                { detectable-decl }
                { [ precondition-decl workframe-body-decl ] |
                   workframe-body-decl }
                }
```

Figure 3.1: BNF Grammer Representing Part of the Brahms Syntax Specification

are empty except for a location. This is to show that the agents can be entirely described in their superstructures but any individual information can be placed in their own personal sections of code.

```
group student{
   attributes:
   /*all attributes here*/
   initial_beliefs:
   /*all beliefs here*/
   initial_facts:
   /*all facts here*/
   workframes:
   /*all workframes here*/
   thoughtframes:
   /*all thoughtframes here*/
}

agent Alex memberof student{
   location: SouthHall;
}
agent Bob memberof student{
   location: Telegraph_Av_2405;
}
```

### 3.1.9 Brahms Syntax

Brahms has an expressive sophisticated syntax for creating systems, agents and objects, the full syntax specification can be found in the appendix or the 'agentisolutions' website[1]. As an example Fig. 3.1 shows the definition of an agent's workframe showing where variables, detectables and the main body of the workframe are placed. Guards are specified by `precondition-decl`.

[1]For the full Brahms syntax (with an informal semantics) see [42].

## 3.2 Applications

Brahms has been used in a variety of projects within NASA, from modelling the NYNEX telephone exchange in the early years to modelling the Mars Rover exploring the Victoria crater. Figure 3.2 provides an insight into where Brahms has been applied from 1992 up until 2008.

### 3.2.1 Extra-Vehicular Activities during Mars exploration

Bordini, Fisher and Sierhuis [9] describe a possible scenario of human-robot teamwork during a Mars exploration mission. An overview of the scenario can be found in Figure 3.3 and a more detailed description is found below.

> During an Extra-Vehicular Activity (EVA) there are two surface astronauts and two EVA Robotic Assistants (ERA) assigned to explore a region of Mars. Both the astronauts and the ERAs have their own agenda to work to, but the ERAs also have the responsibility of ensuring that both the astronauts always have a network connection back to the habitat. The ERAs, like the humans, can be interrupted while performing their assigned tasks. When this happens the ERAs need to be able to handle this interruption without jeopardizing the astronauts, themselves or the mission.
>
> ERAs can be assigned to "Team up" with an astronaut, becoming the astronauts personal agent (PA). The ERA's tasks then involves additional functions such as "follow astronaut" and "astronaut watching". The ERA also acts as a relay point for the connection back to the habitat for the astronaut. The ERA will therefore detect and inform an astronaut when they are moving out of range of communications with the habitat. Also if an astronaut calls to its PA for assistance the PA will have to suspend any activities it is performing and come to the astronaut's aid. However if the PA is engaged in an activity which is more important than another ERA's or another ERA is closer to the astronaut then the ERA can ask the other ERA to temporarily take over the role as the astronaut's PA.
>
> In summary, the ERAs assigned to work with the astronauts have to be completely autonomous robots which can fulfil assigned tasks without human assistance. They need to be able to react to unpredictable situations such as loss in communications, as well as aiding other astronauts and robots when requested.

### 3.2.2 OCAMS

Orbital Communications Adapter (OCA) officer flight controllers in NASA's International Space Station Mission Control Center use different computer systems to up link, down link, mirror, archive, and deliver files to and from the International Space Station (ISS) in real time. The OCA Mirroring System (OCAMS) is a multi-agent software system operational in NASA's Mission Control Center [77], replacing the OCA officer flight controller with an agent system that is based on the behaviour of the human operator. NASA researchers developed a detailed human-behavioural agent model of the OCA officers' work practice behaviour in Brahms. The agent model was based on work practice observations of the OCA officers and the observed decision-making

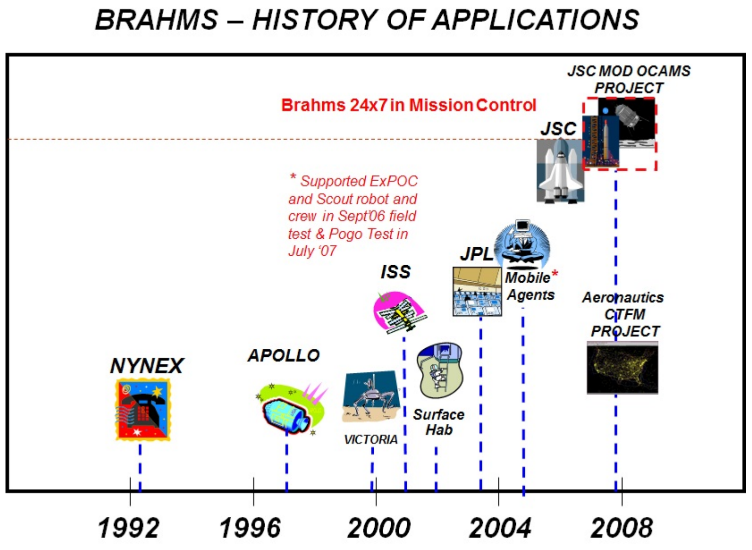


Figure 3.2: A history of Brahms applications

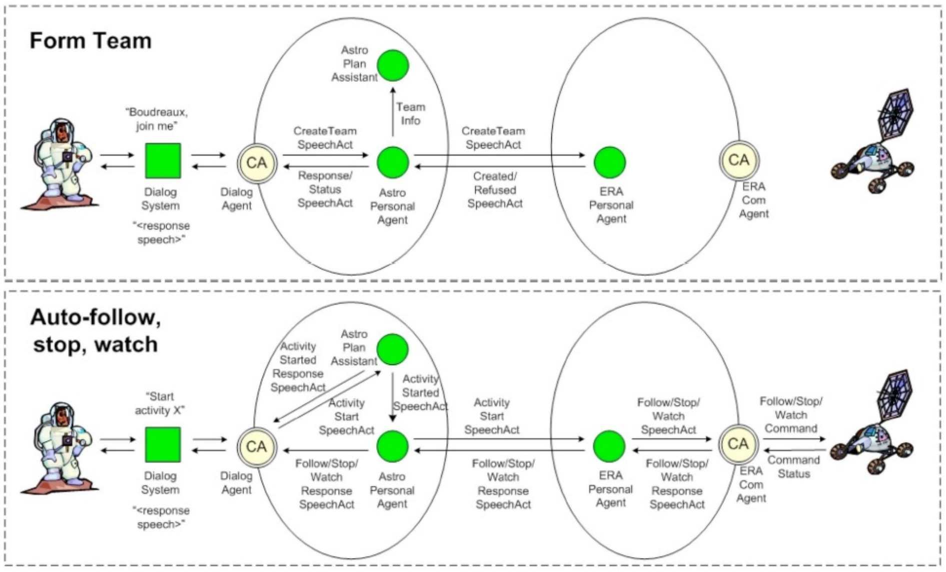


Figure 3.3: EVA Activity on Mars [9]

involved with the current way of doing the work. In the system design and implementation phases, this model of the human work practice behaviour was made part of the OCAMS multi-agent system, enabling the system to behave and make decisions as if it were an OCA officer. Here is a short scenario of how the OCAMS system is used in mission control:

The On-board Data File and Procedures Officer (ODF) sends a request to the OCAMS (personal) agent via their email system. The OCAMS agent parses the request and understands that the ODF has dropped a zip file to be up linked to the ISS on the common server. The OCAMS agent needs to identify the type of file that is being delivered and decide, based on this, what up link procedure needs to be executed. Having done so, the OCAMS agent chooses the procedure and starts executing it, as if it were an OCA officer. The OCAMS agent first transfers the file and performs a virus scan, and then continues to up-link the file to the correct folder on-board the ISS. The OCAMS agent applies the same procedure that an OCA officer would do.

The OCAMS system has been extended over three years [22]. With the latest release the OCAMS system will have completely taken over all routine tasks from the OCA officer, about 80% of the workload. Other flight controllers in mission control will interact with the OCAMS agent as if it were an OCA officer.

#### 3.2.3 Teamwork In Brahms

Intentions in Brahms are represented by the guard conditions on workframes (plans of actions) and actions are represented by Brahms activities which spend simulation time, and in the case of move and communication activities they change locations and other agent's beliefs as well. These intentions and actions can become joint through inheritance from a super class where the agents inherit team's activities and workframes. This isn't ideal as different members of the groups will have different tasks to do within the team and different sub-goals. To model *joint intentions* the modeller has to specifically give agents individual workframes with guard conditions which are linked. Modelling *joint activities* is also difficult; agent's workframes need to be synchronised using guard conditions and Brahms detectables (to detect when other team members are ready to start the *joint activity*) to ensure the agent's operate together. Additional workframes then need to be used to conclude whether *joint activities* have been successful. The abstract nature of the Brahms activities does however alleviate some of the difficulties of modelling a *joint activity* for example: a task for three agents picking an object up together only has the difficulty of ensuring that the agents start and finish their activities at the same time, the agents then conclude the object to be "picked up" after the simulation time elapses.

## 3.3 A Comparison with other Agent Programming Languages

Over the years a plethora of different agent theories, languages and architectures have been proposed and developed. In this section we give the reader an insight into how Brahms and three of the main agent programming languages differ to help explain why we have chosen the Brahms framework. Many of these agent-orientated programming languages are based on Prolog, which uses a logical goal reduction approach, i.e., identifying its goals and reducing them into sub-goals [10].

### 3.3.1 3APL

3APL agents use practical reasoning rules, which have been extended from the recursive rules of imperative programming, to monitor and revise agents' goals [39]. 3APL incorporates features from both imperative and logic programming plus features which allow for a descriptions of agent oriented features, e.g., the querying of agents' beliefs. 3APL also supports agents which have reflective capabilities related to their goals or plans provided by practical reasoning rules. Agents in 3APL follow these characteristics of intelligent agents:

- sophisticated internal mental state made up of beliefs, desires, plans, and intentions, which may change over time
- agents act pro-actively, i.e., goal-directed and respond to changes in a timely fashion
- agents have reflective or meta-level reasoning capabilities

A 3APL program allows a user to define the agent's **capabilities**, **beliefbase**, **rulebase** and **goalbase**. The **capabilities** of an agent are the actions which it can perform such as put block a on block b. This is done using pre and post conditions with an action statement, e.g., a precondition $on(a, table)$ would mean for this action to be performed then block a is on top of the table, with an action statement $aOnb()$ (used to call the action) and post condition $on(a, b)$ which states that after this action is performed block a will be on top of block b. The **beliefbase** is used to state what the agent believes, e.g., $on(a, table)$ means the agent believes block a is on the table. The **rulebase** is used to describe the tasks the agent will complete, e.g., $putAonB()$ $< - aOnb()$. tells the agent to perform the action $aOnb()$. Rules can contain multiple actions which the agent can perform in sequence. Finally in the **goalbase** are the list of rules which are to be performed such as $putAonB()$. Any additional requirements such as the environment or graphical interface, etc. are programmed using C and C++. An example of a 3APL program for the Blocks World problem is as follows:

```
CAPABILITIES
    {on(a,table), on(b,table)} aOnb() {on(a,b)}

BELIEFBASE
    on(a,table).
    on(b,table).

GOALBASE
    putAonB().

RULEBASE
    putAonB() <- aOnb().
```

For more information on 3APL see Hindriks et al. [39]

### 3.3.2 GOAL

GOAL is based on the concept of rational agents, see Chapter 2.2. GOAL is a BDI based language where agents derive their plans from their beliefs and goals. GOAL facilitates the manipulation of an agent's beliefs and goals in order to structure its decision-making. The language is based on common sense notions and basic practical reasoning. The main features of GOAL are [82]:

- Declarative beliefs; the agent's initial beliefs at the start of the simulation
- Declarative goals; the agent's initial goals
- Blind commitment strategy; drop goals only when they have been achieved
- Rule-based action selection; selection of actions based upon rules
- Policy-based intention modules; specific focusing on achieving a subset of the agent's goals using only knowledge relevant to achieving those goals
- Communication at the knowledge level; inter-agent communication to exchange information, and coordinate actions

When programming in GOAL you are allowed to describe the multi-agent system using **environment**, **agentfiles** and **launchpolicy** tags. The **environment** tag uploads an environment from a Java file and the **agentfiles** allows uploading of GOAL agent files to the system. The **launchpolicy** is used to give rules on when the agent starts. For example:

```
environment{
    "environment.jar" .
}

agentfiles{
    "agentA.goal" .
    "agentB.goal" .
}

launchpolicy{
    when entity@env do launch alex : agentA, bob : agentB .
}
```

When defining an agent GOAL allows descriptions of the agent's **knowledge**, **beliefs** and **goals**. In the Blocks World example the **knowledge** section can be used to define the expressions; such as $block(a), block(b)$ to say we have a block a and b, and $on(X, Y)$ to represent the description of any block X can be on any block Y. The **beliefs** section represents what the agent believes such as $on(a, b)$ to state the agent believes block a is on block b. The **goals** section would then contain beliefs the agent wishes to achieve such as $on(b, a)$. A **program** section then details the actions which can be performed and when such as $if\ goal(on(a, b))\ then\ move(X, Y)$ where action $move(x, y)$ is defined in the **action-spec**. Note that $if\ a - goal(on([X|T]))$ is used to describe when a block has been misplaced. Any additional information such as the environment or the graphical interface are programmed using Java. Example code for the Blocks World problem is as follows:

```
main BlocksWorldAgent
{
    knowledge{
        block(a), block(b).
        on(X,Y).
    }

    beliefs{
        on(a,table), on(b,table).
    }

    goals{
        on(a,b).
    }

    program{
        if a-goal(on(X,Y)) then move(X,Y).
        if a-goal(on([X|T])) then move(X,table).
    }

    action-spec{
        move(X,Y) {
        pre{ clear(X), clear(Y), on(X,Z) }
        post{ not(on(X,Z)), on(X,Y) }
    }

}
```

For more information on GOAL see [82]

### 3.3.3 AgentSpeak(L)/Jason

AgentSpeak(L) is an extension of logic programming for the BDI agent architecture, providing a framework for programming BDI agents [67]. An AgentSpeak(L) agent defines its beliefs as a set of ground (first-order) atomic formulae and uses a set of plans to form its plan library. AgentSpeak(L) contains two types of goals: achievement goals; atomic formulas prefixed with the '!' operator and test goals; prefixed with the '?' operator. Achievement goals describe a world state that the agent wants to achieve. Test goals are a test on whether the associated atomic formulae form one of the agent's beliefs.

AgentSpeak(L) forms a reactive planning system where agents react to events related to either changes in beliefs due to perception of the environment, or to changes in the agent's goals due to the execution of plans. Plans are predefined and triggered by changes in beliefs and goals by either by addition '+' or deletion '-'. An AgentSpeak(L) plan has a body, which is a sequence of basic actions (sub-goals) that the agent has to achieve (or test). Basic actions can be atomic operations the agent can perform to change the environment, or actions written as atomic formulae using a set of action symbols rather than predicate symbols.

Jason is an extension of AgentSpeak(L) with additional functionalities such as atomic formulae which are able to have annotations on the sources of the agent's beliefs

[13]. Continuing with the Blocks World example, Jason allows simple declaration of beliefs with $onTop(a, table)$ to represent the agent believing block a is on top of the table. The plans are denoted using the @ symbol, e.g., $@P1$ would describe a plan called P1. In this plan a goal needs to be added with a condition, e.g., goal addition by $+![on(a, b)]$ and condition : *not* $onTop(a, b)$ stating block a must not be on top of block b. When these conditions are met a belief removal and addition are made; $-onTop(a, table)$ to say that block a is no longer on the table and $+onTop(a, b)$ to state that block a is now top of block b. An example of an AgentSpeak(L) Blocks World program is as follows:

```
/*Initial Beliefs*/
onTop(a,table).
onTop(b,table).

/*Plans*/
@P1
+![on(a,b)]
    : not onTop(a,b)
    <-  !putAonB(a,b);
    -onTop(a,table);
    +onTop(a,b);
```

3APL has been applied in main stream areas such as mobile computing [50], where an architecture 3APL-M has been developed to support the development of deliberative multi-agent systems in mobile computing devices. 3APL has also been used to for programming cognitive robots. For more information on AgentSpeak(L) and Jason see Bordini et al. [13]

### 3.3.4 Why Brahms

To decide which agent language/framework to use we drew up a list of requirements to help identify which language would best suit our needs. The list of requirements we generated are as follows:

1. allow for high level descriptions of agent activities with minimal coding
2. can show a distinct time line of events to demonstrate when activities occur
3. an embedded geographic model which requires no external code or environments
4. allow agents to perform human specific behaviour, such as:
    (a) thinking about problems
    (b) reasoning about the implications of beliefs and causality of actions
    (c) off task behaviour and multi-tasking
    (d) making mistakes
    (e) communicating with each other
    (f) taking varied amounts of time to complete tasks
5. and have an already existing user base for simulating human behaviour

When considering these requirements it became clear that Brahms was the only language we could find to match these requirements. Brahms has the high level abstractions of activities all contained within a time line of events. It has a simple geographic syntax for describing the possible locations, the distance between them, their associations with each other and it is very simple to tell agents to move from location to location. Brahms was also developed for simulating humans, so it meets our human behaviour requirements. Brahms has also been used at NASA to simulate human behaviour for over 10 years.

Other agent languages considered were AgentSpeak(L) and Jason, 3APL, and Goal. These are all common and popular languages used to program agents, which is their primary function. Brahms on the other hand was developed to model both humans and machines at NASA [76, 73, 21]. Although representation of humans may be possible in all the above languages, it is only Brahms which has been specifically designed and used to model human behaviour. Brahms achieves this representation of humans by modelling the objects they use, the environment they are in, their thought and work processes, communication, and other concepts where humans and agents will typically collaborate. AgentSpeak(L) and Jason, 3APL, and Goal are focussed on the development and behaviour of the agents, including competitiveness and lack a framework for shared achievement [11]. However, it could be argued that any of these languages could be used for modelling humans, even though they have not been specifically designed to do so. This then leaves the issue of defending how accurate this representation of a human is, whereas with Brahms there are already papers and experiments demonstrating its use in modelling humans [76, 73, 21] and thereby we can safely assume it meets both requirements 4 and 5. Brahms also allows for a high level of abstraction, where capabilities of agents can be assumed. This high level of abstraction, which matches requirement 1, allows for easier modelling of humans because we can assume they can perform certain tasks, such as moving a simple object, without any difficulties. Whereas an agent centred approach like 3APL, GOAL and AgentSpeak(L) do not usually allow for such high level of abstraction, they require us to model every action. Activities in Brahms are inherently linked with time, meaning no extra coding is required which works towards meeting requirement 2. The other agent languages are more interested in the events themselves than the time they take to complete, although it is possible for them to represent this time it will require additional coding making the simulation more verbose. Brahms is also the only language which matches requirement 3; an embedded graphical model. AgentSpeak(L), 3APL and GOAL all require the geographic models to be described in another language such as Java or C, making the simulations much more verbose and add an additional language to the verification process. Brahms' graphical model is simple to program, has very little extra syntax and allows easy reference to these locations in the simulations.

In summary, we are interested in analysing the teamwork aspect of human-agent teams, examining the interactions and work processes the teams use to complete their task. Therefore it only seems natural that we choose a framework designed to model humans and work processes, such as Brahms. Brahms is also the only language we could find that meets all of our requirements easily, making it an obvious choice. Additionally Brahms has already been used to model human-agent-robot teamwork, meaning there are existing models/examples already available for us to apply verification.

# Chapter 4

# Formal Verification - Techniques and Applications

The primary concern of this thesis is to perform formal verification of models of human-agent teamwork. In this chapter we discuss: what we mean by the term formal verification; why we wish to perform agent verification; and which formal verification technique we use and why. The structure of this chapter is as follows:

- Formal Verification; here we explain what we mean by formal verification and formal operational semantics
- Model Checking; here we explain the formal verification technique model checking and the model checking tool Spin
- Other Formal Verification Techniques; here we briefly explain opposing verification techniques to model checking
- Agent Verification; here we explain how agent verification has previously performed on agent based systems
- Verification of Agent Languages; here we explain how verification has been performed on agent languages

## 4.1 Formal Verification

Formal verification represents a family of techniques aimed at assessing the correctness of a system design. These techniques have become very popular in hardware design since they can ensure 100% functional correctness of circuit designs [30]. For example, Kaivola and Narasimhan [46] describe the process they used to verify the floating-point multiplier in the Intel IA-32 Pentium microprocessor. Formal verification is always performed against a set of requirements, i.e., a specification. A formal specification is a concise mathematical description of the behaviour and properties of a system, stating what actions a system can take and what should (or should not) happen [51]. Informal specifications are inadequate for formal verification as they tend to be vague, improper, incomplete, hard to analyse and ambiguous. A multitude of formal languages and logics have been created in order to express as broad a range of properties as possible. *liveness*, *safety* and *fairness* are typical properties to check. The *liveness* concept states the system must at some point perform this action, e.g., "the spacecraft will take off". *Safety* is a concept which states something must never happen, e.g., "while performing manoeuvres in space, no doors can be opened". *Fairness* is a concept used when multiple agents are employed; it states that each agent will fairly get the chance to perform an operation with no agent indefinitely occupying the resource. Formal languages may also need to be able to express properties concerning real-time dynamic systems, probabilistic systems and goal driven systems [11].

Essentially formal verification is a reachability test, testing whether a certain state can be achieved which does not satisfy the specification. Formal verification can however be used to identify other faults in a system which are not part of the specification such as deadlock, livelock, race conditions and termination. Deadlock occurs for instance when a process will not release a shared resource that other processes are waiting to access and cannot progress any further until they access the resource. Livelock is similar to deadlock, such that no progress is made but no blocking occurs [63], e.g., one processor constantly flips a Boolean to true and in response another flips it back to false. There is no strict definitive definition of a race condition, however race conditions generally occur when different processes share a data source without explicit synchronization [60]. Termination analysis is simply a check to identify whether the program will always terminate, such as identifying any infinite loops. Termination is a difficult problem, also known as the halting-problem, and has been the subject to intensive research, for example, in [53, 33, 26].

The most popular approach to formal verification is model checking [27]. Model checking (see Chapter 4.2) creates a model of every single state achievable within a system, the transitions between these states, and also indicates which states are the possible initial states. Every single run (a sequence of state transitions from an initial point to an end point) is checked to ascertain whether or not a formal property holds. Model checking requires a finite model of the system, which we generate from the operational semantics, and the representation of a property to check in some logic.

### 4.1.1 Formal Operational Semantics

Formal operational semantics are interpretations of formal (mathematical) languages. They are used to precisely describe how a system will behave when executed. This is done by describing semantic rules which identify the possible types of states the system can be in and the ways the state can change when applying these rules. These semantic rules can be used to build a model of the system by applying the rules to the initial states of the system and then storing the resulting states in the model.

Formal operational semantics consist of a set of rules which govern how/when and what the system will do at a given time. These rules form a premise and a conclusion, rules with empty premises are axioms. Set theory and logical notations are used to describe both the premise and the conclusion. These notations are able to express the state of the system (e.g., a set of tuples and their values), describe the changes made to the data structures, and the resulting system after the changes have been made [84].

## 4.2 Model Checking

Model checking is the technique which has been used in this thesis to analyse the behaviour of human and agent teamwork. Model checking is a verification technique developed to logically analyse whether a system meets certain specifications. Specifications are described in a precise mathematical language and the system itself is typically represented by a finite state machine (FSM), i.e., a directed graph consisting of vertices and edges. The model is exhaustively searched for reachable states where the specification fails [2]. Specifications checked are unique to the systems requirements, typically of a qualitative (e.g., is the result OK?) and a timed nature (e.g., is task achieved within $x$ hours). General bad states such as deadlock, livelock and race conditions are also checked for. Figure 4.1 shows the model checking process, showing that the process starts with a description of the model and a desired property of the system. A model is then created from this description and a quick test is performed to check its accuracy. Meanwhile, the specification is formalised into a property specification language. The model checker then exhaustively checks every state to determine whether or not the property holds. If the property holds then the verification is complete, if not then an error trace through the model is produced to demonstrate how the property can be violated. Additionally, the model generated can be too large to be held in memory, requiring the model to be refined before verification can continue.

Baier and Katoen [2] detail a list of strengths and weaknesses of model checking:

1. Strengths
   - (a) Wide range of applications: embedded systems, software engineering and hardware design
   - (b) Supports partial verification, i.e., each property can be checked individually
   - (c) Not vulnerable to the likelihood that an error is exposed
   - (d) Provides diagnostic information for debugging purposes
   - (e) System does everything for the user making it simple to use
   - (f) Increasing interest within industry
   - (g) Easily integrated into existing development cycles
   - (h) Verification performed is trustworthy due to its mathematical soundness
2. Weaknesses
   - (a) Unsuitable for data applications as they tend to cover infinite domains
   - (b) *Decidability* can cause issues which can create infinite-state systems
   - (c) Finds design flaws not coding errors or fabrication faults
   - (d) Checks only requirements which have been specified
   - (e) *State-explosion problem* may cause the system to run out of memory
   - (f) Creating the model of the system requires *expertise* to abstract a small enough system model and to convert the specifications into logical formulae
   - (g) Results aren't guaranteed to be correct, the model checker may contain its own software defects

The main reasons why we chose model checking are that it's an automated process and the mathematical soundness of the proof. The main weakness that did become an issue during the course of this thesis was the state-explosion problem, weakness 2(e). Techniques such as declaring sections of code as deterministic and scaling down the non-determinism were used to reduce the state space to resolve this issue. Weakness 2(a) was not an issue since the scenarios were modelled over a finite amount of time with a finite number of choices. Decidability (weakness 2(b)) refers to the model checkers

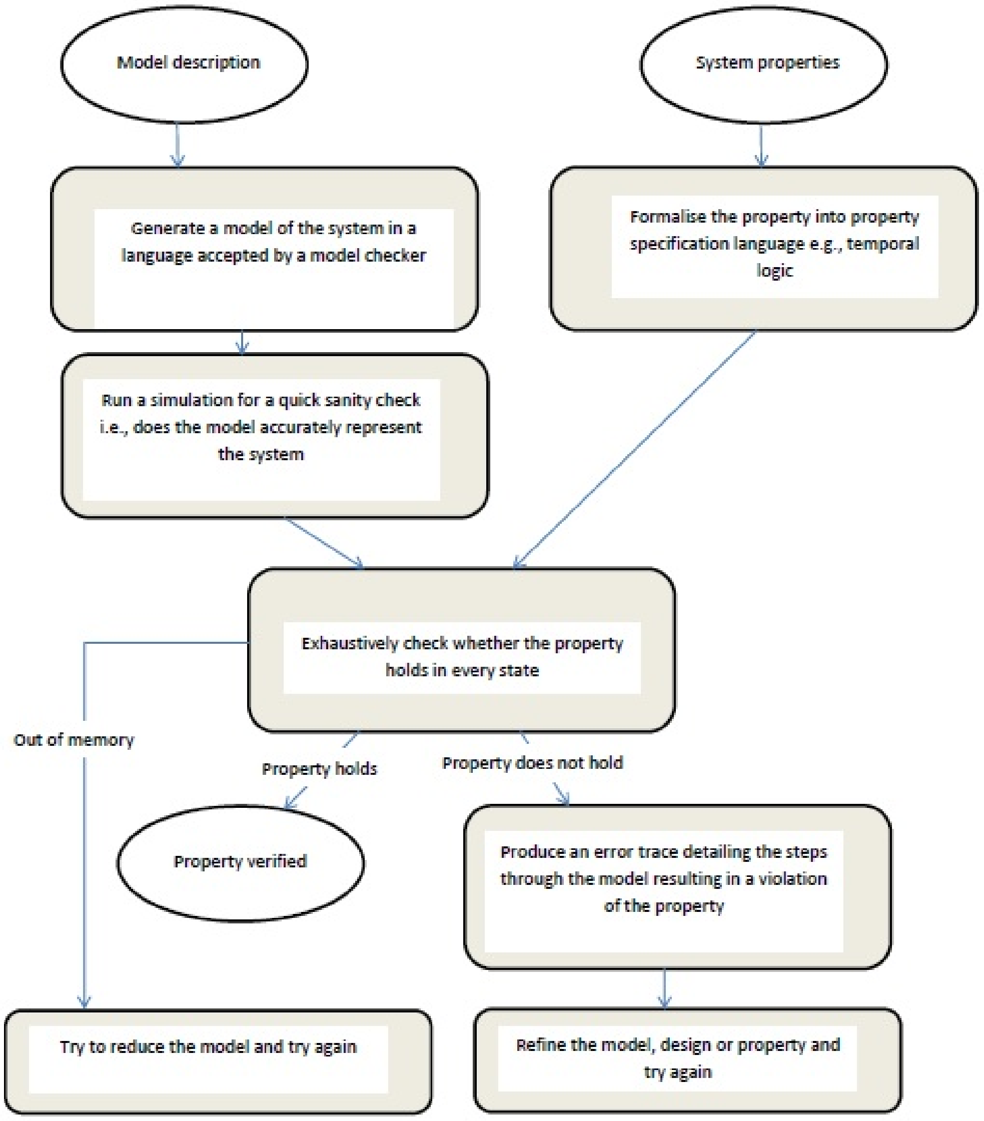

Model description
System properties
Generate a model of the system in a language accepted by a model checker
Formalise the property into property specification language e.g., temporal logic
Run a simulation for a quick sanity check i.e., does the model accurately represent the system
Exhaustively check whether the property holds in every state
Out of memory
Property holds
Property does not hold
Property verified
Produce an error trace detailing the steps through the model resulting in a violation of the property
Try to reduce the model and try again
Refine the model, design or property and try again

Figure 4.1: The model checking process [2]

ability to return a Boolean true or false to a specification of the system. Decidability issues arise when models are either too large to hold in memory or the model loops infinitely. The model checker Spin has techniques to identify and remove infinite loops, and tools to keep models within memory limits. However there is always a possibility that all available memory will become consumed leaving the model checker unable to return a result. We tried to avoid these memory issues by progressivly increasing non-determinism, since non-determinism is a key factor in state space explosions. By gradually increasing the non-determinism we can see at what point memory becomes an issue. 2(c) was not an issue since we are specifically looking for design flaws, coding issues can be resolved using the Brahms simulator. Model checking will only check for properties that it has specifically been asked to check (weakness 2(d)). This weakness was not a concern for us because we were only interested if our tool was able to correctly answer whether our specifications held or not. The model is generated automatically by the system we have created, eliminating the requirement of the user requiring the expertise to create the model, eliminating the issue 2(f). However, 2(f) also mentions that the user needs this expertise to convert the specifications into logical formulae. This means 2(f) is still an issue, the user needs to know how an agent's belief is represented in the *PROMELA* translation and they must know how to convert the specification in to a *PROMELA* 'Never_Claim'. 2(g) may be an issue, it is extremely difficult to tell if the *PROMELA* translation exactly matches the Brahms framework. This means that an error found in a the verification may not be an error in Brahms or vice-versa. Also Brahms is a simulation itself, meaning a Brahms simulation may not accurately represent the situation it was trying to model.

### 4.2.1 Spin

The model checker used for the verification of the human-agent teamwork in this thesis was the Spin model checker [41]. In this section we describe Spin and the syntax for its input language *PROMELA*. Spin is a popular model checking tool designed to verify models of distributed software systems. Spin models concentrate on proving correctness of *process interactions* and also attempt to abstract as much as possible from the internal sequential computations. Spin has a graphical front-end known as XSpin which allows users to define specifications of a high level model of concurrent systems. Spin accepts design specifications written in the verification language *PROMELA* (a Process Meta Language). The *PROMELA* language is very similar to ordinary programming languages, but it has the added capability of handling certain non-deterministic constructs [37]. This high level specification of the systems is tested via interactive simulations to assure a basic level of confidence in the model. The high level model is then optimised using reduction algorithms. The optimised model is used to verify the system. If any counterexamples to the specification are found during verification then they are passed through the high level model to inspect this example in greater detail. Figure 4.2 shows Holzmann's [41] diagram to illustrate the structure of Spin. The diagram shows that *PROMELA*, along with a property in Linear Temporal Logic, can be parsed in order to find: 1) syntax errors 2) produce a simulation or 3) generate a finite state machine for verification. For model checking the state machine is optimised and on-the-fly model checking is performed. On-the-fly model checking refers to checking the specification while generating the model. Any counter-examples produced are related to simulation runs.

Spin formulates models of concurrent systems by interleaving all process automata

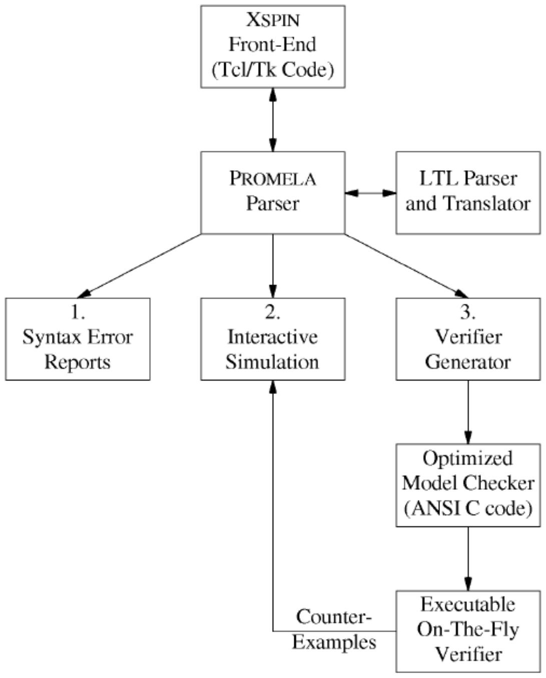

XSPIN Front-End (Tcl/Tk Code)
PROMELA Parser
LTL Parser and Translator
1. Syntax Error Reports
2. Interactive Simulation
3. Verifier Generator
Optimized Model Checker (ANSI C code)
Executable On-The-Fly Verifier
Counter-Examples

Figure 4.2: Structure of the Spin model checker [41]

to form a single automaton, a product forming the global state space. The specifications are represented using temporal logic, which Spin converts into Büchi automata; a finite state machine which accepts words of an infinite length. On execution Spin explores all possible states while synchronously running the Büchi automaton representing the specification. If the language accepted by the product of the Büchi automaton and the global state space is empty then the specification was not satisfied.

**PROMELA**

PROMELA is the input language for the Spin model checker, it was designed to make good abstractions of a system's design easier. *PROMELA* was designed to handle asynchronous processes, buffered and unbuffered message channelling, synchronizing statements, and structured data. Restrictions on *PROMELA* have been placed to make it easier to model and verify client and server behaviour but make it difficult to model complex mathematical behaviour [41].

PROMELA is comprised of processes called "proctypes". These processes represent an individual program, in our case an agent. They all run together asynchronously in a random order, similar to threads in Java. One of the restrictions to processes is that there are no methods, however macros and "goto" statements can be used instead. Macros are similar to methods but are merely replications of code; they cannot return values and macros which declare variables will duplicate the variable declarations if called multiple times. The goto statements tell the process to jump to a different section of code. Some other restrictions applied to *PROMELA* are the data types e.g., no strings, sets, stacks or floating points are allowed. The main functions of *PROMELA* used in this thesis are: enumerations, integers, arrays, goto statements, if-statements, and do-statements. The if-statements and do-statements are different from most programming languages because they can handle multiple conditions and are able to halt a process. Here is an example of an if-statement with multiple conditions, where it will either count up or down depending on when a value A is equal a value B or C:

```
byte counter
active proctype counter(){
    if
    ::(A==B)-> count++;
    ::(A==C)-> count--;
    fi
}
```

Multiple conditions can also be used for non-determinism. For example, the process below two things, either count up or count down. Since the values in the if-statement are 'true Spin will non-deterministically choses which action to take. This is typical method for representing non-determinism (i.e., branching in model generation) in Spin.

```
byte counter
active proctype counter(){
    if
    ::(true)-> count++;
    ::(true)-> count--;
    fi
}
```

The do-statement also operates in this fashion. Blocking occurs when Spin reaches an if-statement or do-statement when no option is available, e.g.,

```
Mtype = {P, C} /*mtype is for enumeration*/
mtype turn = P;
active proctype Producer(){
    do
    ::(turn == P)->
         printf(‘‘Produce’’\n");
         turn = C;
    od
}
active proctype Consumer(){
    do
    ::(turn == C)->
         printf(‘‘Consume\n’’);
         turn = P;
    od
}
```

This example shows a producer and a consumer. The variable turn is initially set to P, so when the producer starts to execute it prints “Produce”. The consumer starts to execute but turn isn’t set to C so it halts, once the Producer sets the turn to C it can then execute.

When Spin generates a model it produces a state for every line of code (including print lines). To stop Spin from doing this wrappers known as a “d_step” can be used, these declare a certain section of code as being deterministic. Code inside a d_step is compressed into a single state, if there is non-determinism in the d_step then Spin makes a random choice making it deterministic, e.g.,

```
byte counter
active proctype counter(){
d_step{
        if
        ::(true)-> count++;
        ::(true)-> count--;
        fi;
}

    if
    ::(true)-> count++;
    ::(true)-> count--;
    fi
}
```

The example above shows two non-deterministic if-statements but a deterministic wrap has been used on the first if-statement to render it deterministic. When generating a model for this code Spin will make a random choice for the first if-statement and then create a branch for the second if-statement displaying the two possible outcomes. The model will either look like Figure 4.3 or Figure 4.4.

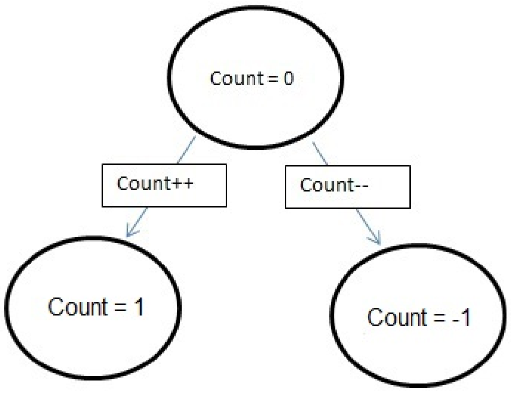


Figure 4.3: Spin randomly chooses to add on the first statement

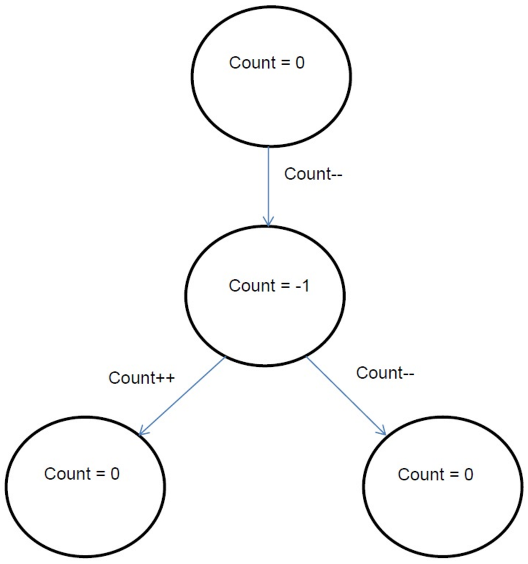


Figure 4.4: Spin randomly chooses to subtract on the first statement

Spin was our chosen model checker mainly for its speed and popularity. Spin is known for being a reliable and fast model checker and its popularity means it is more likely users will have the expertise to use and trust the system. One of the weaknesses of model checking is that it is difficult to create an accurate model of the system which is small enough to model check (weakness 2f in Section 4.2). Spin's input language *PROMELA* helped alleviate this weakness; its high level nature made implementing the semantics easier. Spin's simulation mode also helped with the accuracy by allowing comparisons with the output of the Brahms simulator. Other features such as being able to declare sections of code as deterministic and running the *PROMELA* simulation line by line highlighting all the choice points helped reduce the size of the models generated, alleviating the state-space explosion problem of model checking (problem 2e in Section 4.2).

### 4.2.2 Java Pathfinder

Java Pathfinder (JPF) is a verification tool developed by NASA to model check Java programs. JPF was built in a way so that Java programs execute it to find defects within themselves; properties still need to be specified as input for JPF to operate. JPF is implemented in Java itself and runs as a Virtual Machine (VM) on top of the standard Java VM which causes JPF to run much slower than a normal Java program would. JPF builds a state-transition diagram of the Java program by identifying *execution choices* within the programming code and is then able to traverse all these paths to discover any hidden defects in the code. JPF also has the feature where users can specify scheduling sequences, random values, types of choices and user input. The *state explosion* problem is always an issue for model checking large systems, JPF tries to tackle this issue using *state matching*. *State matching* is when the JPF notices that a choice point it is about to create is similar to a previous choice point encountered, meaning this new path can be abandoned allowing JPF to *backtrack* to this previous choice and take a new unexplored path from there. JPF also allows for manual declaration of where choice points should occur preventing surplus unnecessary states [37].

JPF can be used to identify many defects in Java programs, the core properties JPF will check are *deadlocks* and *unhandled exceptions*. JPF requires manual definitions of properties to be checked, these are mostly checked using "plugins" known as *listeners*. These *listeners* closely monitor all actions JPF makes (not just state transitions) but also actions such as single instructions, creating objects, etc. and will notify once a defect has been discovered. On detection JPF produces a program called a *trace* which identifies the error path and provides full account of the actions that caused the defect.

The first incarnation of JPF performed its model checking through the Spin model checker, described in Section 4.2.1, by translating a program from Java code into *PROMELA*. The current version of JPF builds a model itself using the byte code generated by Java from the *javac* command. JPF does not however apply any form of reduction on the generated model, meaning that the Java programs must have a finite and tractable state space. JPF has been applied to programs having up to 2000 lines of code [37].

In a combined effort with Rungta et al. [71] we use Java Pathfinder (JPF), along with the Brahms semantics generated for this thesis, to model check Brahms simulations. This implementation takes a slightly different verification path than the path to translate the Brahms semantics in *PROMELA*, see Chapter 5 and 12 for more details. JPF was chosen because of its ability to specify where choice points occur, thereby avoiding unnecessary states being produced when generating Brahms models. JPF also has the advantage that Brahms uses hierarchical and high level functions easily represented in Java but difficult in languages such as *PROMELA*.

### 4.2.3 Temporal Logic

In model checking we are required to describe the state of the system and properties we wish to verify. To do this tools such as Spin use temporal logics. Temporal logics are logics which extend classical logics using specific operators which allow us to reason about time [66]. Typical operators used by temporal logics are

- $\Box\varphi$ - $\varphi$ will always be true
- $\Diamond\varphi$ - $\varphi$ will become true some time in the future
- $\bigcirc\varphi$ - $\varphi$ will be true in the next moment
- $\varphi\mathcal{U}\psi$ - $\varphi$ is true until $\psi$ is true
- $\varphi\mathcal{W}\psi$ - $\varphi$ is true unless $\psi$ is true, but $\psi$ does not necessarily need to become true

Temporal logics are ideal for the use in specification and verification of properties of concurrent systems, examples where they have been applied for these purposes are [23, 34, 49]. Temporal logics are ideal for specification because they can easily express properties such as livelock, deadlock and mutual exclusion. The verification itself can be performed using proofs within the logic itself. However, one of the issues with using a standard temporal logic, such as LTL (Linear Temporal Logic), is that the operators cant explicitly express when the property must occur in the future, e.g., a specification can state a property will eventually be true but not that within 5 time steps the property will be true. Important computational properties such as livelock, deadlock and mutual exclusion can be expressed easily and simply in temporal logic making it useful for specification. Verifying that such properties hold for a program specified in temporal logic involves proofs within the logic itself.

There are different types of temporal logic, such as linear [66], branching [5], discrete or dense [19]. However, in this thesis we are only concerned with discrete, linear temporal logics such as LTL which is used by the Spin model checker.

#### LTL Syntax

LTL formulae are constructed using the following connective and proposition symbols.

- A set $\mathcal{P}$ of propositional symbols
- Propositional constants **true** and **false**
- Propositional connecitves $\neg$, $\vee$, $\wedge$, $\Rightarrow$, and $\Leftrightarrow$
- Future-time temporal connectives
  - Unary connectives: $\bigcirc$, $\Box$, $\Diamond$
  - Binary connectives: $\mathcal{U}$, $\mathcal{W}$

The set of well-formed formulae of LTL, WFF, is inductively defined as the smallest set satisfying:

- Any element of $\mathcal{P}$ is in WFF
- **true** and **false** are in WFF
- if $\varphi$ and $\psi$ are in WFF then so are

$$\begin{array}{ccccc} \neg\varphi & \varphi \land \phi & \varphi \lor \phi & \varphi \Rightarrow \phi & \varphi \Leftrightarrow \phi \\ \Diamond\varphi & \Box\varphi & \bigcirc\phi & \varphi\mathcal{U}\psi & \varphi\mathcal{W}\psi \end{array}$$

**LTL Semantics**

A model of a temporal logic can be considered as a sequence of states indexed by the natural numbers $\mathbb{N}$. An LTL model can be represented by a sequence of states such as

$$\mathcal{M} = s_0, s_1, s_2, s_3, ...$$

Where state $s_i$ represents a set of propositions which are satisfied in the $i^{th}$ moment in time. To state when an LTL formula is satisfied we use the notation

$$(\mathcal{M}, i) \vDash \varphi$$

This denotes that the formula $\varphi$ is satisfied in the model $\mathcal{M}$ in the $i^{th}$ state where $i \in \mathbb{N}$. To express a formula that is not satisfied in this model at the same particular state we would use the notation $(\mathcal{M}, i) \nvDash \psi$. To state that a formula is *valid*, i.e., it is satisfied in every state, we express it as $\vDash \varphi$.
The semantics for a proposition is defined by

$$(\mathcal{M}, i) \vDash p \text{ iff } p \in s_i \text{ where } p \in \mathcal{P}.$$

The semantics for a standard propositional connective of classical logic is, for example

$$(\mathcal{M}, i) \vDash \varphi \land \psi \text{ iff } (\mathcal{M}, i) \vDash \varphi \text{ and } (\mathcal{M}, i) \vDash \psi.$$

The semantics of a negated formula is defined as follows

$$(\mathcal{M}, i) \vDash \neg\varphi \text{ iff } (\mathcal{M}, i) \nvDash \varphi.$$

The semantics for the unary future-time temporal connectives are defined as follows

$$(\mathcal{M}, i) \vDash \bigcirc\varphi \text{ iff } (\mathcal{M}, i+1) \vDash \varphi.$$

$$(\mathcal{M}, i) \vDash \Diamond\varphi \text{ iff } \exists j \geq i \text{ s.t. } (\mathcal{M}, j) \vDash \varphi.$$

$$(\mathcal{M}, i) \vDash \Box\varphi \text{ iff } \forall j \geq i \text{ s.t. } (\mathcal{M}, j) \vDash \varphi.$$

The semantics for the binary future-time temporal connectives are defined as follows

$$\begin{gathered} (\mathcal{M}, i) \vDash \varphi\mathcal{U}\psi \text{ iff } \exists k \geq i \text{ s.t. } (\mathcal{M}, k) \vDash \psi \\ \text{And } \forall i \leq j < k \text{ then } (\mathcal{M}, j) \vDash \varphi \end{gathered}$$

$$(\mathcal{M}, i) \vDash \varphi\mathcal{W}\psi \text{ iff } (\mathcal{M}, i) \vDash \varphi\mathcal{U}\psi \text{ or } (\mathcal{M}, i) \vDash \Box\varphi.$$

## 4.3 Other Formal Verification Techniques

Model checking is not the only available technique for performing formal verification. Some other typical mechanisms for performing formal verification are as follows:

- Dynamic Fault Monitoring: also referred to as Runtime Verification because properties are represented by a finite-state automaton (FSA) and checked during an actual execution of the system, i.e., a property $A$ represented as a FSA is used to scan an execution, $\alpha$, to see whether it satisfies $A$. If at any time an error is flagged, i.e., at some point in the execution the requirement is not satisfied ($\alpha \not\models A$), then the violation is investigated [36].

- Formal Proof: the use of mathematics to logically prove whether or not a property will hold in a system. To do this we require a mathematical representation of the behaviour of the system, a logical formula $\alpha$, and a requirement of the system also represented by a logical formula, $A$. Verifying that property $A$ holds involves proving that $\alpha$ implies $A$ is a theorem in the system, i.e., $\vdash \alpha \Rightarrow A$. This process can be automated to deduce whether a requirement holds when given the specification in a traditional logic, such as temporal logic, and a logical specification of the system's behaviour in the language of a theorem prover [32].

- Equivalence Checking: whereby the intended behaviour of a system is represented by a specification $S$ and the actual implementation of the system, $I$. These representations are defined as states in transition systems, and then it is shown that $S$ and $I$ are equivalent [48].

These techniques all have their applications in various domains however little work has been done in applying these techniques to verify multi-agent systems. To apply these techniques to human-agent teamwork scenarios additional work would be required to develop adequate tools to perform the verification. Model checking has however been the preferred technique for verifying multi-agent systems, demonstrating its use with model checkers such as Spin and Java Pathfinder used to verify multi-agent systems. Model checking tools for agents have also been developed such as MCMAS (Model Checking Multi-Agent Systems) [54] and MCAPL (Model Checking Agent Programming Languages) [6] for verifying multi-agent systems programmed in a variety of agent languages.

## 4.4 Agent Verification

The autonomous nature of agents and their ability to work together in a team leads to questions about their behaviour. We need to know if they will do what they are required to do and whether or not they will be able to coordinate their efforts to complete the task. Verification techniques are used in determining whether an agent, or team of agents, will satisfy a system's design objectives. Since agents work in teams, issues such as deadlocks, livelock and race conditions become apparent [12].

Agent programming paradigms such as the beliefs-desires-intentions (BDI) have the concept that agents act *rationally*. This principle of acting rationally means that it is possible to model the various actions of the agents, allowing applications of model checkers [12]. One of the reasons agent based systems have become so popular is due to their ability to automatically handle large systems with superior speed and accuracy

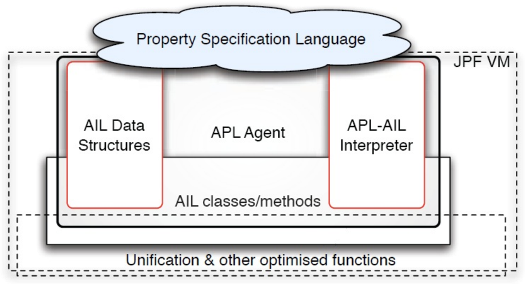


Figure 4.5: Overview of the AIL Architecture [28]

to any human counter-part [89]. For this reason agents are now being incorporated into safety critical systems, where unexpected errors can cost time, money, and endanger lives. Being able to logically test such a system before application makes formal verification appealing.

### 4.4.1 Agent Infrastructure Layer

An already existing technique for verifying multi-agent systems is the Agent Infrastructure Layer (AIL). AIL is a collection of Java classes developed to unify frameworks of the variety of modelling formalisms available, particularly agent programming languages. The collection of Java classes within AIL contain clear and adaptable semantics and are able to implement interpreters for various agent languages. Programs interpreted by AIL are then able to be model checked by Agent Java Pathfinder (AJPF); an extension of the Java Pathfinder model checker customised to support AIL-based interpreters. AIL can be perceived as a basis to which agents, that have been programmed in various languages, can co-exist in a multi-agent system and still provide the capability of model checking these agents using AJPF. AIL does this by identifying the key *operations* that many BDI languages use and incorporates them into an AIL *toolkit* [28]. Fig. 4.5 shows a diagram by Dennis et al. [28] illustrating how the AIL architecture fits within the JPF virtual machine.

## 4.5 Verification of Agent Languages

The verification of agents and multi-agent systems is currently an on-going research topic, with papers such as [25, 70, 6, 29, 1, 90] performing verification on various agent programming languages. In this section we will describe how agents have previously been verified to provide an insight into the possible different routes available for agent verification.

Bordini et al. [6] use the AIL toolkit and the MCAPL (Model Checking Agent Programming Languages) interface to model check a range of agent programming languages. They use the AIL toolkit as an interpretation tool (as it encompasses the main concepts of agent based languages) and the MCAPL interface to perform model checking via AJPF (Agent Java Pathfinder). Using this approach they verified properties of programs programmed in agent languages such as 3APL [39], AgentSpeak(L) encapsulated by Jason [13], GOAL [82] and SAAPL (Simple Abstract Agent Programming Language) [83]. Figure 4.6 shows the approach used by Bordini et al. [6]. Where AIL represents the semantics of the agent languages, AgentSpeak, 3APL, etc. This

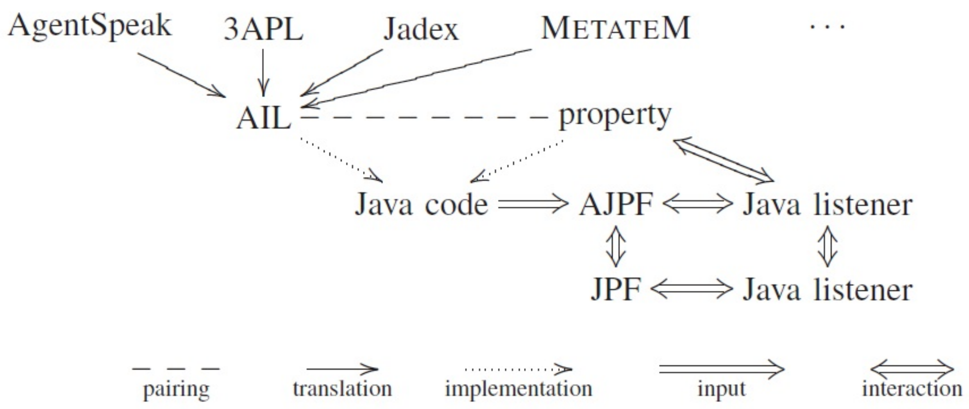


Figure 4.6: Overview of Bordini et al's [6] approach

representation of the semantics of these languages, along with a specified property of the system is translated into Java code. AJPF and JPF together form a model of the Java code presented, this model is then traversed with Java listeners checking for states where the property is violated.

Boer et al. [25] produce a verification framework for goal orientated agents, specifically for agents programmed in the language GOAL. In [25] Boer et al. describe a formal operational semantics for GOAL and construct a temporal logic specifically to prove properties of GOAL agents, which incorporates the belief and goal modalities used in GOAL agents.

In [8] Bordini et al. produce a variation of the AgentSpeak(L) language called AgentSpeak(F) and show how they transform programs written in AgentSpeak(F) into *PROMELA* for Spin verification.

McCallum et al. [56] produce a flexible and expressive framework for the verification and analysis of agents taking part in multiple organisations with distinct roles and disparate obligations. Rather than using an existing agent programming language McCallum et al. citemccallum2006verification produce their own system for modelling agents using organisations, roles, actions, and obligations using Sicstus Prolog [61]. To verify these models McCallum et al. [56] combine the model's rules with a constraint solver [55] which determines whether all the agent's obligations can be fulfilled.

# Chapter 5

# Analysis of the Project

There are a number of routes we could explore to develop verification techniques for human-agent teamwork. In this chapter we aim to provide the reader with an overview of the paths that were considered through the course of the project including the path that was chosen. We present the reader with our process for selecting a framework for representing humans and agents, the verification of this framework, and the method we chose to implement our tool. An overview of the possible paths is provided in 5.1.

## 5.1 Representing Humans, Agents and Robots

This first challenge was how to represent the entities whose actions and interactions are to be verified. For verification purposes the framework needed to be a simulation framework; able to state the actions performed, when they were performed, for how long and the implications of the actions. These requirements were set to allow for easy modelling of human-agent teamwork scenarios for verification. Stating the possible actions and when they can be performed allows for describing branching in the models, the timing of the actions allows for reasoning over temporal properties (e.g., the task will always be completed within the required time frame), and the implications of the actions describes the changes that will be made to the current state. The most desired requirement of this framework was the ability to represent humans, agents and robots accurately. The difficulty of this is the human aspect, what framework can model humans? To model humans the framework needs to be able to represent human behaviour such as multi-tasking, communication, making mistakes, etc. However, to accurately model a human is an impossible task because people can be irrational and unpredictable. For this reason it was decided to only consider expected possible behaviours of the person in the given situation. The languages and frameworks considered for this project were: 3APL, AgentSpeak/Jason, GOAL and Brahms. 3APL, AgentSpeak/Jason and GOAL were considered due to their popularity amongst the agent community, however none of these were designed to model humans or had been previously been used to do so. The possibility of extending these languages to model humans was considered but discounted because the Brahms framework had been designed to model humans and been used to do so.

## 5.2 Verification of Brahms

### 5.2.1 Breaking down Brahms for Verification

As Brahms is a large complex system, rather than try to verify the whole of Brahms we identified a core of Brahms such that it still retains the essence of Brahms but without features that do not offer additional functionality. The workings of all the Brahms functions were analysed and evaluated based on their importance and whether or not they could be replicated using other functions. The results of this work found that only a few core functions could be removed from Brahms. The core functions that could be removed were found to be *Composite Activities* and *Create Agent/Object Activities*. We identified that *Composite Activities* could be replicated using other functions, since they are a collection of activities grouped together. The creation of new agents and objects after initialisation was deemed not to be a necessity of the Brahms framework and could also be removed, especially since this could make verification difficult, e.g., non-deterministic creation of agents could cause a state space explosion when model checking a simulation. Functions such as: *broadcast*, communicate to all agents; *detectArrivalIn*, detect when another agent arrives at an agent's location; and *detectDepartureIn*, detect when an agent leaves an agents location, where considered to be replicable using other functions. The functions that could be considered superfluous for verification were: *min_duration*, set a lower bound for the duration of an activity; label, name the agents, etc. for visualisation in the Brahms output; and *gestures*, simulate a wave or handshake, etc. Table 5.2.1 shows a list of all the keyword in Brahms, keywords in bold font are functions which have been implemented in the project.

### 5.2.2 How to Verify Brahms

With a suitable framework to model the human-agent teamwork we needed to consider how to verify the simulations produced by Brahms. Model checking, see Chapter 4.2, was the technique decided for the verification. The reasons behind this were that it is an automated process and it has already been used to verify agent behaviour [45, 27, 86]. Figure 5.1 describes possible options for verifying Brahms models. The path at the top describes translating Brahms into an agent language which already has model checking capabilities; the middle path describes translating Brahms for verification via the Agent Infra-Structure Layer and Java Pathfinder; and the final path describes storing the Brahms model in Java data structures before translating into the input language of an existing model checker.

We decided to create Java data structures to hold the details of the scenario for translation into the language of the desired model checker. This choice was made because the intermediate representation of the model in Java would act as a central hub for easier translation to multiple input languages for model checkers. This would allow for more than one model checker to be used where results could be compared and different types of verification could be performed; possibly probabilistic and epistemic properties as well as temporal. The other options were discounted because verification via AIL and Java Pathfinder is slow and Brahms only has similarities to the Beliefs-Desires-Intentions paradigm, making it difficult to translate into BDI languages. Within our chosen solution there was also an option to create our own model checker, however the availability of many fast and reliable model checkers such as NuSMV, Spin, PRISM, etc. we discounted this option. Developing our own model checker would mean questioning

| | | | |
|---|---|---|---|
| ActiveClass | delete | mod | ActiveConcept |
| **move** | ActiveInstance | **detect** | name |
| **Agent** | **detectable** | destination | min_duration |
| nowork | **Area** | **detectables** | **not** |
| **object** | **Class** | detectArrivalInSubAreas | package |
| **part of** | ConceptualClass | detectDeparture | InSubAreas |
| **path** | ConceptualConcept | **Concept** | detectDepartureIn |
| display | **primitive_activity** | ConceptualObject | **distance** |
| div | **private** | **Group** | **do** |
| **protected** | **Object** | **priority** | GeographyConcept |
| **double** | **public** | **abort** | end_activity |
| end_condition | quantity | action | **extends** |
| random | **activities** | put | about |
| factframe | receive | **agent** | **FALSE** |
| **fc** | **relations** | **area** | **foreach** |
| **repeat** | **area1** | **relation** | **and** |
| **forone** | resource | **area2** | gesture |
| get | **send** | assigned | **group** |
| source | **attributes** | resources | **areadef** |
| impasse | **string** | **bc** | import |
| icon | symbol | broadcast | inhabitants |
| **then** | **class** | super | **boolean** |
| **initial_beliefs** | **thoughtframe** | **collectall** | **initial_facts** |
| **instanceof** | time_unit | **complete** | **int** |
| **to** | composite_activity | **thoughtframes** | **communicate** |
| is | toSubAreas | conceptual_class | java |
| jimport | type | **conclude** | **known** |
| unassigned | **continue** | **TRUE** | conceptual_object |
| **knownval** | unknown | cost | listof |
| **location** | with | create_area | **long** |
| **when** | create_object | **variables** | create_agent |
| map | **whenever** | **current** | **max_duration** |
| **workframe** | dataframe | **memberof** | **workframes** |
| **dc** | **AreaDef** | detectArrivalIn | |

Table 5.1: List of all Brahms keywords, keywords marked with an asterix have been implemented in the project

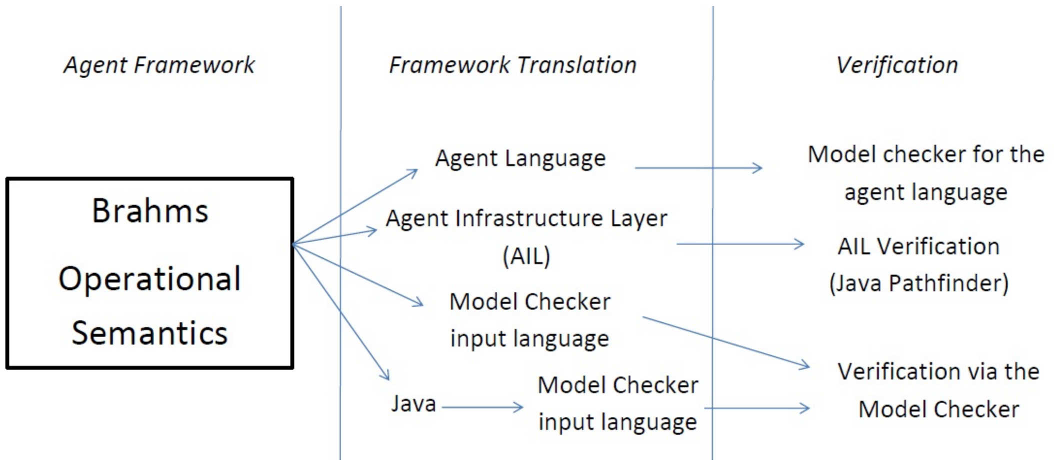

Agent Framework
Framework Translation
Verification
Brahms Operational Semantics
Agent Language
Agent Infrastructure Layer (AIL)
Model Checker input language
Java
Model Checker input language
Model checker for the agent language
AIL Verification (Java Pathfinder)
Verification via the Model Checker

Figure 5.1: Overview of the possible paths to verifying Brahms

the efficiency and reliability of the model checker as well as any translation performed. Since the chosen path allows for possible translations into many different model checkers the choice of model checker wasn't of such great importance. The model checker chosen was the Spin model checker as this is a very popular model checker, and has been evolving since it was made publicly available in 1991. Since then its efficiency, functionality and the confidence in its verification results has increased over time. Spin's input language *PROMELA* is a higher level language than most other model checker input languages, making it easier to represent the Brahms semantics. Spin also has the ability to run *PROMELA* code as a simulation, this makes it possible to compare the output of the *PROMELA* translation with the output of the Brahms simulation. The Spin model checker is discussed in more detail in Section 4.2.1.

## 5.3 The Implementation

Here we explain the process conducted during the implementation of the project. Firstly we needed to develop a formal representation of the semantics. The only previous formal representation available for Brahms was the implementation code, which due to NASA copyright restrictions we had no access. The development of this operational semantics was performed with the aid of the creators of Brahms: Maarten Sierhuis and Ron van Hoof. We used simple Brahms simulations to develop our understanding of Brahms in order to construct the semantics. The formal semantics were published at CLIMA 2011 [79]. The operational semantics for Brahms can be found in Part II, Section 6 of this thesis.

The second stage of the project was to parse the Brahms code and store the Brahms simulation data into Java data structures. The operational semantics produced in [79] acted as a blueprint for storing this data, i.e., matching up the Java data structures to the structures represented in the tuples of the operational semantics. A Brahms parser was developed using ANTLR (Another Tool for Language Recognition), where the syntax is expressed in Backus-Naur Form (BNF, a notation technique for context-free grammars) and the details of the syntax are passed on to a Java class.

The third stage was to implement the Brahms semantics in *PROMELA*. This was performed by following the Brahms operational semantics in Part II, Section 6. Each rule was created in *PROMELA* code, tested on simple Brahms simulations using Spin's simulation runs and the results then compared against results of the simulation generated by Brahms. Steadily each semantic rule was created and tested in this way. The Java code was then developed to automatically generate this *PROMELA* translation using the simulation's data which was parsed from a Brahms simulation. Restrictions implemented in *PROMELA* meant an instance of Brahms semantics had to be created for each Brahms agent and object. This process was also performed iteratively, gradually adding in Brahms functions and testing the output against a Brahms simulation to analyse the correctness of the Brahms translation. The final stage was to produce scenarios and analyse the verification results. The scenarios were developed in increments where each increment was translated to *PROMELA*, tested in a Spin simulation (for results comparison, in case of any translation errors), and verification of a simple property was performed.

# Part II

# Formal Verification of Brahms

Chapter 6

# Formal Semantics of Brahms

This Chapter describes the formal operation semantics we produced for Brahms framework. Here we describe the notation used, the structure of the tuples and the rules themselves. In this chapter we present the reader with 41 rules to describe the formal semantics of Brahms. With so many rules it is difficult to get a clear picture of how Brahms operates, so we present the reader with Figures 6.1 and 6.2 to help explain the semantics.

This section will follow the flow through the Brahms semantics in Figures 6.1 and 6.2. It should be noted that this is only the core aspects of the Brahms semantics, functions such as suspension, detectables, variables, etc. have been removed for simplicity. The Figures 6.1 and 6.2 have been drawn with a relation to traditional data flow diagrams where rectangles with rounded edges start the data flow, rectangular boxes with sharp edges represent a process, diamond boxes represent a yes/no choice, and an elongated oval represents the termination state. Arrows are used in the diagrams to show how the flow moves from process to process, arrows emanating from a diamond are labelled either yes or no to signify which choice they represent. To help describe the flow through these diagrams the states have been labelled A1-A22 and S1-S8. The agent's states are identified using A and the scheduler's by S. It should be realised that the agent's and the scheduler diagrams are not mutually exclusive, i.e., some agent states require the scheduler to be in a certain state before the agent is moved onto another state. Some process states in the diagrams are shaded, these are to identify that non-determinism can occur in these states, e.g., state A14 is shaded and refers to updating a belief or a fact, the non-determinism here occurs because belief and fact updates are assigned a 'certainty', i.e., a percentage chance that the update will occur.

The scheduler, in Figure 6.2, starts off by initialising everything from agents, to objects, etc. in state $S1$. During this initialisation the scheduler informs the agents to start executing, the scheduler then moves into $S2$ where it waits for a response from all the agents. The agents, in Figure 6.1, start off by moving into $A1$ where they initialise themselves, then move on to $A2$ where they then wait for the scheduler. Once the agents have received the command from the scheduler to start executing they move into state $A4$ where they generate a set containing all active thoughtframes. The agent then cycles through states $A4$, $A5$ and $A6$ where it executes all thoughtframes in the set and checks for more thoughtframes to become active until no more thoughtframes are active. The box $A6$ is shaded because the thoughtframes are chosen non-deterministically. The agent then moves into state $A7$ where a set of all active workframes is selected, if this set is empty then the agent moves to $A9$ to set itself as idle. If there are active workframes then the agent moves to state $A11$ where it randomly selects one of these workframes, again $A11$ is shaded due to this random choice. The semantics then check whether the workframe is empty or not in $A10$, if the workframe deed stack is empty then the agent is directed back to state $A4$ to process its thoughtframes. If the workframe deed stack is not empty then it pops the top element off the stack in $A12$. $A13$ then checks if the event is an activity or a conclude. If the event is a conclude then the agent moves to state $A14$ where it processes this conclude, this state is shaded because the belief and fact attributed to this conclude may or may not be updated based upon the belief and fact certainty of the conclude. If the event is an activity then the agent moves to state $A15$ where it selects a duration for this activity between the minimum and maximum value, the box is shaded to represent this non-determinism. The agent then sends this value to the scheduler in state $A16$ and waits for a response from the scheduler. Once the scheduler receives all durations from all the agents it moves to state $S4$ and calculates which is the shortest. If all the agents had found no active workframes in state $A8$ and moved to state $A9$ then they would all have sent the scheduler a duration of -1, if this is the case then the scheduler will be directed to states $S7$ and $S8$ from state $S5$ to terminate the simulation. If the scheduler did find a duration greater than -1 in $S5$ then it moves its clock forward by this duration in state $S6$ and moves back to waiting for a duration from all the agents in state $S2$. The agents will now have received a duration from the scheduler and will move from $A16$ into $A18$, if the scheduler had sent a -1 for the duration then they will move to $A19$ and terminate. When the scheduler sends a duration greater than -1 the agents move into state $A22$ where they check to see whether they have an activity to deduct time from, if they had set themselves idle then they would not have a current activity to do this with. They then process states $A21$ and $A20$ to update their clocks and deduct time from their activities, they are then directed back to state $A10$ to continue popping events off the deed stack. Once the deed stack becomes empty they will be directed from state $A10$ to $A4$ to start processing thoughtframes and eventually move onto the next workframe.

## 6.1 Semantics: Notation

The following conventions refer to components of the system, and agent and object states.

**Agents:** $ag$ is used to express the identity of an *agent*, e.g., $ag_{Alex}$ would represent an agent named $Alex$, while $Ag$ represents the *set* of all agents. When referring to arbitrary agents we use names such as $i$ and $j$, and when we are referring to the number of agents we use $n$. For example, when we use the term $\forall ag_i \in Ags$ we are referring to all arbitrary agents in the set of all agents, and when we are using something that requires two arbitrary agents, such as communication, we will say that arbitrary agent $ag_i$ communicates to arbitrary agent $ag_j$.

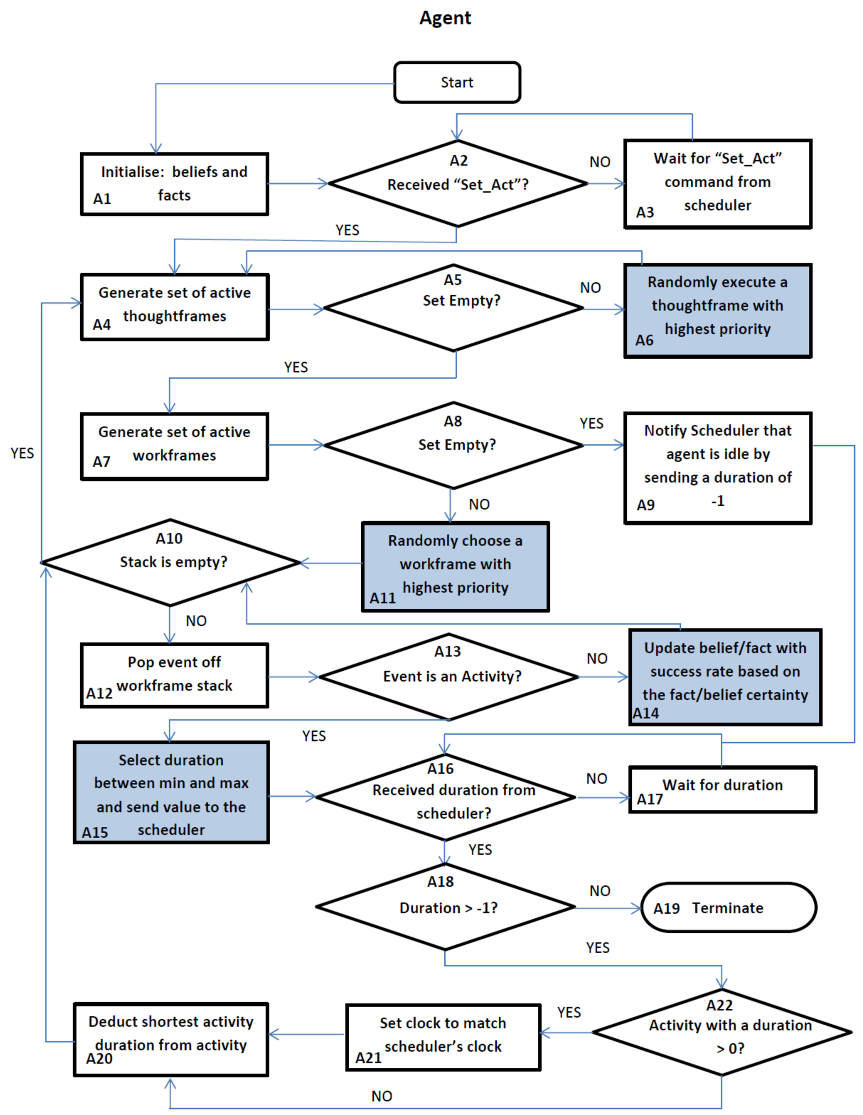
**Beliefs:** $b$ represents the atomic formula of a *belief*, while $B$ represents a *set* of beliefs. In Brahms the overall system may have beliefs which are represented by $B_\xi$.

**Facts:** $f$ represents the atomic formula of a *fact*, while $F$ represents a *set* of facts.

**Workframes:** Workframes are represented as the tuple

$$\langle W^g, W^{pri}, W^r, W^D, W^V, W_{ins}\rangle$$

Where



Agent
Start
A1 Initialise: beliefs and facts
A2 Received "Set_Act"?
NO
A3 Wait for "Set_Act" command from scheduler
YES
A4 Generate set of active thoughtframes
A5 Set Empty?
NO
A6 Randomly execute a thoughtframe with highest priority
YES
A7 Generate set of active workframes
A8 Set Empty?
YES
A9 Notify Scheduler that agent is idle by sending a duration of -1
NO
YES
A10 Stack is empty?
A11 Randomly choose a workframe with highest priority
NO
A12 Pop event off workframe stack
A13 Event is an Activity?
NO
A14 Update belief/fact with success rate based on the fact/belief certainty
YES
A15 Select duration between min and max and send value to the scheduler
A16 Received duration from scheduler?
NO
A17 Wait for duration
YES
A18 Duration > -1?
NO
A19 Terminate
YES
A20 Deduct shortest activity duration from activity
A21 Set clock to match scheduler's clock
YES
A22 Activity with a duration > 0?
NO

Figure 6.1: Overview of a Brahms Agent's Semantics

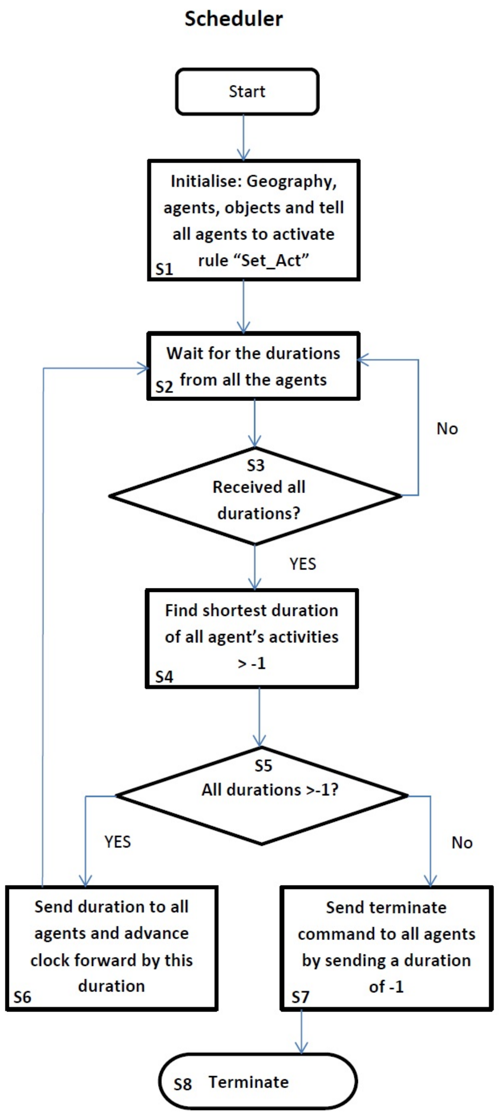

Scheduler
Start
Initialise: Geography, agents, objects and tell all agents to activate rule "Set_Act"
S1
Wait for the durations from all the agents
S2
No
S3
Received all durations?
YES
Find shortest duration of all agent's activities > -1
S4
S5
All durations >-1?
YES
No
Send duration to all agents and advance clock forward by this duration
S6
Send terminate command to all agents by sending a duration of -1
S7
S8 Terminate

Figure 6.2: Overview of the Scheduler's Semantics

- $W^g$ is the workframe's guard.
- $W^{pri}$ is the workframe's priority. Priorities are represented in Brahms as a natural number, $\mathbb{N}$, however in this semantics we add decimal values to these numbers to account for priorities of suspended, impassed and current workframes over generic workframes yet to be instantiated.
- $W^r$ is the workframe's repeat variable. The repeat variable can take the values $true$, $false$, and *once*.
- $W^D$ is the workframe's detectables. This is a tuple $\langle d^g, d^{type}\rangle$, where $d^g$ represents the detectables guard condition and $d^{type}$ represents the detectables type; *impasse*, *continue*, *complete*, or *abort*.
- $W^V$ is the workframe's variables, $\beta^V$ for the current workframe. A single variable is identified using $v$, each variable has a type which is identifed by $v^{type}$ which can take the values $forone$, $foreach$, and *collectall*.
- $\langle W_0 ... W_n\rangle$ is a set representing instansiations of the workframe. These instantiations are necessary when a workframe contains variables, an instantiation is created for every possible combination of assignments that the variables can have; variables can be assigned to agents, objects and locations.
- $W^{Concludes}$ is used to represent all the conclude statements inside the workframes stack of instructions. This is used when a workframe has been instructed to process only concludes and ignore actvities.

When referring to workframes $W$ refers to any arbitrary workframe, $\beta$ represents the current workframe, e.g., $\beta^{pri}$ would refer to the current workframe's priority, and $\mathcal{WF}$ represents a set of workframes. Occasionally to save space in the tuple we represent the first 6 elements of the workframe tuple as $W_d$, i.e., the workframe's header data. This shortened form of the tuple looks as follows $\langle W_d, W_{ins}\rangle$ where $W_{ins}$ represents the stack of instructions the workframe is to perform, such as concludes and activities.

**Thoughtframes:** Thoughtframes are represented in a similar fashion except $\alpha$ represents the *current thoughtframe*, *TF* represents a *set* of thoughtframes, while $\mathcal{T}$ represents any arbitrary *thoughtframe*.

**Activities and Concludes:** Activities and concludes are broken down into the following types

- $\texttt{Prim\_Act}^t$ is a primitive activity of duration $t$.
- $\texttt{Comms(}ag_j, b\texttt{)}^t$ is a communication activity to agent j, sending belief b with a duration $t$.
- $\texttt{Move(Loc = new)}^t$ is a move activity from the current location `Loc` to the new location `new` $t$.
- *conclude*$(b)$ is a conclude asserting the belief $b$.
- *conclude*$(f)$ is a conclude asserting the fact $f$.

**Environment:** In this semantics additional details outside of the agent's and object's own perceptions are referred to as belonging to the environment. To represent this environment we use the identification $\xi$.

**Time:** $T$ represents the time in general, while a specific duration for an activity is represented by $t$. The time $T$ is always associated with either an agent, object or the environment, e.g., $T_i$ refers to current time of agent $ag_i$ and $T_\xi$ refers to the time of the global clock, or the system clock, in the environment. Time is represented as a natural number, $\mathbb{N}$, with the exception of the termination condition which takes the value of -1.

**Stage:** The semantics are organised into "stages". Stages refer to the names of the operational semantic rules that may be applicable at that time, wild cards ($*$) are used to refer to multiple rules with identical prefixes. There is also a "*fin*" stage which indicates an agent is ready for the next cycle, and an "*idle*" stage which means it currently has no applicable thoughtframes or workframes. To describe the stage of an agent i we use the notation $ag_i^{stage}$

**Methods:** To keep the semantic rules as simple as possible we shorten some actions into Java like method calls. The methods used are as follows:

- $MinTime(\forall ag_i | T_i \in B_\xi)$. This method is used when all the agents have informed the scheduler of when their next activity is due to finish. This method examines all the durations of all the agent's activities and identifies which is the smallest, $\forall ag_i | T_i \in B_\xi$ expresses that the method examines the durations, in the environment's belief base, for all the agents.
- $Max\_Pri()$. This method is used to find the thoughtframe or workframe of the highest priority, e.g., $Max\_Pri(\forall \mathcal{T} \in TF_i | B_i \models \mathcal{T}^g)$ finds the thoughtframe in the set of all thoughtframes such that the thoughtframe's guard condition is met in the agent's belief base.
- $selectVar()$. This method matches all the agents/objects/locations that meet the requirements set out in the workframe or thoughtframe's guard condition and assigns each set of agents etc. to a workframe or thoughtframe instance.
- $Random()$. This method is used to show a that random selection is being made, e.g., $Random(W_0...W_n)$ randomly selects a workframe out of the set of workframes $W_0...W_n$.
- $concludes(W_1...W_n)$. This method is used in the rule $\mathsf{Var\_all}$, it takes all the workframes instances $W_1...W_n$ and extracts all the conclude statements from it. This rule is needed because a *collectAll* variable takes all the conclude statements from every instance and processes them at the same time.

## 6.2 Semantics: Structure

The operational semantics are broken up into two parts; the scheduler semantic rules and the agents semantic rules. We use a 5-tuple description (shown in Definition 1) to represent the state of the scheduler, a 9-tuple description (shown in Definition 2) to represent the state of the agent, and a transition rule (shown in Definition 3) to show how the states transform. We use first-order logic with set theoretic operations, but restricted to the sets available within the semantic structures, to express when the rule is active and to state how the tuple changes when the rule fires.

**Definition 1.** *The system configuration is a 5-tuple description* $\langle Ags, ag_i, B_\xi, F, T_\xi \rangle$ *where*

$$\frac{\langle \mathit{Starting\,Tuple}\rangle}{\xrightarrow[\mathit{Conditions\,Required\,For\,Actions}]{\mathit{Actions\,Performed}}}$$
$$\langle \mathit{Resulting\,Tuple}\rangle$$

Figure 6.3: Simplified Template for the Operational Semantics Transition Rules

| | |
|---|---|
| $Ags$ | *- is the first element of the tuple in the set of all agents;* |
| $ag_i$ | *- is the second is the current agent under consideration;* |
| $B_\xi$ | *- is the third is the belief base of the system;* |
| $F$ | *- is the fourth is the set of facts in the environment;* |
| $T_\xi$ | *- is and the fifth is the current time of the system;* |

**Definition 2.** *The agents and objects within a system have a 9-tuple representation* $\langle ag_i, \mathcal{T}, W, stage, B, F, T_i, TF, WF\rangle$ *where*

| | |
|---|---|
| $ag_i$ | *- is the first element is the identification of the agent;* |
| $\mathcal{T}$ | *- the second is the current thoughtframe;* |
| $W$ | *- the third is the current workframe;* |
| $stage$ | *- the fourth is the stage the agent is at;* |
| $B$ | *- the fifth is the set of beliefs the agent has;* |
| $F$ | *- the sixth is the set of facts;* |
| $T_i$ | *- the seventh is the time of the agent;* |
| $TF$ | *- eighth is the set of thoughtframes the agent has;* |
| $WF$ | *- and the ninth is the agent's set of workframes.* |

*The fourth element of the tuple, the stage, explains which set of rules the agent is currently considering or if the agent is in a finish (fin) or idle (idle) stage.*

**Definition 3.** *A transition rule is denoted by Figure 6.3 where*

| | |
|---|---|
| $\langle \mathit{Starting\,Tuple}\rangle$ | *- represents the system's or the agent's tuple before the rule is applied;* |
| $\mathit{Conditions\,Required\,For\,Actions}$ | *- states the conditions required for the rule to fire;* |
| $\mathit{Actions\,Performed}$ | *- represents the actions performed by the rule;* |
| $\mathit{Resulting\,Tuple}$ | *- represents the tuple after the rule has fired;* |

### 6.2.1 Timing

The timing in Brahms works by the use of a global system clock coupled with agents having their own internal clocks. The system scheduler asks each agent how long each of their activities are, finds the time of the shortest activity and then tells each agent to move their clock forward by this time. However it should be noted that during a simulation agents are not aware of their internal clocks, the clocks are used behind the scenes to keep all agents synchronised. Traditionally Brahms simulations are modelled with a 'Clock' agent to broadcast a simulation time to all the agents to give them an awareness of time. Workframes that the agents are currently working on can be interrupted if a new higher priority thought/workframe becomes active, or if a fact change in the system causes an impasse via a detectable. The following structure shows how agents are moved forward in time by the scheduler. It shows every agent from $Ag_0$ to $Ag_n$ being moved forward in time, once an agent moves forward in time it reaches an intermediary point $X$ where it will then make a $Choice$ on its next set of actions. $\xi$ represents the scheduler, showing that all the agents and the scheduler move as one from time point to time point.

$$\frac{\begin{array}{c} Ag_0 \xrightarrow{LocalClock+t} X, X \xrightarrow{Choice} Ag'_0 \\ . \\ . \\ . \\ Ag_n \xrightarrow{LocalClock+t} X, X \xrightarrow{Choice} Ag'_n \end{array}}{\xi \xrightarrow{LocalClock+t} \xi'}$$

## 6.3 Semantic Rules

### 6.3.1 Scheduler Semantics

The scheduler is the central system of Brahms, it decides when and what value the global clock will take and it starts and terminates the execution of the system. For the scheduler to start/continue execution all agents must be in a '*fin*' (*finished*) or 'idle' (*idle*) state and the global clock must not be less than zero. For Brahms to terminate all the agents need to be in an idle state where they have no workframes/thoughtframes which have their guard condition met.

**Sch_run**. Start agents running for the new clock tick. This rule states that if all agents in the system are either in a finished or idle state and the global clock is not minus one then all agents are directed to the '*Set_Act*' semantic rule.

<u>Rule:</u> Sch_run

$$\langle Ags, ag_i, B_\xi, F, T_\xi \rangle$$

$$\xrightarrow[\forall ag_i \in Ags | ag_i^{stage} \in \{fin, idle\} \wedge (T_\xi \neq -1)]{ag_{i'} = ag_i[ag_i^{stage} \in \{fin, idle\} / ag_i^{stage} \in \{Set\_Act\}]}$$

$$\langle Ags, ag_{i'}, B_\xi, F, T_\xi \rangle$$

**Sch_rcvd**. Receives the activity durations from all agents. This rule identifies when the Scheduler has received all the durations from all agents. It states that if all agents are in a waiting or idle state then the Scheduler will check all the agents end activity times, calculate the smallest value and set its time to this. For this rule to activate all the agents need to be considering the rules *Pop_PA*∗, *Pop_MA*∗ or *Pop_CA*∗ where * represents a wild card for any suffix of the word.

<u>RULE:</u> Sch_rcvd

$$\langle Ags, ag_i, B_\xi, F, T_\xi\rangle \xrightarrow[\forall ag_i \in Ags | stage \in \{Pop\_PA*, Pop\_MA*, Pop\_CA*)\} \vee idle, (T_\xi \neq -1)]{T_{\xi'} = T_\xi[T_\xi / T_\xi + MinTime(\forall ag_i | T_i \in B_\xi)]} \langle Ags, ag_i, B_\xi, F, T_{\xi'}\rangle$$

the notation $ag_{i'} = ag_i[ag_i^{stage} \in \{fin, idle\} / ag_i^{stage} \in \{Set\_Act\}]$ indicates that the stage value of $ag_i$ has been replaced by *Set_Act*.

**Sch_term**. This termination condition happens when all agents are in an idle state, to signal the termination it sets the global clock to minus one.

<u>RULE:</u> Sch_Term

$$\langle Ags, ag_i, B_\xi, F, T_\xi\rangle \xrightarrow[\forall ag_i \in Ags | stage \in \{idle\}]{T_{\xi'} = T_\xi[T_\xi / T_\xi = -1]} \langle Ags, ag_i, B_\xi, F, T_{\xi'}\rangle$$

### 6.3.2 Agent Semantics

The Brahms system operates on a simple cycle of handling:

*Thoughtframes* → *Detectables* → *Workframes*

### 6.3.3 Set_* rules

Rules with the prefix of 'Set_*' are used at the start of every cycle. These are used to determine whether or not the agent/object will be *idle* (no active workframe or thoughtframe) for the duration of this cycle. Those that are *idle* will do nothing until this rule is next invoked by the system, those that are not *idle* are directed to checking thoughtframes.

**Set_Act**. If the agent is currently checking 'Set_*' rules, has no current thoughtframe and the agent has a workframe or a thoughtframe with its guard condition met then this rule directs the agent to the 'Tf_*' rules. Whether or not the agent has an active workframe or not is not an issue.

<u>RULE:</u> Set_Act

$$\langle ag_i, \alpha, \beta, Set\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\alpha \in \{\emptyset\} \wedge (\exists \mathcal{T} \in TF_i | B_i \models \mathcal{T}^g \vee \exists W \in WF_i | B_i \models W^g)]{ag_i[ag_i^{stage} \in \{Set\_*\} / ag_i^{stage} \in \{Tf\_*\}]} \langle ag_i, \alpha, \beta, Tf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Set_Idle**. If the agent has no current thoughtframes or workframes with their preconditions met then place the agent in an idle state. Additionally the agent can not have an active thoughtframe but can possibly have an active workframe.

<u>Rule:</u> Set_Idle

$$\langle ag_i, \alpha, \beta, Set\_*, B_i, F, T_i, TF_i, WF_i \rangle$$
$$\xrightarrow[\alpha \in \{\emptyset\} \wedge \beta \in \{\emptyset\} \wedge \neg \exists \mathcal{T} \in TF_i | B_i \models \mathcal{T}^g \wedge \neg \exists W \in WF_i | B_i \models W^g]{ag_i[ag_i^{stage} \in \{Set\_*\} / ag_i^{stage} \in \{idle\}]}$$
$$\langle ag_i, \alpha, \beta, idle, B_i, F, T_i, TF_i, WF_i \rangle$$

### 6.3.4 Tf_* rules (Thoughtframes)

The agent is now in a state where it is selecting a thoughtframe to run. The agent will not have any thoughtframes currently active. When selecting the thoughtframe to run it will choose the thoughtframe with the highest priority, but if there is more than one then a random selection will be made.

**Tf_Select**. If there is a thoughtframe(s) with preconditions met then perform a selection based on the thoughtframe's priority. The agent can not have a current thoughtframe but can possibly have an active workframe. The thoughtframe is selected using the *Max_pri* method which choses the thoughtframe based on the priority. The agent is then passed onto rules to execute the thoughtframe, the chosen rule depends on the repeat variable of the thoughtframe(true, false or once).

<u>Rule:</u> Tf_Select

$$\langle ag_i, \alpha, \beta, Tf\_*, B_i, F, T_i, TF_i, WF_i \rangle$$
$$\xrightarrow[\alpha \in \{\emptyset\} \wedge \exists \mathcal{T} \in TF_i | B_i \models \mathcal{T}^g]{\alpha' = \alpha[\alpha / Max\_Pri(\forall \mathcal{T} \in TF_i | B_i \models \mathcal{T}^g)] \wedge ag_i[ag_i^{stage} \in \{Set\_*\} / ag_i^{stage} \in \{Tf\_true, Tf\_false, Tf\_once\}]}$$
$$\langle ag_i, \alpha', \beta, \{Tf\_true, Tf\_false, Tf\_once\}, B_i, F, T_i, TF_i, WF_i \rangle$$

**Tf_true (Repeat = true)**. If the repeat variable on the thoughtframe is true then the agent is just directed to 'Pop_Tf*' rules.

<u>Rule:</u> Tf_true

$$\langle ag_i, \alpha, \beta, Tf\_true, B_i, F, T_i, WF_i, TF_i \rangle$$
$$\xrightarrow[\alpha^{r=true} \wedge \beta \in \{\emptyset\}]{ag_i[ag_i^{stage} \in \{Tf\_true\} / ag_i^{stage} \in \{Pop\_Tf*\}]}$$
$$\langle ag_i, \alpha, \beta, Pop\_Tf*, B_i, F, T_i, TF_i, WF_i \rangle$$

**Tf_once (Repeat = once)**. If repeat variable is set to once, change to false then move to 'Pop_Tf*' rules.

<u>Rule:</u> Tf_once

$$\langle ag_i, \alpha, \beta, Tf\_once, B_i, F, T_i, TF_i, WF_i \rangle$$
$$\xrightarrow[\alpha^{r=once} \wedge \beta \in \{\emptyset\}]{\alpha' = \alpha[\alpha^{r=once} / \alpha^{r=false}] \wedge TF'_i = TF_i[\alpha / \alpha'] \wedge ag_i[ag_i^{stage} \in \{Tf\_once\} / ag_i^{stage} \in \{Pop\_Tf*\}]}$$
$$\langle ag_i, \alpha, \beta, Pop\_Tf*, B_i, F, T_i, TF_i, WF_i \rangle$$

**Tf_false(Repeat = false)**. If repeat variable is set to false, then delete thoughtframe from the set of thoughtframes.

RULE: Tf_false

$$\langle ag_i, \alpha, \beta, \mathit{Tf\_false}, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\alpha^{r=false} \wedge \beta \in \{\emptyset\}]{TF'_i = TF_i[TF_i - \alpha] \wedge ag_i[ag_i^{stage} \in \{\mathit{Tf\_false}\} / ag_i^{stage} \in \{\mathit{Pop\_Tf}*\}]}$$
$$\langle ag_i, \alpha, \beta, \mathit{Pop\_Tf}*, B_i, F, T_i, TF'_i, WF_i\rangle$$

**Tf_exit**. If there are no thoughtframes to be executed then the agent is directed towards checking all the detectables.

RULE: Tf_exit

$$\langle ag_i, \alpha, \beta, Tf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\neg\exists \mathcal{T} \in TF_i | B \models \mathcal{T}^g \wedge \alpha \in \{\emptyset\}]{ag_i[ag_i^{stage} \in \{\mathit{Tf\_*}\} / ag_i^{stage} \in \{\mathit{Det\_*}\}]}$$
$$\langle ag_i, \alpha, \beta, \mathit{Det\_*}, B_i, F, T_i\, TF_i, WF_i\rangle$$

### 6.3.5 Wf_* rules (Workframes)

The agent is now in a state where it is selecting a workframe to run. When selecting the workframe to run it will choose the workframe with the highest priority, if there is more than one workframe with the highest priority then a random selection is made between these workframes.

**Wf_select**. If there is no current workframe then a simple selection process occurs taking the workframe with the highest priority. The agent must have no workframes or thoughtframes assigned to it.

RULE: Wf_Select

$$\langle ag_i, \alpha, \beta, \mathit{Wf\_*}, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\alpha \in \{\emptyset\} \wedge \beta \in \{\emptyset\} \wedge \exists W \in WF_i | B_i \models W^g]{\beta' = \beta[\beta / \mathit{Max\_Pri}(\forall W \in WF_i | B_i \models W^g)] \wedge ag_i[ag_i^{stage} \in \{\mathit{Set\_*}\} / ag_i^{stage} \in \{\mathit{Wf\_true}, \mathit{Wf\_false}, \mathit{Wf\_once}\}]}$$
$$\langle ag_i, \alpha, \beta', \{\mathit{Wf\_true}, \mathit{Wf\_false}, \mathit{Wf\_once}\}, B_i, F, T_i, TF_i, WF_i\rangle$$

**Wf_suspend**. If an agent is currently working on a workframe, but there exists a workframe with its guard condition met that has higher priority then the current workframe is suspended and the progress the agent has made through this workframe is recorded. The priority of the suspended workframe is increased by 0.2, priorities are usually integers but this gives suspended workframes higher priority over those which normally would have the same priority. Note. $\beta_d$ represents the workframe's deed stack and $\beta_{ins}$ refers to the workframe's instructions, such as the workframe's repeat values, etc.

RULE: Wf_Suspend

$$\langle ag_i, \alpha, \beta, Wf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\alpha \in \{\emptyset\} \wedge \beta \notin \{\emptyset\} \wedge \exists W \in WF_i | B_i \models W^g \wedge W^{pri} > (\beta^{pri} + 0.3)]{\beta' = \beta[\beta^{pri} / (\beta^{pri} + 0.2)] \wedge WF'_i = WF'_i[WF_i \cup \beta'] \wedge \beta'' \in \{\emptyset\}}$$

$$\langle ag_i, \alpha, \beta'', \mathit{Wf\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_{i'}\rangle$$

**Wf_true (Repeat = true)**. If there does not exist such a workframe with a greater priority then execute the currently selected workframe. 0.3 is added to the current workframes priority when checking whether to suspend, so that the current workframe is not suspended for another suspended workframe of priority only 0.2 higher. The agent is then passed onto rules for processing variables, rules with prefix 'Var_*'

<ins>Rule:</ins> Wf_true

$$\langle ag_i, \alpha, \beta, \mathit{Tf\_true}, B_i, F, T_i, \mathit{WF}_i, \mathit{TF}_i\rangle$$
$$\xrightarrow[\beta^{r=true}\wedge\alpha\in\{\emptyset\}\wedge\neg\exists W\in WF_i|B_i\models W^g\wedge W^{pri}>\beta^{pri+0.3})]{ag_i[ag_i^{stage}\in\{\mathit{Wf\_true}\}/ag_i^{stage}\in\{Pop\_Wf*\}]}$$
$$\langle ag_i, \alpha, \beta, Pop\_Wf*, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$

**Wf_once (Repeat = once)**. If the current workframe has the repeat value once then the repeat value of this workframe is changed to false and the agent is passed onto rules for processing variables.

<ins>Rule:</ins> Wf_once

$$\langle ag_i, \alpha, \beta, Wf\_once, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$
$$\xrightarrow[\neg\exists W\in WF_i|B_i\models W^g\wedge W^{pri}>\beta^{pri+0.3}]{\beta^{r=once}\wedge\alpha\in\{\emptyset\}\wedge\beta'=\beta[\beta^{r=once}/(\beta^{r=false}]\wedge WF'_i=WF_i[\beta/\beta']\wedge ag_i[ag_i^{stage}\in\{\mathit{Wf\_once}\}/ag_i^{stage}\in\{Var\_*\}]}$$
$$\langle ag_i, \alpha, \beta, \mathit{Var\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF'}_i\rangle$$

**Wf_false(Repeat = false)**. If the current workframe has the repeat value false then it is deleted from the set of workframes and the agent is passed onto processing variables.

<ins>Rule:</ins> Wf_false

$$\langle ag_i, \alpha, \beta, \mathit{Wf\_(false)}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$
$$\xrightarrow[\beta^{r=false}\wedge\neg\exists W\in WF_i|B_i\models W^g\&W^{pri}>\beta^{pri+0.3}]{WF'_i=WF_i[WF_i-\beta]\wedge ag_i[ag_i^{stage}\in\{\mathit{Wf\_false}\}/ag_i^{stage}\in\{Var\_*\}]}$$
$$\langle ag_i, \alpha, \beta, \mathit{Var\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF'}_i\rangle$$

### 6.3.6 Det_* rules (Detectables)

Detectables are additional guards contained within a workframe which when activated (though facts not beliefs) will trigger a belief update from the facts and will then decide how the rest of the workframe will be executed. The possible executions are Continue, Complete, Impasse and Abort.

**Det_cont**. When a detectable's guard condition is met and the detectable is of type Continue then the workframe updates its beliefs from the facts detected and carries on unchanged.

<ins>Rule:</ins> Det_cont

$$\langle ag_i, \alpha, \beta, \mathit{Det\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$

$$\xrightarrow[\exists d\in\beta^D|d^g\models F\wedge d^{type=continue}\wedge(\neg\exists d' in\beta^D|d'^g\models F\wedge(d'^{type=impasse}\vee d'^{type=abort}\vee d'^{type=complete}))]{B'_i=B_i\cup d^g\wedge ag_i[ag_i^{stage}\in\{Det\_*\}/ag_i^{stage}\in\{Wf\_*\}]}$$

$$\langle ag_i, \alpha, \beta, Wf\_*, B'_i, F, T_i, TF_i, WF_i\rangle$$

Here $d$ is used to represent a detectable, $\beta^D$ is the workframe $\beta$'s set of detectables. Notation to express parts of the detectables: $d^g$ represents the detectables guard condition and $d^{type}$ refers to the detectables type whether it is continue, complete or abort.

**Det_comp**. When a detectable's guard condition is met and the detectable is of type *complete* then the workframe updates its beliefs from the facts detected and deletes all activities from the workframe leaving only concludes.

RULE: Det_comp

$$\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$$\xrightarrow[\exists d\in\beta^D|d^g\models F\wedge d^{type=complete}\wedge(\neg\exists d' in\beta^D|d'^g\models F\wedge(d'^{type=impasse}\vee d'^{type=abort}))]{\beta'=\beta[\beta_{ins}/\beta^{Concludes}]\wedge B'_i=B_i\cup d^g\wedge ag_i[ag_i^{stage}\in\{Det\_*\}/ag_i^{stage}\in\{Wf\_*\}]}$$

$$\langle ag_i, \alpha, \beta', Wf\_*, B'_i, F, T_i, TF_i, WF_i\rangle$$

$\beta^{Concludes}$ is used to refer to conclude events within the workframe $\beta$.

**Det_impasse**. When the detectable is of type *impasse* the beliefs are updated from the facts detected but the workframe is suspended. To suspend the workframe a new workframe is created out of this workframe instance and added to the set of workframes with repeat set to false. The priority of this new workframe is fractionally larger than the previous (but smaller than a suspended).

RULE: Det_impasse

$$\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$$\xrightarrow[\exists d\in\beta^D|d^g\models F\wedge d^{type=impasse}\wedge(\neg\exists d' in\beta^D|d'^g\models F\wedge d'^{type=abort})]{\beta'=\beta[\beta^{pri}/(\beta^{pri}+0.1)\wedge\beta^g\cup\neg d^g)]\wedge B'_i=B_i\cup d^g\wedge WF'_i=WF_i\cup\beta'\wedge ag_i[ag_i^{stage}\in\{Det\_*\}/ag_i^{stage}\in\{Wf\_*\}]}$$

$$\langle ag_i, \alpha, \beta', Wf\_*, B'_i, F, T_i, TF_i, WF_i\rangle$$

**Det_abort**. If the detectable is of type *abort* then the belief base is updated and the agent's assignment to the workframe is removed.

RULE: Det_abort

$$\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$$\xrightarrow[\exists d\in\beta^D|d^g\models F\wedge d^{type=abort}]{\beta'\in\{\emptyset\}\wedge B'_i=B_i\cup d^g\wedge ag_i[ag_i^{stage}\in\{Det\_*\}/ag_i^{stage}\in\{Wf\_*\}]}$$

$$\langle ag_i, \alpha, \beta', Wf\_*, B'_i, F, T_i, TF_i, WF_i\rangle$$

**Det_empty**. If there are no active detectables found then the agent is moved to the 'workframes' rule set denoted 'Wf_*'.

RULE: Det_empty

$$\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$$\xrightarrow[\neg\exists d\in\beta^{D}|d^{g}\models F]{ag_i[ag_i^{stage}\in\{Det\_*\}/ag_i^{stage}\in\{Wf\_*\}]}$$
$$\langle ag_i, \alpha, \beta, Wf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

### 6.3.7 Var_* rules (Variables)

Variables are used to represent quantification in Brahms. Variables operate on both workframes and thoughtframes, however for simplicity only workframes have been modelled to handle variables. Thoughtframes would operate variables in exactly the same way.

<u>Rule:</u> Var_empty

$$\langle ag_i, \alpha, \beta, Var\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\beta\notin\{\emptyset\}\wedge\beta^{V}\in\{\emptyset\}]{ag_i[ag_i^{stage}\in\{Var\_*\}/ag_i^{stage}\in\{Pop\_*\}]}$$
$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

Note. Where $\beta^V$ represents the variables contained within workframe $\beta$.

**Var_set**. Workframes with variables have an additional stack. This additional stack stores instances of the workframe with the differing instantiations that can be created with the variables. If the set of options is empty then a selection process called 'selectVar()' is called. 'selectVar()' will match all agents/objects which match the name and conditions, assign each to an instance of the workframe then places the instances onto the stack. Note. $\langle\beta_d, [\emptyset], [\beta_{ins}]\rangle$ represents a workframe $\beta$ with a deed stack $d$, a set of empty workframe instances and the workframe's set of instructions $\beta_{ins}$

<u>Rule:</u> Var_set

$$\langle ag_i, \alpha, \beta, Var\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\beta=\langle\beta_d,\emptyset,\beta_{ins}\rangle]{\beta'=\langle\beta_d,[\emptyset\cup selectVar()],\beta_{ins}\rangle}$$
$$\langle ag_i, \alpha, \beta', Var\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Var_one**. When the variable is of type 'forone' and a set of workframe instances has been generated then the first workframe instance is selected and set as the current workframe. The subset of variables in the workframe are then deleted. This is how Brahms performs unification.

<u>Rule:</u> Var_one

$$\langle ag_i, \alpha, \beta, Var\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\beta=\langle\beta_d,W_0\ldots W_n,\beta_{ins}\rangle\wedge\exists v\in\beta^{V}|v^{type}=forone]{\beta'=\langle\beta_d,Random(W_0\ldots W_n),\beta_{ins}\rangle\wedge ag_i[ag_i^{stage}\in\{Var\_*\}/ag_i^{stage}\in\{Pop\_*\}]}$$
$$\langle ag_i, \alpha, \beta', Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

'Random' refers to a random selection of one of the instances and '$v^{type}$' represents the variables type (forone, foreach or collectall).

**Var_each**. When the variable is of type 'foreach' and the subset of the workframe is not empty then the instances of the workframes are added to the set of workframes and the first instance is set as the current workframe. The instances are given a slightly increased priority and a repeat value of false so they will never be repeated. This represents Brahms operating on a multitude of tasks sequentially.

RULE: Var_each

$$\langle ag_i, \alpha, \beta, Var\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\beta=\langle\beta_d, W_0 \ldots W_n, \beta_{ins}\rangle \wedge \exists v \in \beta^V | v^{type}=foreach]{WF'_i=WF_i \cup (W_0[W_0^{pri}/(\beta^{pri}+0.1), W_0^r/W_0^r=false] \ldots W_n[W_n^{pri}/(\beta^{pri}+0.1), W_n^r/W_n^r=false])}$$
$$\langle ag_i, \alpha, W_0, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Var_all**. The 'collectall' variable operates in a similar fashion to the previous variables, however when it selects the first workframe from the subset it merges all the concludes from the other work frames into this workframe. This effectively is how Brahms handles a job which has multiple consequences, e.g., By completing task A, I also complete task B.

RULE: Var_all

$$\langle ag_i, \alpha, \beta, Var\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\beta=\langle\beta_d, W_0 \ldots W_n, \beta_{ins}\rangle \wedge \exists v \in \beta^V | v^{type}=forall]{\beta'=concludes(W_0 \ldots W_n) \wedge ag_i[ag_i^{stage} \in \{Var\_*\}/ag_i^{stage} \in \{Pop\_*\}]}$$
$$\langle ag_i, \alpha, \beta', Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$concludes(W_1 \ldots W_n)$ is a method which takes all the workframe instances $W_1 \ldots W_n$ and extracts the concludes statements.

### 6.3.8 Pop_* rules (Popstack)

Thoughtframes and workframes all have their own stack of instructions. These rules presented demonstrate how the events are "popped" off these instruction stacks. The events can be *activites* or *concludes*, so these rules show how Brahms treats these different instructions.

**Pop_Wfconc***. When a conclude action is found it is removed from the top of the instruction stack. Concludes can update the *beliefs*, the *facts* or both. Three different rules are used for concludes: one for updating *beliefs*; one for *facts*; and one for both. Brahms additionally has probabilities that beliefs will be updated, these probabilities have not been taken into account in these semantics. **Pop_Wfconc*** is neccessary only for workframes, there is no rule **Pop_Tfconc*** thoughtframes since there are no activities to interupt execution.

RULE: Pop_WfconcB

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[b \in B_i \wedge \beta=\langle\beta_d, conclude(b')^{belief}; \beta_{ins}\rangle]{B'_i=(B_i/b) \cup b'}$$
$$\langle ag_i, \alpha, \beta, Pop\_*, B'_i, F, T_i, TF_i, WF_i\rangle$$

The "belief" superscript on the conclude is to show the conclude is for updating beliefs only. The statement $conclude(b')$ represents a conclude statement a belief update $b'$

and $B_i' = (B_i/b) \cup b'$ represents removing the old belief where $b$ and replacing it with the new belief $b'$.

<u>RULE:</u> Pop_WfconcF

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[f \in F \wedge \beta = \langle \beta_d, conclude(f')^{fact}; \beta_{ins}\rangle]{F=(F/f)\cup f'} \langle ag_i, \alpha, \beta, Pop\_*, B_i, F', T_i, TF_i, WF_i\rangle$$

<u>RULE:</u> Pop_WfconcBF

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[b \in B_i \wedge b \in F \wedge \beta = \langle \beta_d, conclude(b')^{belief \wedge fact}; \beta_{ins}\rangle]{F'=(F/b)\cup b' \wedge B_i'=(B_i/b)\cup b'} \langle ag_i, \alpha, \beta, Pop\_*, B_i', F', T_i, TF_i, WF_i\rangle$$

**Pop_concWf***. When agents have finished performing an activity they need to finalise belief updates before they can flag themselves as finished for the cycle. This rule here is for doing exactly this, if a conclude is the next event it will carry out the belief/fact update. Here only 'Pop_concWfB' is described, this shows how it is done with just belief updates. Fact and belief/fact updates will be as previously shown.

<u>RULE:</u> Pop_concWfB

$$\langle ag_i, \emptyset, \beta, Pop\_concWf*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[b \in B_i \wedge \beta = \langle \beta_d, conclude(b')^{belief}; \beta_{ins}\rangle]{B_i'=(B_i/b)\cup b'} \langle ag_i, \alpha, \beta, Pop\_concWf*, B_i', F, T_i, TF_i, WF_i\rangle$$

**Pop_notConc**. This rule is for when the agent is finalising beliefs after an activity but has not found a *conclude* event, the event could be an *activity* or simply empty.

<u>RULE:</u> Pop_notConc

$$\langle ag_i, \alpha, \beta, Pop\_concWf*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\beta = \langle \beta_d, (Prim\_Act \vee Move \vee Comms); \beta_{ins}\rangle]{ag_i[ag_i^{stage} \in \{Pop\_concWf*\}/ag_i^{stage} \in \{fin\}]} \langle ag_i, \alpha, \beta, fin, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_PA***. When a primitive activity is started the agents send the duration of their current activity to the scheduler. The scheduler receives all the activity times then determines which activity time is the smallest and updates its own clock based on this duration. When an agent's time is different to the system clock's it then changes accordingly and subtracts the time increment from the duration of its activity.

**Pop_PASend**. This is the rule the agent's use to send the duration of their next event to the scheduler.

<u>RULE:</u> Pop_PASend

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[T_\xi=T_i\wedge\beta=\langle\beta_d, Prim\_Act^t;\beta_{ins}\rangle]{B_\xi=B_\xi\cup(T_i=T_i+t)} \langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_PA(t>0)**. This rule is invoked when the agent's time is no longer the same as the schedulers time. Additionally this rule checks whether the current activity's duration will be greater than zero after updating the times and durations.

<u>RULE:</u> Pop_PA(t>0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[T_\xi!=Ti\wedge(T_i+t-T_\xi)>0\wedge\beta=\langle\beta_d, Prim\_Act^t;\beta_{ins}\rangle]{t'=(T_\xi-T_i)\wedge T_i=T_\xi\wedge Prim\_Act^t=Prim\_Act^t[t/t']\wedge ag_i[ag_i^{stage}\in\{Pop\_concWf*\}/ag_i^{stage}\in\{fin\}]} \langle ag_i, \alpha, \beta, fin, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_PA(t=0)**. This rule is for when the agent's activity is due to finish at the end of the next clock tick. This rule directs the agent to only executing conclude statements before finishing for the cycle.

<u>RULE:</u> Pop_PA(t=0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[T_\xi!=Ti\wedge(T_i+t-T_\xi)=0\wedge\beta=\langle\beta_d, Prim\_Act^t;\beta_{ins}\rangle]{T_i=T_\xi\wedge\beta=\langle\beta_d,\beta_{ins}-Prim\_Act^t\rangle} \langle ag_i, \alpha, \beta, Pop\_concWF*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_move***. Move activities are very similar to primitive activities, except when the activity terminates a belief update is performed to change the agents and the environments beliefs of the agent's current location. This belief update occurs when the agent notices that the duration of the move has reached zero after the clock update.
**Pop_moveSend**. This is the rule the agent's use to send the duration of their next event to the scheduler.

<u>RULE:</u> Pop_moveSend

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[T_\xi=T_i\wedge\beta=\langle\beta_d, move(Loc=new)^t;\beta_{ins}\rangle]{B_\xi=B_\xi\cup(T_i=T_i+t)} \langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

Note. 'Loc = new' refers to the allocation of the *location* to the *new location.*

**Pop_move(t>0)**. Like for primitive activities, the move activity needs a rule for when the activity still has time remaining after the clock tick.

<u>RULE:</u> Pop_move(t>0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$$\xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) > 0 \wedge \beta = \langle \beta_d, move(Loc=new)^t; \beta_{ins} \rangle]{t' = (T_\xi - T_i) \wedge T_i = T_\xi \wedge move(Loc=new)^t = move(Loc=new)^t[t/t'] \wedge ag_i[ag_i^{stage} \in \{Pop\_concWf*\}/ag_i^{stage} \in \{fin\}]}$$

$$\langle ag_i, \alpha, \beta, fin, B_i, F, T_i, TF_i, WF_i \rangle$$

**Pop_move(t=0)**. Likewise, the move activity needs a rule for when the activity duration ends.

<u>Rule:</u> Pop_move(t=0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) = 0 \wedge \beta = \langle \beta_d, move(Loc=new)^t; \beta_{ins} \rangle]{T_i = T_\xi \wedge \beta = \langle \beta_d, \beta_{ins} - move(Loc=new)^t \rangle \wedge B'_i = B_i[Loc=old/Loc=new] \wedge F' = F[Loc=old/Loc=new]}$$

$$\langle ag_i, \alpha, \beta, Pop\_concWF*, B'_i, F', T_i, TF_i, WF_i \rangle$$

Note. 'old' refers to the previous *location* of the agent.

**Pop_comm***. Communication is very similar to a move activity, except the agent doesn't update its own beliefs or the environments beliefs but it updates another agents beliefs.
**Pop_commSend**. Sends the scheduler the time of next event when processing a communication.

<u>Rule:</u> Pop_commSend

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[T_\xi = T_i \wedge \beta = \langle \beta_d, Comms(ag_j, b')^t; \beta_{ins} \rangle]{B_\xi = B_\xi \cup (T_i = T_i + t)}$$

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

Note. $Comms(ag_j, b')$ represents a communication to agent $j$, sending the belief $b'$.

**Pop_comm(t>0)**. For when the communication has time remaining after the system clock tick.

<u>Rule:</u> Pop_comm(t>0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) > 0 \wedge \beta = \langle \beta_d, Comms(ag_j, b')^t; \beta_{ins} \rangle]{t' = (T_\xi - T_i) \wedge T_i = T_\xi \wedge Comms(ag_j, b')^t = Comms(ag_j, b')^t[t/t'] \wedge ag_i[ag_i^{stage} \in \{Pop\_concWf*\}/ag_i^{stage} \in \{fin\}]}$$

$$\langle ag_i, \alpha, \beta, fin, B_i, F, T_i, TF_i, WF_i \rangle$$

**Pop_comm(t=0)**. Rule for when the communication activity duration ends.

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) = 0 \wedge \beta = \langle \beta_d, Comms(ag_j, b')^t; \beta_{ins} \rangle \wedge b \in B_j]{T_i = T_\xi \wedge \beta = \langle \beta_d, \beta_{ins} - Comms(ag_j, b')^t \rangle \wedge B'_j = B_j[b/b']}$$

$$\langle ag_i, \alpha, \beta, Pop\_concWF*, B_i, F, T_i, TF_i, WF_i \rangle$$

Note. Belief exchange via communication is handed directly in Brahms, i.e. when an agent communicates with another, it directly changes the other agent's beliefs.

**Pop_emptyTf**. Concludes do not use up any simulation time during execution, since thoughtframes only contain concludes then an agent will keep executing thoughtframes until it no longer has any to execute. This rule is for selecting a new thoughtframe when the current one becomes empty.

RULE: Pop_emptyTf

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\alpha=\langle\alpha_d,\emptyset\rangle]{\alpha\in\{\emptyset\}\wedge ag_i[ag_i^{stage}\in\{Pop\_*\}/ag_i^{stage}\in\{Tf\_*\}]} \langle ag_i, \alpha, \beta, Tf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_emptyWf**. A workframe which only contains concludes will act like a thoughtframe. This rule is for such workframes so the agent can keep select another workframes when the current one becomes empty.

RULE: Pop_emptyWf

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\beta=\langle\beta,\emptyset\rangle]{\beta\in\{\emptyset\}\wedge ag_i[ag_i^{stage}\in\{Pop\_*\}/ag_i^{stage}\in\{Wf\_*\}]} \langle ag_i, \alpha, \beta, Wf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

## 6.4 Justification of the Semantics

Justifying a formal semantics is never an easy task, especially when there is nothing formal to compare it against. The only formal basis of the Brahms framework is the implementation code, which for confidentiality reasons we had no access to. The formal semantics was produced by testing Brahms functions and analysing the output to identify how they affect simulations. The problem with this method is that some Brahms functions are not apparent when using or examining the system. For this reason we collaborated with Maarten Sierhuis, who designed Brahms, to develop the semantics. Any correctness issues were discussed with Maarten Sierhuis, to maintain as close a likeness to the design of Brahms as possible. When we had the final draft of the semantics we decided to involve NASA engineers, who use Brahms regularly, to confirm their correctness. The NASA engineers confirmed the correctness of the semantics and created their own implementation of the semantics for their own verification purposes, see the work by Neha Rungta in Chapter 12 which we collaborated with. Overall the semantics have been seen and closely examined by those who know the tool the best and know what it was designed to do. They have studied the rules and also implemented them for their own verification purposes, thus confirming that they accurately represent Brahms and its intended purpose.

# Chapter 7

# Translation to *PROMELA*

In this chapter we describe how we implement operational semantic rules for Brahms in *PROMELA*. The aim of this section is to informally justify that the *PROMELA* code generated by our tool accurately represents the Brahms simulations it was generated from. To help the reader understand how the *PROMELA* matches the semantic rules we again represent the semantic rules, as they are in Chapter 6, for a comparison.

## 7.1 Parsing into Java Data Structures

Before implementing the operational semantics of Brahms in *PROMELA* an intermediate representation of the Brahms data structures was implemented in Java. This intermediate representation stores all the information for the initialisation of a Brahms scenario, such as the workframes, attributes, beliefs, etc. This intermediate representation follows the structure of the tuples found in the Brahms semantics such as the set of workframes, set of thoughtframes, set of agents, etc. This was created using a language recognition tool called ANTLR (Another Tool for Language Recognition) [64]. With this tool a parser was created to read Brahms code and export the data from the simulation into the Java data structures. Figure 7.1 describes our process for verifying Brahms scenarios and highlights the potential for flexible translation to many different verification systems.

The system's semantic structure takes the form $\langle Ags, ag_i, B_\xi, F, T_\xi \rangle$. The system's tuples are mainly represented in a Java class called `MultiAgentSystem.java`. In this class there are Java Set data structures used to store all the agents ($Ags$ in the semantic's tuple) and objects (named 'agents' and 'objects'). These sets store instances of Java classes called `agent.java` for agents and `object.java` for objects. There are also Java Sets for the groups and classes of agents and objects (for hierarchical storage of agents and objects) which provide a reference to which group or class the agents or objects belong. The belief base of the system $B_\xi$ is not represented using the Java data structures as this mainly refers to the agent's simulation time and the duration of their current activities which is represented in the *PROMELA* translation. $F$ refers to the facts of the system which are initially stored in Java Sets inside the classes `agent.java` and `object.java`, since in Brahms facts are asserted into the system by agents during initialisation and throughout the simulation. Once all agents and objects have been initialised the facts from all the agents and objects are collated into a single Java Set inside `MulitAgentSystem.java` called 'facts'. The system's simulation time is represented in the system's tuple by $T_\xi$, simulation time is always zero at initialisation so

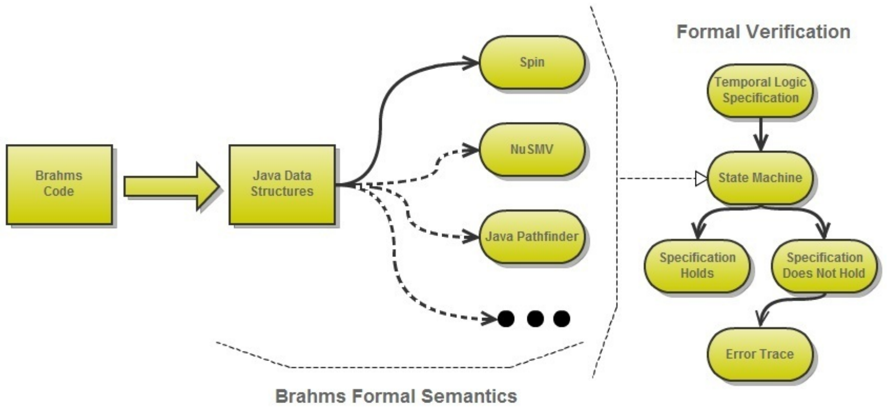


Figure 7.1: The translation and verification process

does not need to be represented in the Java data structures.

The tuple for the agents is $\langle ag_i, \mathcal{T}, W, stage, B, F, T_i, TF, WF \rangle$. The current agent under consideration $ag_i$ does not need to be represented in the Java data structures because the *PROMELA* translation has instantiations of the semantics for each agent. The current workframe and thoughtframe $\mathcal{T}$ and $W$ are only important during a simulation run, i.e., on initialisation they are always empty. The same applies for *stage* which refers to the semantic rule currently under consideration by the agent; *stage* is represented in *PROMELA* by the current position in the programming code. $B$ and $F$ refer to the beliefs and facts of the agent, and are stored in Java Sets inside the Java classes `agent.java` and `object.java`. As described above the simulation time of the agent, $T$, is always zero on initialisation so again is only represented in the *PROMELA* translation.

## 7.2 *PROMELA* Translation

Although *PROMELA* is an appropriate input language for model checking its restrictive data types and control structures made it difficult to write generic code that will apply to *any* model. As such we choose to generate an individual instantiation of the semantic rules for every agent. Due to the restrictions on control structures it was difficult to directly implement the rules, i.e., having a method for each rule and if-statements deciding which method to call. For this reason the semantics were implemented using nested do-while loops, where loops would represent semantic rules. However, due to the nature of this method it means some loops span across more than one rule and some rules span across more than one loop. We refer to this as a partial instantiation of the operational rules, which have been tailored for a particular model of interest. This partial instantiation is generated automatically from the Java representation.

The analysis of this implementation consists of an informal comparison of the *PROMELA* data structures against the complex data structures of the semantics and an informal analysis of the operational rules against the partial instantiations. The "Home Care" system, shown in Part III and Section 8, has been used as a specific example to help illustrate the implementation. It should be noted that we do not provide a fully formal proof that the operational semantics accurately capture the Brahms simulator. So both systems can be viewed separately as mechanisms for exploring models of human-agent teamwork even though they have not yet been proved equivalent. While we believe the translation faithfully captures the formal semantics, a formal proof of all these aspects would take considerable time and is beyond the scope of this thesis.

This section will informally discuss how each semantic rule from Section 6 is represented in *PROMELA*, how it differs yet represents the same actions being performed. The semantic rules will be presented again in this section to remind the reader of the rule being referred to.

### 7.2.1 Representing the Scheduler in *PROMELA*

In Brahms the scheduler is used as a global arbiter, informing the agents when to execute and for how many time steps to execute for. When implementing the scheduler we generate partial instantiations of the scheduler rules which act, not on a list of unknown agents, $Ags$, but upon the specific agents we know to exist in a given specific model. These are different from those found in the operational semantics, because all the data structures have been replaced with instantiations of the agents and objects, etc. The only variables used by the scheduler are:

- integer '`cntEnvironment`', to represent the current time
- enumeration '`turn`', which can be either an object/agent's name or the Environment
- Boolean, '`EnvironmentActive`', decides when the system is to terminate.

All these variables are globally visible so that communication or message passing via channels is not necessary. The *PROMELA* translation simulates the Brahms system scheduler by representing it as a `proctype` named '`proc_Environment`'. The global clock is represented by an integer. Agents are also represented using `proctypes` and the scheduler determines the order of execution through the variable '`turn`' by assigning '`turn`' to the name of an agent who then has exclusive execution rights. Once an agent has executed, '`turn`' is re-assigned to the scheduler and the agent sets the value of its '`timeRemaining`' (e.g.,'`Robot_timeRemaining`') variable to the remaining seconds of its current activity. Once all agents have executed, the scheduler sets the global clock to the current time + the smallest duration of the agents' activities. On the next cycle the agents deduct the time difference between their personal clocks and the global clock from their current activity, and then synchronise their clock to the global clock. The *PROMELA* translation differs from the semantics in that the agents are executed in a prescribed order; however this order is secondary to the overall synchronisation provided by the scheduler.

**Matching the Scheduler's Rules.** The *PROMELA* code captures all the scheduler rules in a loop containing a conditional expression with one condition representing the guard for each rule. If the relevant condition evaluates to true then code representing the rule is executed. We now describe the changes in the *PROMELA* data structures to represent how the *PROMELA* translation alters its data structures with respect to the operational semantics. See [79] or Section 6 for notation and the full operational semantics.

The scheduler has three rules; $\mathsf{Sch\_run}$, $\mathsf{Sch\_Term}$ and $\mathsf{Sch\_rcvd}$, determining the instructions sent to the agents.

Rule: Sch_run

$$\langle Ags, ag_i, B_\xi, F, T_\xi\rangle$$

$$\xrightarrow[\forall ag_i \in Ags | ag_i^{stage} \in \{fin, idle\} \wedge (T_\xi \neq -1)]{ag_{i'} = ag_i[ag_i^{stage} \in \{fin, idle\} / ag_i^{stage} \in \{Set\_Act\}]}$$

$$\langle Ags, ag_{i'}, B_\xi, F, T_\xi\rangle$$

Rule: Sch_Term

$$\langle Ags, ag_i, B_\xi, F, T_\xi\rangle$$

$$\xrightarrow[\forall ag_i \in Ags | stage \in \{idle\}]{T_{\xi'} = T_\xi[T_\xi / T_\xi = -1]}$$

$$\langle Ags, ag_i, B_\xi, F, T_{\xi'}\rangle$$

Rule: Sch_rcvd

$$\langle Ags, ag_i, B_\xi, F, T_\xi\rangle$$

$$\xrightarrow[\forall ag_i \in Ags | stage \in \{Pop\_PA*, Pop\_MA*, Pop\_CA*)\} \vee idle, (T_\xi \neq -1)]{T_{\xi'} = T_\xi[T_\xi / T_\xi + MinTime(\forall ag_i | T_i \in B_\xi)]}$$

$$\langle Ags, ag_i, B_\xi, F, T_{\xi'}\rangle$$

Sch_run becomes active if all the agents are either finished (in the *fin* stage) or idle (the *idle* stage) and the simulation has not yet finished ($T_\xi \neq -1$). In *PROMELA*:

- a set of Boolean variables represent when agents are *idle* (e.g., '`RobotActive`') is set to *false* if the *Robot* is *idle*);
- a set of integers representing the time remaining for each agent's current activity are used to judge whether an agent is in the *fin* stage (e.g., if '`Robot_timeRemaining`' is zero then the *Robot* is in the *fin* stage); and
- *PROMELA* will terminate if the simulation has concluded so it is not necessary to check explicitly for $T_\xi = -1$.

The condition for Sch_run represents the conditions from the rule, i.e., that all the agents must be in a finished or idle state and the time must not be $-1$:

$$\forall ag \in Ags | stage_{ag} \in \{fin, idle\} \wedge (T_\xi \neq -1) \,.$$

An agent is idle when it has no workframes or thoughtframes to execute, this is represented in *PROMELA* using a Boolean with agent's name and a suffix 'Active'. The Boolean with the suffix 'Active' is used to decide whether the scheduler selects the rule Sch_run or Sch_Term. On initialisation this Boolean is set true for all agents. The agents then check all their workframes guard conditions and if no workframes are active the agent will change the Boolean to false. If all agents have their 'Active' Boolean set to false then the rule Sch_Term is executed which terminates all processes. The rule Sch_run is used to tell agents to start processing their current workframes or check for active workframes, etc. The operational semantics suggest that this is a parallel process, i.e., all the agents execute their activities together. If this was implemented in *PROMELA* as a parallel process it would cause unnecessary branching in model, branching to represent every possible interleaving of agent executions. Instead we model this in *PROMELA* as a sequential process by sleeping and waking agents. *PROMELA* does not have the option to wake or sleep processes so an enumeration called 'turn' is used, when an agent is slept turn is changed to 'Environment' to hand control to the scheduler and when the scheduler wakes an agent it does so by assigning 'turn' to the name of the agent. Guard conditions in if-statements using the enumeration 'turn' are used to imitate the agents sleeping, in *PROMELA* an if-statement with no conditions satisfied will halt the process halts until one evaluates true. The rule Sch_rcvd is used to move the simulation clock forward. When all agents have identified the duration their activities they notify the scheduler who takes the smallest of these values and moves the clock forward by this value. In *PROMELA* each agent has a 'timeRemaining' variable which it assigns as duration of its current activity. When checking the 'Activity' Boolean for the agents this 'timeRemaining' variable is also checked, if there is an agent whose 'timeRemaining' variable is greater than zero then Sch_rcvd is selected. Sch_rcvd takes precedence over Sch_run and Sch_Term.

The following code demonstrates how these three rules are constructed within a do-while loop:

```
active proctype proc_Environment(){
do
::(/*Agents are active and all ‘timeRemaining’
   variables equal 0, rule Sch_run*/)->
   if
   ::(turn == environment) -> turn = Robot;
   fi;
   if
   ::(turn == environment) -> turn = Clock;
   fi
   ...
::(/*An Agent has ‘timeRemaining’ variable
   greater than zero, rule Sch_rcvd*/)->
   if
   ::(/*Lowest duration = Robot*/)
      cntEnvironment =  cntEnvironment+Robot_timeRemaining;
   ::(/*Lowest = Clock*/)
      ...
   fi
::(/*No active agents, rule Sch_Term*/)
   environmentActive = false;
od}
```

### 7.2.2 From Agent Semantics to *PROMELA* Processes

**Representing the Agent's Data Structures in *PROMELA*.** The *PROMELA* translation has a separate proctype for each agent labelled '`proc_`' followed by the agent's name (e.g.,'`proc_Robot`'). The name of the agent is a member of the scheduler's enumeration data structure '`turn`'. So the agent only executes when this variable holds the agent's name. The components of the 9-tuple that represent an agent are primarily

| Index | Description |
|---|---|
| 0 | Thoughtframe ID number |
| 1 | Boolean guard condition, e.g.,1 = thoughtframe is active |
| 2 | Priority of the thoughtframe |
| 3 | Repeat, e.g.,0 = delete, 3 = always |
| 4 | Last deed on stack |
| . . | . . |
| $i$ | Top deed on stack |

Table 7.1: Current Thoughtframe: *tf_stackRobot*.

| Index | Description |
|---|---|
| 0 | Workframe ID number |
| 1 | Boolean guard condition, e.g.,1 = workframe is active |
| 2 | Priority of the workframe |
| 3 | Repeat, e.g.,0 = delete, 3 = always |
| 4 | Boolean to flag a communication or move activity |
| 5 | Boolean to flag the workframe is in impasse |
| 6 . . | Last deed on stack . . |
| $i$ | Top deed on stack |

Table 7.2: Current Workframe: *wf_stackRobot*.

represented by arrays. These arrays are referred to by name in the partial instantiations of the operational rules. For instance, $\mathcal{T}$, the agent's current thoughtframe is represented as a one-dimensional array and treated as a stack. The array is labelled '`tf_stack`' followed by the agent's name, e.g.,'`tf_stackRobot`' with the corresponding pointer '`tf_RobotTop`' to identify the top element. The first four indices of the array (elements 0-3) are used to store the header data of the thoughtframe (like the array depicted for workframes below without rows labelled 4 and 5). For an example see Table 7.1.

The current workframe is represented in a similar fashion. The first six indices (three in the case of the current thoughtframe) of the array (elements 0-5) are used to store the workframe header data. Below the header information are a stack of *deeds* which may represent belief updates or activities, e.g., see Table 7.2.

We do not represent the current stage of the agent's reasoning cycle explicitly, but do so implicitly by the order in which rules are represented in the *PROMELA* code. Beliefs and facts in Brahms are tied to the attributes and relations of an agent, e.g., agent `Robot` believes Bob's attribute `AskedForFood = true`. To model this in *PROMELA* every agent is assigned a belief about every attribute, even if it does not own that attribute. This belief is represented as a Boolean array either as an integer array or a enumeration array, depending on the attribute. The name of the belief is the name of the agent followed by the name of the attribute, e.g., `RobotAskedForFood` represents the Robot's beliefs about the attribute `AskedForFood`. The Clock object will also have

| | |
|---|---|
| 0 = Robot's ID | Robot believes the Robot askedForFood = false |
| 1 = Clock's ID | Robot believes the Clock askedForFood = false |
| 2 = Bob's ID | Robot believes that Bob AskedForFood = true |
| 3 = House's ID | Robot believes the House AskedForFood = false |

Table 7.3: Beliefs concerning *RobotAskedForFood*.

| | 0<br>0 = Robot's ID | 1<br>1 = Clock's ID | 2<br>2 = Bob's ID | 3<br>3 = House's ID |
|---|---|---|---|---|
| 0 = Robot's ID | 0 | 0 | 1 | 0 |
| 1 = Clock's ID | 0 | 0 | 0 | 0 |
| 2 = Bob's ID | 0 | 0 | 0 | 0 |
| 3 = House's ID | 0 | 0 | 0 | 0 |

Table 7.4: Relation *RobotIsFriendOf*.

`ClockAskedForFood` even though it does not have this attribute. The index of the array is the ID number of the agent whom the belief concerns, e.g., see Table 7.3.

Beliefs about *relationships* take a slightly different form. Relationships involve two agents, e.g., agent Robot '*is a friend of*' agent Bob. Beliefs about relationships are represented by a two-dimensional array where both x-y indices represent agents and the value 1 represents the relationship exists and 0 when it does not. The value is a Boolean on whether the relationship exists. The Table 7.4 represents a relationship belief that the Robot believes the Robot '*is a friend of*' Bob.

**F** in the operational semantics tuple describes the set of *facts*. Facts are identical to beliefs except they represent what the value of the attribute actually is, not what is believed about the attribute. Facts are represented using the same array structures as beliefs except the array's name starts with 'fact' followed by the attribute name e.g., see Table 7.5.

Thoughtframe and workframe sets are represented as two-dimensional arrays where the first index represents the thoughtframe or workframe and the second represents the elements of the thoughtframe or workframe. These are named '`tf`' or '`wf`' followed by the name of the agent. A pointer to the top of the thoughtframe's stack is represented using a one dimensional array called '`tfTop`' followed by the name of the agent. This array stores the pointer for each thoughtframe in the set. The Table 7.6 shows a thoughtframe at a certain index has a depth given by the value in the array, e.g., the thoughtframe at index 0 has depth = i.

The representation of thoughtframes is depicted in the Table 7.7; this is similar to workframes shown in Table 7.8 but without the rows labelled 4 (`Comm/Move`) and 5 (`impasse`).

| | |
|---|---|
| 0 = Robot's ID | Robot's AskedForFood = false |
| 1 = Clock's ID | Clock's AskedForFood = false |
| 2 = Bob's ID | Bob's AskedForFood = true |

Table 7.5: Fact: *factAskedForFood*.

| 0 | 1 | 2 |
|---|---|---|
| Thoughtframe at index 0 has depth = i | Thoughtframe at index 1 has depth k | Thoughtframe at index 2 has depth j |

Table 7.6: Thoughtframe (*tfTopRobot*) Depths

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | ID = 0 | ID = 1 | ID = 2 |
| 1 | Guard = 0 | Guard = 1 | Guard = 0 |
| 2 | Priority = 2 | Priority = 1000 | Priority = 0 |
| 3 | Repeat = 3 | Repeat = 1 | Repeat = 3 |
| 4 | Last Deed | Last Deed | Last Deed |
| . . . | . . . | . . . | . . . |
| *i* | Top Deed | . | . |
| *j* | | . | Top Deed |
| *k* | | Top Deed | |

Table 7.7: Thoughtframes: *tfRobot*.

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | ID = 0 | ID = 1 | ID = 2 |
| 1 | Guard = 1 | Guard = 1 | Guard = 1 |
| 2 | Priority = 1 | Priority = 10 | Priority = 4 |
| 3 | Repeat = 3 | Repeat = 3 | Repeat = 3 |
| 4 | Comm/Move = 0 | Comm/Move = 0 | Comm/Move = 0 |
| 5 | impasse = 0 | impasse = 1 | impasse = 0 |
| 6 | Last Deed | Last Deed | Last Deed |
| . . | . . | . . | . . |
| *i* | . | . | Top Deed |
| *j* | . | Top Deed | |
| *k* | Top Deed | | |

Table 7.8: Workframes: *wfRobot*.

**Additionally.** Agents also have other data structures to identify their current state: an integer to represent their current time; an integer to represent how long they have remaining on their current activity; and a Boolean to state whether or not they are active.

**Matching the Agent's Semantic Rules in *PROMELA*.** When the scheduler's '`turn`' enumeration is an agent name then control passes to the agent rules. Like the scheduler rules these are represented by a loop that checks the rule pre-conditions in turn. To explain how the *PROMELA* translation matches Brahms we show how one of the operational semantic rules is represented in *PROMELA*. A comparison will be made describing how the operational semantic rules are programmed in *PROMELA*.

<u>RULE:</u> Set_Act

$$\langle ag_i, \alpha, \beta, Set\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\alpha\in\{\emptyset\}\wedge(\exists\mathcal{T}\in TF_i|B_i\models\mathcal{T}^g\vee\exists W\in WF_i|B_i\models W^g)]{ag_i[ag_i^{stage}\in\{Set\_*\}/ag_i^{stage}\in\{Tf\_*\}]}$$
$$\langle ag_i, \alpha, \beta, Tf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

<u>RULE:</u> Set_Idle

$$\langle ag_i, \alpha, \beta, Set\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\alpha\in\{\emptyset\}\wedge\beta\in\{\emptyset\}\wedge\neg\exists\mathcal{T}\in TF_i|B_i\models\mathcal{T}^g\wedge\neg\exists W\in WF_i|B_i\models W^g]{ag_i[ag_i^{stage}\in\{Set\_*\}/ag_i^{stage}\in\{idle\}]}$$
$$\langle ag_i, \alpha, \beta, idle, B_i, F, T_i, TF_i, WF_i\rangle$$

Set_Act is a simple rule which determines whether or not the agent is active. The rule states the agents' examine whether or not they have an active workframe or thoughtframe to process, if they do then this rule is activates and puts the agent into a state where it can operate its thoughtframes. If there are no frames active then Set_Idle is activated which effectively puts the agent to sleep for this cycle. In *PROMELA* the agents first of all check which thoughtframes and workframes are active. The guard conditions for each workframe (and thoughtframe) are coded as an if-statement in *PROMELA*. If the guard is satisfied then the ID number of the frame is passed to a macro to set the frame as active e.g.,

```
/* Workframe wf_getFood with ID = 10, has no variables
and agent name is Robot*/
if
::(Robot_askedFood[AlexID] ==true)->
    printf("Workframe wf_getFood is active\n");
    RobotwfActive(10)
::else->
    RobotwfNotActive(10)
fi;
```

$RobotwfNotActive(10)$ and RobotwfActive(10) in the above code represent the macros to declare the workframe with identification number 10 active or inactive. These macros loop through all frames in the set of workframes (or thoughtframes) array, any frame which matches the ID number passed to it has its active flag set to true (or false) in the header data, also the macro keeps a check on which has the highest priority. Once all workframe and thoughtframe guard conditions have been evaluated a further quick check is performed to see if any are active. If no frames are active then the agent's 'timeRemaining' variable is set to -1 to indicate the agent is idle, if there are active frames then the agent starts operating its thoughtframes.

Rule: Tf_Select

$$\langle ag_i, \alpha, \beta, Tf\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[\alpha \in \{\emptyset\} \wedge \exists \mathcal{T} \in TF_i | B_i \models \mathcal{T}^g]{\alpha' = \alpha[\alpha / Max\_Pri(\forall \mathcal{T} \in TF_i | B_i \models \mathcal{T}^g)] \wedge ag_i[ag_i^{stage} \in \{Set\_*\} / ag_i^{stage} \in \{Tf\_true, Tf\_false, Tf\_once\}]}$$

$$\langle ag_i, \alpha', \beta, \{Tf\_true, Tf\_false, Tf\_once\}, B_i, F, T_i, TF_i, WF_i \rangle$$

Tf_Select determines which thoughtframe is to be selected for execution. For the rule to be activated there needs to exist at least one thoughtframe in the set of thoughtframes whose guard conditions evaluates to true with respect to the belief base ($\exists \mathcal{T} \in TF_i | B_i \models T^g$). Rule Set_Act has already determined which (if any) are active. Tf_Select states that the "current thoughtframe" entry in the tuple must be empty, which is represented in *PROMELA* by a pointer to the current workframe's top element. If the value of this pointer is negative then there is no current thoughtframe. If the current thoughtframe is empty and a thoughtframe is active then the Tf_Select procedure will be invoked.

The following code shows how *PROMELA* decides if Wf_Select is active, note that the guards on the do-while statements have been replaced with comments to simplify the code:

```
bool active = false;
do
::(/*workframe is active in array*/)  ->
                  active = true; break;
::(/*workframe is not active*/)  ->  skip;
::(/*End of array*/)             ->  break;
od;
if
::(active==true &&  /*current workframe pointer*/ == -1)->
   /*Select a workframe*/
::else ->   /*Set idle*/
fi;
```

Tf_Select performs a selection process to find the active thoughtframe with the highest priority ($\beta = Max_{pri}(\mathcal{T} \in TF_i | B \models \mathcal{T}^g)$). The *PROMELA* translation loops through the array of thoughtframes, checks the guard condition and the priority of each thoughtframe (index 1 and 2 in the thoughtframe array shown earlier). It builds a temporary array of thoughtframes that share the maximum priority among all the active workframes. Finally the *PROMELA* code arbitrarily selects one thoughtframe from this temporary array. Example code to explain this is as follows:

```
/*Loop through set of thoughtframes*/
do
::(/*thoughtframe is active, priority = highest*/)->
   /*Add to temporary array*/
::else->  skip;
```

```
od;
/*Loop through temporary array*/
do
::(/*Not the last, Reject*/) ->   skip;
::(/*Not the last, Accept*/)  ->
   /*Upload workframes data into current workframe*/
::else->   /*Upload thoughtframes data into current thoughtframe*/
od;
```

The rules (Tf_true, Tf_once, Tf_false) are used to determine whether or not the thoughtframes may be repeated after execution; either once, always or never.

**Tf_true (Repeat = true)**. If the repeat variable on the thoughtframe is true then the agent is just directed to 'Pop_Tf*' rules.

<u>RULE:</u> Tf_true

$$\langle ag_i, \alpha, \beta, \mathit{Tf\_true}, B_i, F, T_i, \mathit{WF}_i, \mathit{TF}_i\rangle$$
$$\xrightarrow[\alpha^{r=true} \wedge \beta \in \{\emptyset\}]{ag_i[ag_i^{stage} \in \{\mathit{Tf\_true}\}/ag_i^{stage} \in \{\mathit{Pop\_Tf}*\}]}$$
$$\langle ag_i, \alpha, \beta, \mathit{Pop\_Tf}*, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$

**Tf_once (Repeat = once)**. If repeat variable is set to once, change to false then move to 'Pop_Tf*' rules.

<u>RULE:</u> Tf_once

$$\langle ag_i, \alpha, \beta, \mathit{Tf\_once}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$
$$\xrightarrow[\alpha^{r=once} \wedge \beta \in \{\emptyset\}]{\alpha' = \alpha[\alpha^{r=once}/\alpha^{r=false}] \wedge \mathit{TF}'_i = \mathit{TF}_i[\alpha/\alpha'] \wedge ag_i[ag_i^{stage} \in \{\mathit{Tf\_once}\}/ag_i^{stage} \in \{\mathit{Pop\_Tf}*\}]}$$
$$\langle ag_i, \alpha, \beta, \mathit{Pop\_Tf}*, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$

where we use the notation $\alpha' = \alpha[\alpha^{r=once}/(\alpha^{r=false}]$ to indicate that the repeat value of $\alpha$ has been replaced by $\alpha^{r=false}$.

**Tf_false(Repeat = false)**. If repeat variable is set to false, then delete thoughtframe from the set of thoughtframes.

<u>RULE:</u> Tf_false

$$\langle ag_i, \alpha, \beta, \mathit{Tf\_false}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$
$$\xrightarrow[\alpha^{r=false} \wedge \beta \in \{\emptyset\}]{\mathit{TF}'_i = \mathit{TF}_i[\mathit{TF}_i - \alpha] \wedge ag_i[ag_i^{stage} \in \{\mathit{Tf\_false}\}/ag_i^{stage} \in \{\mathit{Pop\_Tf}*\}]}$$
$$\langle ag_i, \alpha, \beta, \mathit{Pop\_Tf}*, B_i, F, T_i, \mathit{TF}'_i, \mathit{WF}_i\rangle$$

In *PROMELA* these semantic rules are performed by checking an if-statement which has a condition suitable for each of the rules above. Each condition checks the repeat variable in the header data of the thoughtframe. The repeat variable can be: 0 - delete, 1 - never repeat, 2 - repeat once and 3 - always repeat. These integers are used so a change in repeat status can be handled by a simple subtraction of the repeat variable e.g., if a workframe is marked as repeat once it is assigned a value of 2, after executing this workframe this value is reduced to 1 (never repeat) and finally to 0 which marks the workframe for deletion. Before every cycle the scheduler checks all the agents' workframes and thoughtframes for any with a repeat variable of 0 (delete) and deletes the frame from the array.

```
/*tfRobot[i].elements[3] represents the repeat variable
in the header data of the thoughtframe at index i*/
if
::(tfRobot[i].elements[3] < 3)->
    tfRobot[i].elements[3] = tfRobot[i].elements[3] - 1;
    printf("Thoughtframes repeat variable is reduced\n");
::else ->
    skip;
    printf("Thoughtframes repeat variable is to always repeat\n");
fi;
```

Tf_exit is the rule which decides if there are no thoughtframes to execute and directs the agent onto examining the detectables of its current workframe.

<u>RULE:</u> Tf_exit

$$\langle ag_i, \alpha, \beta, Tf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$$\xrightarrow[\neg\exists\mathcal{T}\in TF_i|B\models\mathcal{T}^g\wedge\alpha\in\{\emptyset\}]{ag_i[ag_i^{stage}\in\{Tf\_*\}/ag_i^{stage}\in\{Det\_*\}]}$$

$$\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i TF_i, WF_i\rangle$$

In the *PROMELA* translation the guard conditions on the thoughtframes are checked after every execution of a thoughtframe, if a thoughtframe is found then Tf_Select is chosen but if not then Tf_exit is chosen. These two rules are represented using a single if-statement where one condition is Tf_Select and the other is Tf_exit. The two rules are decided by a loop which executes before this if-statement, this loop iterates through all the thoughtframes until it finds one which is active and then breaks. If the counter on this loop reaches the max number of thoughtframes then no active thoughtframes were found and selects the Tf_exit condition which contains a goto workframes command. The workframes section of code firstly examines detectables on the workframes, so correctly moves onto checking detectables.

```
/*tfRobotIndex represents index of the last thoughtframe in the
array and i is the counter from the loop*/
if
::(i <= tfRobotIndex)->
    printf("Selecting a thoughtframe\n");
    ...
:: else ->
    printf("Moving onto Workframes\n");
    goto workframes;
fi;
```

The workframes set of rules operate in the same fashion as the thoughtframes; rules such as Wf_Select, Wf_true, Wf_false, Wf_once. The only difference with thoughtframes and workframes is that workframes don't need a Wf_exit rule and that workframes can suspend using Wf_Suspend.

<u>RULE:</u> Wf_Suspend

$$\langle ag_i, \alpha, \beta, Wf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$$\xrightarrow[\alpha \in \{\emptyset\} \wedge \beta \notin \{\emptyset\} \wedge \exists W \in WF_i | B_i \models W^g \wedge W^{pri} > (\beta^{pri}+0.3)]{\beta' = \beta[\beta^{pri}/(\beta^{pri}+0.2)] \wedge WF'_i = WF'_i[WF_i \cup \beta'] \wedge \beta'' \in \{\emptyset\}}$$

$$\langle ag_i, \alpha, \beta'', Wf\_*, B_i, F, T_i, TF_i, WF_{i'} \rangle$$

Wf_Suspend is selected when the agent has a current workframe but another workframe of higher priority has become active. This rule creates a copy of the current workframe (in its current altered condition) and adds this workframe copy onto the set of workframes. The priority of this workframe copy is +0.2 of the original and the repeat variable is set to never repeat. Wf_Suspend is represented in *PROMELA* with an if-statement before Wf_Select, this if-statement checks to see if the agent has a current workframe; by checking if the pointer pointing to the current element in the workframe array is empty (i.e., equal to -1). Nested inside this if-statement is a loop which loops through all active workframes checking to see if one has a priority higher than the current workframe's + 0.3. If this is the case then the pointer to the last workframe in the set of workframes is increased and the elements of the current workframe are uploaded at this position and the pointer pointing to the current position in the current workframe is reset to -1 to indicate the current workframe is now empty. The following code shows how *PROMELA* implementation checks whether to suspend a workframe:

```
if /*Check if top element of current workframe stack is empty.
     This is checked by making sure the pointer to the top element
     in the stack is not empty*/
::(wf_RobotTop != -1) ->
    if /*Check if priority of highest active workframe is greater
        than the current +3*/
    ::(pri > (wf_stackRobot[2]+3)) ->
        /*increase number of workframes in set*/
        wfRobotIndex = wfRobotIndex +1;
        i = 0;
        /*Cycle through all elements in the current workframe to
        add them to the new temporary workframe in the set of
        all workframes*/
        do
        ::(i <= wf_RobotTop) ->
            if
            ::(i == 2) ->
                /*Check the priority of the workframe and perform a
                check to test whether workframe is already suspended,
                i.e., if the priority is a multiple of 10 then it
                has not already been suspended*/*/
                j = wf_stackRobot[i];
                do
                ::(j > 0) ->
                    j = j - 10;
                ::(j == 0) ->
                    wfRobot[wfRobotIndex].elements[i] =
                        wf_stackRobot[i] + 2;
                    break;
                ::else ->
                    /*Workframe has already been suspended*/
                    break;
                od;
                j = 0;
                i = i+1;
            ::(i == 3) ->
                /*This is the repeat variable, because it is
                suspended the repeat variable must be set to once, i.e.,
                 equal to 1*/
                wfRobot[wfRobotIndex].elements[i] = 1;
                i = i+1;
            ::else ->
                /*Save element from the current workframe
                in the set of all workframes*/
                wfRobot[wfRobotIndex].elements[i] = wf_stackRobot[i];
                i = i+1;
            fi;
        ::else ->
            break;
        od;
        /*Mark the top element in the stack and reset*/
        wfTopRobot[wfRobotIndex] = wf_RobotTop;
        wf_RobotTop = -1;
    ::else ->
        /*No need to suspend the current workframe*/
        skip;
    fi;
    i = 0;
::else ->
    /*There is no current workframe to suspend*/
    skip;
fi;
```

### 7.2.3 Det_* rules (Detectables)

Detectables are additional guards contained within a workframe which when activated (through facts not beliefs) will trigger a belief update from the facts and will then decide how the rest of the workframe will be executed. The possible executions are Continue, Complete, Impasse and Abort.

**Det_cont**. When a detectables guard condition is met and the detectable is of type Continue then the workframe updates its beliefs from the facts detected and carries on unchanged.

Rule: Det_cont

$$\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

$$\xrightarrow[\exists d \in \beta^D | d^g \models F \wedge d^{type=continue} \wedge (\neg \exists d' in \beta^D | d'^g \models F \wedge (d'^{type=impasse} \vee d'^{type=abort} \vee d'^{type=complete}))]{B'_i = B_i \cup d^g \wedge ag_i[ag_i^{stage} \in \{Det\_*\} / ag_i^{stage} \in \{Wf\_*\}]}$$

$$\langle ag_i, \alpha, \beta, Wf\_*, B'_i, F, T_i, TF_i, WF_i\rangle$$

Here $d$ is used to represent a detectable, $\beta^D$ is the workframe $\beta$'s set of detectables. We also use notation here to express parts of the detectables: $d^g$ represents the detectables guard condition and $d^{type}$ refers to the detectables type whether it is continue, complete or abort.

**Det_comp**. When a detectable's guard condition is met and the detectable is of type *complete* then the workframe updates its beliefs from the facts detected and deletes all activities from the workframe leaving only concludes.

<u>RULE:</u> Det_comp

$$\frac{\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\exists d \in \beta^D | d^g \models F \wedge d^{type=complete} \wedge (\neg \exists d' in \beta^D | d'^g \models F \wedge (d'^{type=impasse} \vee d'^{type=abort}))]{\beta'=\beta[\beta_{ins}/\beta^{Concludes}] \wedge B'_i = B_i \cup d^g \wedge ag_i[ag_i^{stage} \in \{Det\_*\}/ag_i^{stage} \in \{Wf\_*\}]}}{\langle ag_i, \alpha, \beta', Wf\_*, B'_i, F, T_i, TF_i, WF_i\rangle}$$

$\beta^{Concludes}$ is used to refer to conclude events within the workframe $\beta$.

**Det_impasse**. When the detectable is of type *impasse* the beliefs are updated from the facts detected but the workframe is suspended. To suspend the workframe a new workframe is created from what remains of the current workframe instance, this is then added to the set of workframes with a repeat value of false. The priority of this new workframe is fractionally larger than the previous (but smaller than a suspended).

<u>RULE:</u> Det_impasse

$$\frac{\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\exists d \in \beta^D | d^g \models F \wedge d^{type=impasse} \wedge (\neg \exists d' in \beta^D | d'^g \models F \wedge d'^{type=abort})]{\beta'=\beta[\beta^{pri}/(\beta^{pri}+0.1) \wedge \beta^g \cup \neg d^g)] \wedge B'_i = B_i \cup d^g \wedge WF'_i = WF_i \cup \beta' \wedge ag_i[ag_i^{stage} \in \{Det\_*\}/ag_i^{stage} \in \{Wf\_*\}]}}{\langle ag_i, \alpha, \beta', Wf\_*, B'_i, F, T_i, TF_i, WF_i\rangle}$$

**Det_abort**. If the detectable is of type *abort* then the belief base is updated and the agent's assignment to the workframe is removed.

<u>RULE:</u> Det_abort

$$\frac{\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\exists d \in \beta^D | d^g \models F \wedge d^{type=abort}]{\beta' \in \{\emptyset\} \wedge B'_i = B_i \cup d^g \wedge ag_i[ag_i^{stage} \in \{Det\_*\}/ag_i^{stage} \in \{Wf\_*\}]}}{\langle ag_i, \alpha, \beta', Wf\_*, B'_i, F, T_i, TF_i, WF_i\rangle}$$

**Det_empty**. If there are no active detectables found then agent is moved to the 'workframes' rule set denoted 'Wf_*'.

<u>RULE:</u> Det_empty

$$\frac{\langle ag_i, \alpha, \beta, Det\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\neg \exists d \in \beta^D | d^g \models F]{ag_i[ag_i^{stage} \in \{Det\_*\}/ag_i^{stage} \in \{Wf\_*\}]}}{\langle ag_i, \alpha, \beta, Wf\_*, B_i, F, T_i, TF_i, WF_i\rangle}$$

Detectables are checked just before a current workframe pops an activity element off the stack. In the *PROMELA* translation a loop with multiple conditions is used to pop elements off the stack and check if its activities or concludes being processed, the basic conditions are: stack is empty, activity found and conclude is found. The activity duration is what decides this, i.e., duration >0 is an activity, duration <0 is a conclude and duration = 0 is empty. If-statements are used to represent the guard conditions of all the detectables. The following code shows how activities, concludes and end of activities are distinguished:

```
/*wf_stackRobot is an array containing all elements of the
current workframe.  wf_RobotTop is the pointer to the
current position in the stack.  This has to be >5 because
0-5 is the header data.  det_complete will be explained
later*/
do
::(wf_stackRobot[wf_RobotTop] == 0 &&
  wf_RobotTop > 5);
   /*Element is 0, activity has finished*/
   ...
::(wf_stackRobot[wf_RobotTop] >= 1 &&
det_complete == false && ... && wf_RobotTop > 5)->
   /*Element is > 0 therefore an activity*/
   /*Check detectables*/
   ...
::(wf_stackRobot[wf_RobotTop] >= 1 &&
det_complete == true && ... && wf_RobotTop > 5)->
   /*Element is > 0 therefore an activity, but det_complete
   is true so activity must be discarded*/
   ...
::((wf_stackRobot[wf_RobotTop] < 0 && ... &&
   wf_RobotTop > 5)->)
   /*Element is < 0, processing a conclude*/
   ...
od
```

The code generated is generic to all workframes so all detectables for every workframe will be checked. To ensure a detectable from another workframe does not fire the if-statement has an additional condition: that the ID number of the current workframe must match the ID of the workframe which has this detectable. If the detectable is active then a belief update on the agent is made matching the agent's belief to the fact, also a variable is set to identify what kind of detectable has fired (abort, continue, etc.) called activeDetectableType which is set to identify which detectable was fired for that workframe (workframes can have more than one detectable). Example code for a workframes detectable:

```
/*fact_hasEmptyPlate[AlexID] == true is the detectables guard.
wf_stackRobot[0] is the current workframe stack and index 0
references the workframe ID number, which in this example is 11*/
```

```
if
::(fact_hasEmptyPlate[AlexID] ==true && wf_stackRobot[0] == 11)->
    Robot_hasEmptyPlate[AlexID]  = fact_hasEmptyPlate[AlexID] ;
    activeDetectableType = abort;
    activeDetectableID = 0;
fi
```

After all the detectables have been checked an if-statement then decides what to do based on the type of detectable which was active. Each of the conditions in this if-statement represent a rule (Det_continue, Det_abort, Det_impasse, Det_complete and Det_empty). The conditions examine the variable activeDetectableType to identify which is to be processed e.g.,

```
 if
 ::(activeDetectableType == impasse)->
     ...
 ::(activeDetectableType == abort)->
     ...
 ::(activeDetectableType == continue)->
     ...
 ::(activeDetectableType == complete)->
     ...
 ::(activeDetectableType == null /*i.e., empty*/)->
     ...
 fi;
```

Impassing a workframe is a similar to suspending one, i.e., adding another workframe to the set with an increased priority, a repeat variable set to false, and resetting the current workframe pointer to -1. A continue variable has no additional consequences, simply continues to pop the stack. Abort deletes the current workframe by setting the pointer to -1. Complete sets a variable called det_complete to true, this variable is used in the if-statement for deciding if the event is an activity, empty or conclude; if this variable is true and an activity is found then the activity is discarded. If the variable is null then it jumps to popping the stack.

### 7.2.4 Pop_* rules (Popstack)

Thoughtframes and workframes all have their own stack of instructions. These rules presented demonstrate how the events are "popped" off these instruction stacks. The events can be activities or concludes, so these rules show how Brahms treats these different instructions.

**Pop_Wfconc***. When a conclude action is found it is removed from the top of the instruction stack. Concludes can update the beliefs, the facts or both. We represent three different rules for concludes: one for updating beliefs; one for facts; and one for both. Brahms additionally has probabilities that beliefs will be updated, these probabilities have not been taken into account in these semantics. With concludes we only describe the rules for workframes because performing concludes in thoughtframes are identical but on a different stack.

RULE: Pop_WfconcB

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[b \in B_i \wedge \beta = \langle \beta_d, conclude(b')^{belief}; \beta_{ins}\rangle]{B'_i = (B_i/b) \cup b'}$$
$$\langle ag_i, \alpha, \beta, Pop\_*, B'_i, F, T_i, TF_i, WF_i\rangle$$

Note. The "belief" superscript is to show the conclude is for updating beliefs only.

<u>Rule:</u> Pop_WfconcF

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[f \in F \wedge \beta = \langle \beta_d, conclude(f')^{fact}; \beta_{ins}\rangle]{F = (F/f) \cup f'}$$
$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F', T_i, TF_i, WF_i\rangle$$

<u>Rule:</u> Pop_WfconcBF

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[b \in B_i \wedge b \in F \wedge \beta = \langle \beta_d, conclude(b')^{belief \wedge fact}; \beta_{ins}\rangle]{F' = (F/b) \cup b' \wedge B'_i = (B_i/b) \cup b'}$$
$$\langle ag_i, \alpha, \beta, Pop\_*, B'_i, F', T_i, TF_i, WF_i\rangle$$

**Pop_concWf***. When agents have finished performing an activity they need to finalise belief updates before they can flag themselves as finished for the cycle. This rule here is for doing exactly this, if a conclude is the next event it will carry out the belief/fact update. Here we describe only 'Pop_concWfB', this shows how it is done with just belief updates. Fact and belief/fact updates will be as previously shown.

<u>Rule:</u> Pop_concWfB

$$\langle ag_i, \emptyset, \beta, Pop\_concWf*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[b \in B_i \wedge \beta = \langle \beta_d, conclude(b')^{belief}; \beta_{ins}\rangle]{B'_i = (B_i/b) \cup b'}$$
$$\langle ag_i, \alpha, \beta, Pop\_concWf*, B'_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_notConc**. This is for when the agent is finalising beliefs after an activity but has not found a *conclude* event, the event could be an *activity* or simply empty.

<u>Rule:</u> Pop_notConc

$$\langle ag_i, \alpha, \beta, Pop\_concWf*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\beta = \langle \beta_d, (Prim\_Act \vee Move \vee Comms); \beta_{ins}\rangle]{ag_i[ag_i^{stage} \in \{Pop\_concWf*\}/ag_i^{stage} \in \{fin\}]}$$
$$\langle ag_i, \alpha, \beta, fin, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_PA***. When a primitive activity is started the agents send the duration of their current activity to the scheduler. The scheduler receives all the activity times then determines which activity time is the smallest and updates its own clock based on this duration. When an agent's time is different to the system clock's it then changes accordingly and subtracts the time increment from the duration of its activity.

**Pop_PASend**. This is the rule the agent's use to send the duration of their next event to the scheduler.

<u>RULE:</u> Pop_PASend

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[T_\xi = T_i \wedge \beta = \langle \beta_d, Prim\_Act^t; \beta_{ins}\rangle]{B_\xi = B_\xi \cup (T_i = T_i + t)} \langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_PA(t>0)**. This rule is invoked when the agent's time is no longer the same as the scheduler's time. Additionally this rule checks whether the current activity's duration will be greater than zero after updating the times and durations.

<u>RULE:</u> Pop_PA(t>0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) > 0 \wedge \beta = \langle \beta_d, Prim\_Act^t; \beta_{ins}\rangle]{t' = (T_\xi - T_i) \wedge T_i = T_\xi \wedge Prim\_Act^t = Prim\_Act^t[t/t'] \wedge ag_i[ag_i^{stage} \in \{Pop\_concWf*\}/ag_i^{stage} \in \{fin\}]} \langle ag_i, \alpha, \beta, fin, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_PA(t=0)**. This rule is for when the agent's activity is due to finish at the end of the next clock tick. This rule directs the agent to only executing conclude statements before finishing for the cycle.

<u>RULE:</u> Pop_PA(t=0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) = 0 \wedge \beta = \langle \beta_d, Prim\_Act^t; \beta_{ins}\rangle]{T_i = T_\xi \wedge \beta = \langle \beta_d, \beta_{ins} - Prim\_Act^t\rangle} \langle ag_i, \alpha, \beta, Pop\_concWF*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_move***. Move activities are very similar to primitive activities, except when the activity terminates a belief update is performed to change the agents and the environments beliefs of the agent's current location. This belief update occurs when the agent notices that the duration of the move has reached zero after the clock update.
**Pop_PASend**. This is the rule the agent's use to send the duration of their next event to the scheduler.

<u>RULE:</u> Pop_moveSend

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[T_\xi = T_i \wedge \beta = \langle \beta_d, move(Loc = new)^t; \beta_{ins}\rangle]{B_\xi = B_\xi \cup (T_i = T_i + t)} \langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

Note. 'Loc = new' refers to the allocation of the *location* to the *new location.*

**Pop_move(t>0)**. Like for primitive activities, the move activity needs a rule for when the activity still has time remaining after the clock tick.

<u>RULE:</u> Pop_move(t>0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$
$$\xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) > 0 \wedge \beta = \langle \beta_d, move(Loc=new)^t; \beta_{ins} \rangle]{t' = (T_\xi - T_i) \wedge T_i = T_\xi \wedge move(Loc=new)^t = move(Loc=new)^t [t/t'] \wedge ag_i [ag_i^{stage} \in \{Pop\_concWf*\} / ag_i^{stage} \in \{fin\}]}$$
$$\langle ag_i, \alpha, \beta, fin, B_i, F, T_i, TF_i, WF_i \rangle$$

**Pop_move(t=0)**. Likewise, the move activity needs a rule for when the activity duration ends.

<u>RULE:</u> Pop_move(t=0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$
$$\xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) = 0 \wedge \beta = \langle \beta_d, move(Loc=new)^t; \beta_{ins} \rangle]{T_i = T_\xi \wedge \beta = \langle \beta_d, \beta_{ins} - move(Loc=new)^t \rangle \wedge B_i' = B_i [Loc=old / Loc=new] \wedge F' = F[Loc=old / Loc=new]}$$
$$\langle ag_i, \alpha, \beta, Pop\_concWF*, B_i', F', T_i, TF_i, WF_i \rangle$$

Note. 'old' refers to the previous *location* of the agent.

**Pop_comm***. Communication is very similar to a move activity, except the agent does not update its own beliefs or the environments beliefs but it updates another agents beliefs.
**Pop_commSend**. This rule sends the scheduler the duration of the communication.

<u>RULE:</u> Pop_commSend

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$
$$\xrightarrow[T_\xi = T_i \wedge \beta = \langle \beta_d, Comms(ag_j, b')^t; \beta_{ins} \rangle]{B_\xi = B_\xi \cup (T_i = T_i + t)}$$
$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

**Pop_comm(t>0)**. For when the communication has time remaining after the system clock tick.

<u>RULE:</u> Pop_comm(t>0)

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$
$$\xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) > 0 \wedge \beta = \langle \beta_d, Comms(ag_j, b')^t; \beta_{ins} \rangle]{t' = (T_\xi - T_i) \wedge T_i = T_\xi \wedge Comms(ag_j, b')^t = Comms(ag_j, b')^t [t/t'] \wedge ag_i [ag_i^{stage} \in \{Pop\_concWf*\} / ag_i^{stage} \in \{fin\}]}$$
$$\langle ag_i, \alpha, \beta, fin, B_i, F, T_i, TF_i, WF_i \rangle$$

**Pop_comm(t=0)**. Rule for when the communication activity duration ends.

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$
$$\xrightarrow[T_\xi != Ti \wedge (T_i + t - T_\xi) = 0 \wedge \beta = \langle \beta_d, Comms(ag_j, b')^t; \beta_{ins} \rangle \wedge b \in B_j]{T_i = T_\xi \wedge \beta = \langle \beta_d, \beta_{ins} - Comms(ag_j, b')^t \rangle \wedge B_j' = B_j [b/b']}$$

$$\langle ag_i, \alpha, \beta, Pop\_concWF*, B_i, F, T_i, TF_i, WF_i\rangle$$

Note. Belief exchange via communication is handed directly in Brahms, i.e., when an agent communicates with another, it directly changes the other agent's beliefs.

**Pop_emptyTf**. Concludes do not use any simulation time during execution, since thoughtframes only contain concludes then an agent will keep executing thoughtframes until it no longer has any to execute. This rule is for selecting a new thoughtframe when the current one becomes empty.

Rule: Pop_emptyTf

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\alpha=\langle\alpha_d,\emptyset\rangle]{\alpha\in\{\emptyset\}\wedge ag_i[ag_i^{stage}\in\{Pop\_*\}/ag_i^{stage}\in\{Tf\_*\}]}$$
$$\langle ag_i, \alpha, \beta, Tf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_emptyWf**. A workframe which only contains concludes will act like a thoughtframe. This rule is for such workframes so the agent can keep select another workframes when the current one becomes empty.

Rule: Pop_emptyWf

$$\langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\beta=\langle\beta,\emptyset\rangle]{\beta\in\{\emptyset\}\wedge ag_i[ag_i^{stage}\in\{Pop\_*\}/ag_i^{stage}\in\{Wf\_*\}]}$$
$$\langle ag_i, \alpha, \beta, Wf\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

**Pop_* rules in *PROMELA***. The rules for popping the elements off the stack are all handled by the same do-while loop used with detectables. The loop has conditions for activities, concludes and for when the activity is finished. Concludes have been handled as negative integers in *PROMELA* with each conclude having its own negative integer. In the do-while loop one of the conditions specifies the number is negative (i.e., a conclude), this section of code contains many if-statements checking which conclude the event is representing. The appropriate belief/fact updates are then made. These statements collectively represent rules Pop_WfconcB, Pop_WfconcF and Pop_WfconcBF.

```
/*Represents the conclude section of the loop.
wf_stackRobot[wf_RobotTop] refers to the current
element on the deed stack.  -84 represents the ID
number of a conclude*/
do
::(wf_stackRobot[wf_RobotTop] < 0 && ... && wf_RobotTop > 5)->
   /*Element is < 0, processing a conclude*/
   if
   ::(wf_stackRobot[wf_RobotTop] == -84 &&
     wf_RobotTop > 5) ->
       /*Update the belief*/
       Robot_checkMedicationA[RobotID] = false;
```

```
        /*Update the fact*/
        fact_checkMedicationA[RobotID] = false;
   fi;
   if
   ::(wf_stackRobot[wf_RobotTop] == -75 &&
     wf_RobotTop > 5) ->
   ...
   fi;
od
```

The activity duration in *PROMELA* is handled using a variable called timeRemaining and by marking the difference in time between the global clock and the agent's own clock. The *PROMELA* coordinates the sending and receiving of the duration of activity for the agents. The agents execute the rule Pop_PASend where they find the duration of their activity and store it in their timeRemaining variable, they then pass control back to the scheduler. The following code shows how the Pop_PASend is tied in with the detectables.

```
/*Check if a detectable has been fired, if not, i.e., "null"
perform Pop_PASend*/
if
::(activeDetectableType == impasse)->
   ...
::(activeDetectableType == continue)->
   ...
::(activeDetectableType == abort)->
   ...
::(activeDetectableType == complete)->
   ...
::(activeDetectableType == null)->
   /****represents Pop_PASend****/
   Robot_timeRemaining = wf_stackRobot[wf_RobotTop];
   /*Set agent's time = scheduler's time*/
   cntRobot = cntEnvironment;
   /*pass control to the scheduler*/
   turn = Environment
fi;
```

Eventually all agents perform this task and the scheduler will be at the end of the cycle. The scheduler uses a collection of if-statements to determine the shortest duration of all the timeRemaining variables and update its clock; the current time + this duration. The scheduler then returns to issuing control of the cpu back the agents. The agents are then in a different section of code (representing Pop_PA(t>0) and Pop_PA(t=0)) which is asking them if their time is different to the global clocks. If their time is different, the clock has moved forward, then the time difference is subtracted from their current activity and their clock is set to the same time as the global clock. An additional if-statement is used to determine if the rule Pop_PA(t>0) is active, a check for Pop_PA(t=0) is not made because time can not be deducted if the activity is duration is 0. When Pop_PA(t>0) activates the time is deducted from the activity's duration, then rule Pop_PA(t=0)) is checked. If the duration has been reset to 0 (i.e., Pop_PA(t=0)) is active) then the agent sets a 'concludes' flag to true and returns to pop the stack. This concludes flag directs the agent to the rules Pop_concWF*, as represented in the semantics of Pop_PA(t=0)). This is because the agent needs to make the appropriate belief updates after an activity has been finished.

```
if /*check if time is different to scheduler*/
::(cntRobot != cntEnvironment && turn == ag_Robot) ->

   /*find time difference*/
   timeDeduction = cntEnvironment - cntRobot;

   /*Set agent's time to match the scheduler*/
   cntRobot = cntEnvironment;

   /*Sanity check: incase there is no activity or if
   it is a communication*/
   if
   ::(wf_RobotTop != -1 && commRobot == false)->
      /****rule Pop_PA(t>0)****/
      if
      ::(wf_stackRobot[wf_RobotTop] > 0)->

         /*Find the new duration of the activity*/
         new = wf_stackRobot[wf_RobotTop] - timeDeduction;

         /*Assign new value for the activites duration*/
         wf_stackRobot[wf_RobotTop] = new;

         /*rule Pop_(PA=0); Rule is nested because Pop_(PA=0) needs
         to be processed as soon as t=0*/
         if
         ::(wf_stackRobot[wf_RobotTop] == 0)->
            concludes = true; /*the 'concludes flag'*/
            goto popstack; /*go back to Pop_* rules*/
         ::else->
    /*move on to checking thoughtframes*/
            goto thoughtframes;
         fi;

      ::else->
         skip;
      fi;
   ::else->
      skip;
   fi;

/*Time is the same as the schedulers*/
::(cntRobot == cntEnvironment && turn == ag_Robot) ->
   /*process thoughtframes*/
   goto thoughtframes;
fi;
```

The *PROMELA* translation handles move and communication activities using negative numbers, like concludes. Pop_Move* and Pop_Comm* rules are initially handled as concludes; this is because move and communication activities are essentially primitive activities followed by a belief/fact update. Therefore they are treated as such in the *PROMELA* translation. The first time a move or communication is encountered a flag in the header data is set to true and a primitive activity is pushed onto the stack. The primitive activity is processed as normal and when it is finished it is removed from the stack. The negative number of the move or communication activity is now again at the top of the stack, but this time the flag indicates to process the belief update. A move activity (Pop_Move*) operates in the same way but the belief updates corresponds to the change in location. The following code describes the implementation of a move activity:

```
/*The move activity is represented as -16 on
the stack*/
::(wf_stackRobot[wf_RobotTop] == -16 && wf_RobotTop > 5) ->
   /*Check if we have previously visited this activity
      wf_stackRobot[4] == 0 means we have not
      wf_stackRobot[4] == 1 means we have
      and check if we are only processing concludes*/
   if
   ::(wf_stackRobot[4] == 0 && concludes == false)->
      /*increase size of the deed stack*/
      wf_RobotTop = wf_RobotTop+1;

  /*Find ID number of the robots current location*/
      findID(Robot_location[RobotID]);
      currentLoc = searchID - 8;

  /*Find the ID number of the destination*/
      findID(sinkTwo);
      targetLoc = searchID - 8;

  /*Locate the distance and set as the
     agent's activity time*/
      wf_stackRobot[wf_RobotTop] =
         adjacency[currentLoc].edges[targetLoc];

  /*Set current activity duration as the distance
  to travel to new location*/
      Robot_timeRemaining = wf_stackRobot[wf_RobotTop];

  /*Flag that we have visited this move*/
      wf_stackRobot[4] = 1;
   /*Guard to check if it is concludes only*/
   ::(wf_stackRobot[4] == 0 && concludes == true)->
      concludes = false;
      goto processWorkframes;

/*Guard for if the move has previously been visited*/
::(wf_stackRobot[4] == 1)->
      /*Update the belief*/
      Robot_location[RobotID] = sinkTwo;

      /*Update the fact*/
      fact_location[RobotID] = sinkTwo;

  /*Remove activity from the stack
  and reset the flag*/
      wf_RobotTop--;
      wf_stackRobot[4] = 0;
fi;
```

### 7.2.5 Var_* rules (Variables)

Variables are used to represent quantification and unification in Brahms. A variable tells an agent to perform an activity in relation to an agent (or agents) who meet a certain requirement, a requirement which is defined in the guard condition. For example: request a hammer from an agent who is currently holding a hammer, or take a supply of nails to all agents who are holding a hammer. Variables operate in the same way on both workframes and thoughtframes.

**Var_empty**. When a workframe without variables is found it is forwarded to popping the stack.

Rule: Var_empty

$$\langle ag_i, \alpha, \beta, Var\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\beta\notin\{\emptyset\}\wedge\beta^V\in\{\emptyset\}]{ag_i[ag_i^{stage}\in\{Var\_*\}/ag_i^{stage}\in\{Pop\_*\}]} \langle ag_i, \alpha, \beta, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

Note. Where $\beta^V$ represents the variables contained within workframe $\beta$.

This rule is not directly handled in the *PROMELA* translation, but it is actually handled using the Java code. The *PROMELA* code is written from this Java code and there are if-statements which checks the variables contained within a frame. If the set of variables is empty then the **Var_*** rules are bypassed.

**Var_set**. Frames with variables create instances of themselves with the variables assigned to specific agents and objects. The 'selectVar()' command is what assigns these agents and objects to the variables.

Rule: Var_set

$$\langle ag_i, \alpha, \beta, Var\_*, B_i, F, T_i, TF_i, WF_i\rangle \xrightarrow[\beta=\langle\beta_d,\emptyset,\beta_{ins}\rangle]{\beta'=\langle\beta_d,[\emptyset\cup selectVar()],\beta_{ins}\rangle} \langle ag_i, \alpha, \beta', Var\_*, B_i, F, T_i, TF_i, WF_i\rangle$$

If the Java code identifies a frame as having a variable then it writes the *PROMELA* code differently. The first difference being the guard condition, if there are no variables then the whole guard condition is represented in a single if-statement, if there are variables then the guards need to be broken up into those tied to variables and those not e.g., a guard condition saying the current agent has battery value greater than 50% does not have a variable tied to it, but a guard condition saying forone agent who is holding a hammer does. With a frame for variables the guard conditions are then broke up into if-statements and do-while statements. There is always an initial if-statement which holds all the guard conditions not tied to a variable, if none exist then the if-statement states:

```
if
::(true)->
   ...
fi;
```

A do-while loop is then used for each variable in the frame e.g., if there are 3 variables then there will be 3 do-while loops. Each of these loops then represent a rule of type **Var_one**, **Var_each** or **Var_all** depending on what type the variable it is.

**Var_one**. A 'forone' variable is used for unification, to essentially select a single agent or object which meets the requirements in the guard condition.

<u>RULE:</u> Var_one

$$\langle ag_i, \alpha, \beta, \mathit{Var\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$
$$\xrightarrow[\beta=\langle\beta_d, W_0 \ldots W_n, \beta_{ins}\rangle \wedge \exists v \in \beta^V | v^{type}=forone]{\beta'=\langle\beta_d, Random(W_0 \ldots W_n), \beta_{ins}\rangle \wedge ag_i[ag_i^{stage} \in \{\mathit{Var\_*}\}/ag_i^{stage} \in \{\mathit{Pop\_*}\}]}$$
$$\langle ag_i, \alpha, \beta', \mathit{Pop\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$

Note. Where 'Random' is a random selection of one of the instances and '$v^{type}$' represents the variables type (forone, foreach or collectall).

**Var_each**. A 'foreach' variable will operate the action on all the agents or objects which meet the requirements in the guard condition, this is performed sequentially.

<u>RULE:</u> Var_each

$$\langle ag_i, \alpha, \beta, \mathit{Var\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$
$$\xrightarrow[\beta=\langle\beta_d, W_0 \ldots W_n, \beta_{ins}\rangle \wedge \exists v \in \beta^V | v^{type}=foreach]{\mathit{WF'}_i=\mathit{WF}_i \cup (W_0[W_0^{pri}/(\beta^{pri}+0.1), W_0^r/W_0^r=false] \ldots W_n[W_n^{pri}/(\beta^{pri}+0.1), W_n^r/W_n^r=false])}$$
$$\langle ag_i, \alpha, W_0, \mathit{Pop\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$

**Var_all**. The 'collectall' variable is similar to the 'foreach' except it performs the task on all the agents instantly. An example where this could be used would be picking up a pile of objects in a single move e.g., pick up all papers in stack A.

<u>RULE:</u> Var_all

$$\langle ag_i, \alpha, \beta, \mathit{Var\_*}, B_i, F, T_i, \mathit{TF}_i, \mathit{WF}_i\rangle$$
$$\xrightarrow[\beta=\langle\beta_d, W_0 \ldots W_n, \beta_{ins}\rangle \wedge \exists v \in \beta^V | v^{type}=forall]{\beta'=concludes(W_0 \ldots W_n) \wedge ag_i[ag_i^{stage} \in \{\mathit{Var\_*}\}/ag_i^{stage} \in \{\mathit{Pop\_*}\}]}$$

$$\langle ag_i, \alpha, \beta', Pop\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

Note. $concludes(W_1...W_n)$ is a method which takes all the workframe instances $W_1...W_n$ and extracts the concludes statements.

Each of these rules **Var_one**, **Var_each** and **Var_all** are performed inside a do-while loop. The guard conditions for each variable (each variable can have multiple guards) are placed inside the do-while loop specific to that variable. The do-while loops then loop through all the agents and objects and checks if they match the required guard condition. A check is also performed on whether the agent is in the correct class or group, as variables specify groups or classes of agents and objects e.g., 'foreach' agent in class 'Robot' who meets the requirement A, B and C. The do-while loops for all the variable types are very similar with only some minor differences. The do-while loop representing the rule **Var_each** is the most simple, this adds every agent who meets the requirements to an array associated to the frame. The **Var_all** is almost identical except sets a flag to identify it as a 'collectall'. The **Var_one** variable however only wants to select a single agent or object so uses a break statement when one is found. However, the semantics describe this selection process as a random selection where this is simply taking the first agent or object it finds. It is planned in future work to make this selection a more random process. The following is *PROMELA* pseudocode of a workframe with a foreach and a forone:

```
if
::(/*non-variable guards are true*/)->
   do
   ::(/*The forone variables guards are true*/)->
      do
      ::(/*The foreach variables guards are true*/)
         /*Add forone variable to array at index 0*/
         /*Add foreach variable to array at index 1*/
         /*Increment array pointer*/
 /*Mark the frame as active*/
      ::(/*A guard is false for the foreach variable*/)
        skip;
      ::(/*Looped through all agents or objects*/)->
         break;
      od;
      if
      ::(/*Array pointer has not been incremented, i.e., equals 0*/)
         /*set frame as inactive because no agents were found
         for the foreach variable*/
      ::else->
         skip;
      fi;
      break;

   ::(/*A guard is false for the forone variable*/)->
      skip;
   ::(/*Looped through all agents or objects*/)->
      break;
```

| variable | Index0 | Index1 | Index2 |
|---|---|---|---|
| *forone* | Bob | Bob | Bob |
| *foreach* | Robot | Careworker | RobotHouse |
| *forone* | Robot | Robot | Robot |

Table 7.9: A table holding the values of variables in a workframe.

```
   od;
::else->
   /*Set frame as inactive*/
fi;
```

The details of the agents and objects used for the variables for the frame are stored in a two dimensional array; one index for the variables and the other for storing the agents and objects matching that variable. An example table to demonstrate this array is shown in Table 7.9. This table shows three different variables being used in the workframe; 2 forone and a foreach. In this table there is a forone which has selected Bob, another forone which has selected the Robot and a foreach which has selected the Robot, the Careworker and the RobotHouse.

Each workframe and thoughtframe with variables has an array like Table 7.9 to reference which variables have been selected. One of the requirements for **Var_set** to become active is that there must not already be a set of values tied to the variables. The *PROMELA* code identifies this by checking if the array pointer has a value greater than -1. Using Table 7.9 as an example, the pointer will start at value 2 (because that is the largest index) and decrement every time the workframe is been processed, thereby performing the workframe for each agent matching the requirements. Eventually this pointer will become -1, this will indicate that a new set of variable values needs to be assigned, i.e., rule **Var_set** is active. Some additional code is required when processing concludes and activities which use these variables; for concludes the agent or object the belief is about needs to be referenced from the array (like in Table 7.9) and for an activity such as communication the agent will need to look up which agent or object the communication will be made to. For an example, we provide some pseudocode to show how a conclude with a variable will behave in a hypothetical workframe which uses a variable to locate who the Robot will feed. The element off the deed stack is a conclude of ID -72 and **wf_RobotTop** $>$ **5** is to ensure we are not in the header data of the workframe. The **findID** is a macro used to find the ID number of an agent when given their name. Inside the brackets is a reference to the current variable for the workframe, **Robot_feed_var** refers to the array holding the variables for the agent Robot and the workframe feed. The string **Robot_feed_index** refers to the set of variables under consideration and the **0** in the second index refers to the first variable. The macro **findID** will return the value of agent's ID number using the integer named **searchID**. The line stating **Robot_fed**[**searchID**] refers to the Robots belief about the attribute 'fed' which belongs to the agent who's ID number is stored in **searchID**. This line then concludes that the belief is changed to true. This pseudocode is as follows:

```
::(wf_stackRobot[wf_RobotTop] == -72 && wf_RobotTop > 5) ->
   findID(Robot_feed_var[Robot_feed_index].var_elements[0]);
   Robot_fed[searchID] = true;
```

## 7.3 Justification of The *PROMELA* Translation

The operational semantics created for Brahms in [79] were directly used for developing the *PROMELA* implementation of the Brahms framework. The operational semantics had no formal basis for an in-depth comparison to the Brahms framework, so instead of performing an analysis against the formal semantics we decided to perform a direct comparison between the outputs of the *PROMELA* against the actual Brahms framework. The Spin model checker has the ability to run *PROMELA* code as a simulation as well as for creating a model for model checking. This simulation mode was used to view the updates of beliefs and facts so a comparison could be made when the same Brahms simulation was ran in the Brahms framework. As the semantic rules were implemented tests were performed to analyse their correctness, in respect to the Brahms framework. The tests were performed incrementally. The tests were based on the RobotHelper scenario, where the scenario would steadily get more detailed. So at the beginning the agents would start, initialise their beliefs and then terminate. More detailed activities would be introduced, such as starting a workframe and exiting the workframe. The analyses on the correctness of the system were mainly based on two factors; the global clock and belief/fact updates. The simulations were run in Brahms first; details were noted on what beliefs changed, when they changed and what they were changed to. The same simulations were run in the *PROMELA* implementation and the results were compared. The process was repeated until all the desired functions were implemented and all the bugs were fixed so that the outputs matched.

The final comparison was performed using the Spin model checker. The scenarios were created (The RobotHelper and DigitalNurse) and simple properties were incrementally verified after every small change made. Making a comparison to the output produced by Brahms here was slightly more difficult because there would be some small amounts of non-determinism. Initially models were made as deterministic as possible to simplify the process. The results of the verification were analysed manually against the output of Brahms, e.g., if a property verified had specified that Bob would always take his medicine then the results of a simulation ran in the Brahms framework would be checked to see if this was the case. With the addition of non-determinism the analysis would be conducted across multiple simulations, simulations which had been engineered to produce all the possible outputs. However, the larger the models became the more difficult it was to do this. When the models were too large for manual testing we would test a handful of samples manually and then make a judgement on whether the property should or should not be verifiable.

# Part III

# Case Studies

# Chapter 8

# Home Helper Robot Scenario

## 8.1 Overview of Scenario

This first scenario was developed as a test bed for the verification of human-agent teamwork. The idea behind this scenario was to invent a case study, one based on a possible real life scenario, which demonstrates low level teamwork involving humans and agents. The scenario was incrementally built adding functionality along with specifications to verify, this was to ensure the correctness of the verification and of the scenario. While doing this, simple issues with the design of the scenario would prevent some specifications from being verifiable, e.g., a robot not activating a workframe to clean some dirty plates because a guard condition was incorrectly defined. The low level teamwork this scenario was interested in evaluating was that of individual agents who communicate with each other to fulfil their own personal goals, e.g., a person requesting food from a robot, and the robot complying with the request. Since this was a simple scenario there was little emphasis on joint activities and joint goals. The non-determinism in this scenario was presented through an outside event, i.e., the event of a fire. The reason behind this choice of non-determinism is because we want to program the agents with deterministic protocols, but then we want to test how they manage in the event of an emergency.

In this scenario there is a person with dementia (Bob), a helper robot, a human care worker and a house agent. The helper robot is mobile and can move about the house assisting the person with various tasks. The house agent has the role of detecting information, informing the person and reminding them of things where necessary. The care worker is called for when the robot/agent are unable to assist. Such domestic health-care scenarios typically involve assisting the elderly or infirm; see for example [57, 65].

Figure 8.1 provides a description of this scenario explaining the roles of the person with dementia, the helper robot, the human care worker and the house agent. To help explain Figure 8.1 we provide the following description of all the humans and agents roles below:
The helper robot:

1. fetches drinks, cooks food, and delivers them to the person
2. collects dirty dishes and puts them in the dishwasher
3. fetches medicines
4. records whether the person has their medication

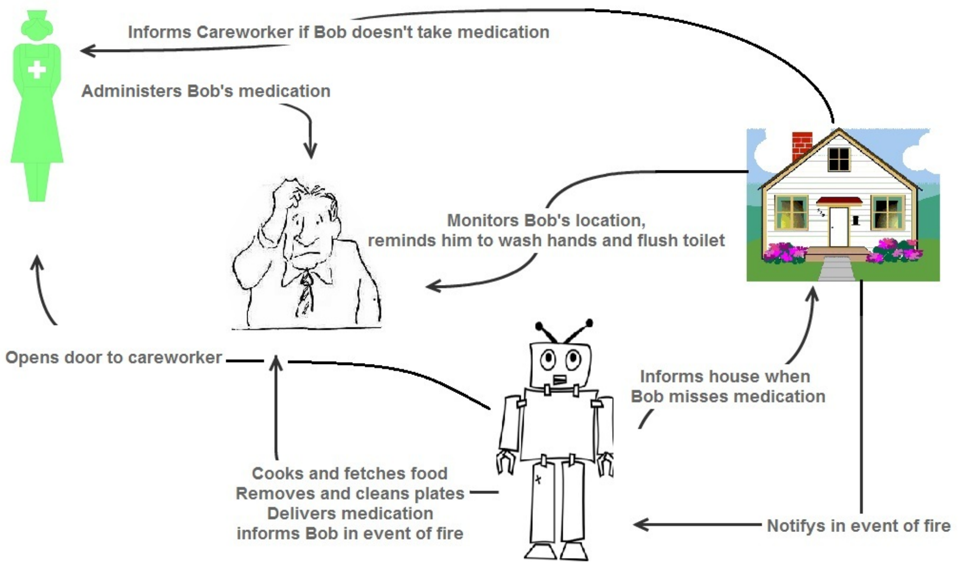

Informs Careworker if Bob doesn't take medication
Administers Bob's medication
Monitors Bob's location, reminds him to wash hands and flush toilet
Opens door to careworker
Informs house when Bob misses medication
Cooks and fetches food
Removes and cleans plates
Delivers medication
informs Bob in event of fire
Notifys in event of fire

Figure 8.1: Overview of the Robot Helper Scenario

5. informs the person of what to do in case of an emergency, e.g., a fire
6. communicates with the house agent
7. answers the door

The house agent:

1. informs the helper robot of the person's location
2. reminds the person to flush the toilet
3. monitors the person's location
4. notifies robot of fire
5. informs careworker if person does not take medication

The care worker:

1. administers the medication; which we assume is 100% successful

The person:

1. requests food
2. goes to the toilet at regular intervals
3. watches television

## 8.2 Brahms Representation

This scenario is modelled in Brahms using five agents and one object: *Robot*, *The House*, *Environment*, *careworker* and *Bob* (our elderly person) are the four agents and *Clock* is the object. The *Clock* is used for termination of the simulation (i.e., after 20 hours) and provides the notion of time used by the simulation, e.g., governing when the human's hunger increases. The *Environment* is a simple agent which decides if, and when, a fire alarm will occur. *Bob*'s role is to mainly watch television and to perform simple everyday tasks such as to eat and go to the toilet. Thoughtframes are used to update beliefs about how hungry he is and how much he needs the toilet.

When his hunger reaches a certain threshold a workframe activates and Bob requests food. A similar workframe will trigger a visit to the toilet. These workframes have a higher priority than the workframe for watching television, so when they become active the 'television' workframe *suspends*. The workframe for going to the toilet activates other workframes to flush the toilet and wash his hands once finished. Two versions of these workframes exist: representing whether or not he remembers to perform the task, each have the same guard conditions and priority so only one will execute at random. Bob also has workframes for taking his medication and thoughtframes that govern whether or not he chooses to do so.

The helper *Robot* remains idle until a command is made or it detects Bob requires attention. When Bob makes a request for food the Robot prepares and delivers the food. There is a detectable in the Robot's "wait for instructions" workframe which detects when Bob has finished eating; this triggers a belief update which in turn triggers a workframe to clear the plates. The Robot also has workframes to deliver medicine to Bob; activated at pre-allocated times. The Robot places the drugs on Bob's tray and then monitors them, checking every hour if they have been taken. The workframe governing this is shown in Fig. 8.2. A detectable `takenMedicationC` aborts the workframe if the drugs have been taken and then updates the Robot's beliefs. If the drugs have not been taken the workframe reminds Bob to take his medication. The Robot counts the number of times it reminds Bob, and after 2 reminders the Robot notifies the House. The Robot also instructs Bob to evacuate the house if a fire alarm has sounded and answers the door to the care worker.

*The House* is 'intelligent'. It has the responsibility for monitoring Bob, giving him instructions based on his location and detecting any fire. The House's default workframe monitors Bob, and has detectables which update the House's beliefs about Bob's location. When Bob's location is on the toilet a new workframe is fired, this workframe contains an 'abort' detectable which quits the activity when Bob leaves the toilet and activates a new workframe which detects Bob's location and uses this to decide whether or not Bob has left without flushing the toilet. Bob is then reminded if necessary. The default monitoring workframe also has a detectable for the fire alarm, this aborts the current activity and activates a workframe which sounds an alarm and notifies the Robot and Bob. Finally, the House has a workframe for when it has been notified that Bob has failed to take his medicine, the House then informs the care worker.

The *Care Worker* performs outside activities which are abstracted into a single activity. When the care worker is called they will only make their way once they have finished their current activity. When the care worker arrives they ring the doorbell and once they are let in by the helper robot they administer the medication and inform the robot that the patient has taken the medication. The care worker then leaves and continues with their outside activities.

```
workframe wf_checkMedicationC {
   repeat: true;
   priority: 3;
   detectables:
      detectable takenMedicationC{
         when(whenever)
         detect((Bob.hasMedicationC = false),
          dc:100)
         then abort; }
   when(knownval(current.perceivedtime > 14)and
      knownval(Bob.hasMedicationC = true) and
      knownval(current.checkMedicationC = true))
   do {
      checkMedication();
      remindMedicationC();
      conclude((current.checkMedicationC =
       false));
      conclude((current.missedMedicationC =
       current.missedMedicationC + 1));
      }}
```

Figure 8.2: The Robot's workframe to remind Bob about medication

## 8.3 "Home Care" Verification

We next consider the actual verification of the human-agent-robot teamwork in this "home care" scenario.

### 8.3.1 Desirable Properties to Prove

We develop a range of logical properties for the scenario; recall that in *temporal logic*, $\Diamond\phi$ means that "$\phi$ will be true at *some* moment in the future", while $\Box\phi$ means that "$\phi$ will be true at *all* future moments". We describe the properties verified and classify these just by the core aspect they represent,, i.e., properties labelled F$n$ relate to the fire alarm; labelled by T$n$ relate to the toilet; H$n$ relate to hunger and M$n$ relate to medicine. The propositions used in the properties are all based on the beliefs of the agents or facts in the system. We expect all of these properties to hold apart from M1.

**FIRE ALARM.**

F1: This property was designed to check if *The House* will generate a fire alarm when a fire occurs. The property, explained in English, is: If a fire occurs then, eventually, *The House* believes the fire alarm has been activated. The property, as a temporal logic formula is: $\Box(\text{a} \Rightarrow \Diamond\ \text{b})$ where

$\text{a} =$ There is a fire

b = *The House* believes the fire alarm is sounding

F2: This property was designed to check that *The House* agent will shut the alarm off when *Bob* is no longer in danger. The property, explained in English, is: If the hourse believes a fire alarm is sounding, and *Bob* believes he has left *The House* then eventually *The House* will no longer believe the fire alarm is sounding. The property, as a temporal logic formula is: $\Box((a \wedge b) \Rightarrow \Diamond\neg a)$ where

a = *The House* believes the fire alarm is sounding

b = *Bob* believes he has evacuated the house

F3: This property was designed to check that the *Robot* will remind *Bob* to leave the house in the event of a fire. The property, explained in English, is: If *The House* believes a fire alarm is sounding and *Bob* believes he has not left *The House*, then the *Robot* eventually believes it has alerted *Bob* about the fire. Logical requirement is: $\Box((a \wedge \neg b) \Rightarrow \Diamond c)$ where

a = *The House* believes the fire alarm is sounding

b = *Bob* believes he has evacuated the house

c = *Robot* believes it has alerted *Bob* about the fire

**TOILET.**

T1: This property was designed to check that *Bob* does go to the toilet, since $T2$ will always evaluate to true if *Bob* does not go to the toilet. The property, explained in English, is: Eventually *Bob* believes he is on the toilet. The property, as a temporal logic formula is: $\Diamond a$ where

a = Bob believes his location is on the toilet

T2: This property was designed to check that *The House* agent does remind *Bob* to perform tasks, such as wash his hands after using the toilet. If *Bob* goes to the toilet he can forget to flush it and, if so, we verify that he will be reminded by the House. To simplify the property we assume that *Bob* will wash his hands once reminded. The property, explained in English, is: if *Bob* believes he is on the toilet then eventually the toilet will be flushed. The property, as a temporal logic formula is: $\Box(a \Rightarrow \Diamond b)$ where

a = *Bob* believes his location is the toilet

b = the toilet flushed is true

**HUNGER.**

H1: This property was designed to check if the *Robot* delivers *Bob* his food within a certain time. The issue with formulating this property was that we needed to accurately measure how long it takes for the *Robot* to return with the food. The most accurate and simplest method was to have the count performed by *Bob*. The count performed by *Bob* was not considered part of the scenario, since it would be unrealistic to expect an elderly person to do so. The count is activated as soon as *Bob* requests food and stopped and soon as the *Robot* places the food on the tray. The property, explained in English, is: If *Bob* believes he has asked for food then eventually the *Bob* believes the time he asked for food is less than 1 hour. The property, as a temporal logic formula is: $\square(a \Rightarrow \square\neg b)$ where

T = 1 hour

a = *Bob* believes his "asked for food" variable is true

b = *Bob* believes the time since he asked for food is greater than $T$

H2: This property was designed to check that the *Robot* will collect the dishes once *Bob* has finished eating. To simplify this property we assume that when the *Robot* is at the dishwasher then the dishes are inside the dishwasher. This is because the only time the *Robot* is at the dishwasher is to fill it up. The property, explained in English, is: Once *Bob* believes his plate is empty, then eventually the *Robot's* location is the dishwasher. The property, as a temporal logic formula is: $\square(a \Rightarrow \lozenge(a \Rightarrow b))$ where

a = *Robot* believes Bob's plate is empty

b = *Robot* believes its location is the dishWasher

**MEDICINE.**

M1: This property was designed to evaluate to false to show that our verification system did not always evaluate properties as true. The property itself was designed to verify either *Bob* always takes his medication or the *Robot* never reminds him to do so. Which should evaluate as false since if *Bob* does not take his medication then the *Robot* should remind him. The property, explained in English, is: Always *Bob* does not take believe he has taken his medication and always the *Robot* does not remind *Bob* to take his medication. In this case we *shouldn't* be able to verify: $\square\neg a \wedge \square\neg b$ where

a = The *Robot* believes it has reminded *Bob* to take his medication

b = *Bob* believes he has taken his medication

M2: This property was designed to test that the *Robot* will remind *Bob* to take his medication if he forgets to. It is difficult to verify that a communication occurs with our tool, so instead we verify that the recipient's set of beliefs are updated with the message details. The property, explained in English, is: If *Bob* believes he has his medication, but believes he has not taken it, then eventually *Bob* will believe he has taken it. The property, as a temporal logic formula is: $\square(a \Rightarrow \lozenge\neg b)$ where

a = *Bob* believes he has his medication

b = The *Robot* believes *Bob* has not taken his medication

M3: This property was designed to test whether *The House* will be informed when *Bob* does not take his medication. The property, explained in English, is: if the facts state that *Bob* has not taken any medicine, then *The House* will believe that *Bob* has not taken it. The property, as a temporal logic formula is: $\lozenge(a \Rightarrow b)$ where a = the fact states *Bob* has not taken his medication

b = *The House* believes *Bob* has not taken his medication

M4: This property was designed to test whether the *Care Worker* will arrive to administer *Bob's* medication when he does not take it, within a certain time limit. The property, explained in English, is: If *Care Worker* believes that *Bob* has not taken his medication then *The House* believes it has been less than 2 hours sine it notified the *Care Worker*, and eventually *Bob* takes his medication. The property, as a temporal logic formula is: $\Box((a \Rightarrow \Box\neg b) \wedge \Diamond c)$ where

$T$ = 2 hours

a = The *Care Worker* believes *Bob* has not taken his medication

b = *The House* believes the time since the *Care Worker* was informed of failure to take medication $> T$)

c = The *Care Worker* believes their location is Bob's chair

### 8.3.2 Verification Results

The properties F1, F2, F3, T1, T2, H1, H2, M2, M3 were all verified using `Spin` (i.e. the property holds on all paths from every initial state) in times ranging from T1 of 29.9 seconds to H1 of 848 seconds. As expected `Spin` shows that the property M1 is false and the time taken to find a trace in the model was 421 seconds. The property M4 was run multiple times to observe how changing the duration of the *Care Worker's* other duties affected the outcome. `Spin` was able to verify M4 so long as the *Care Worker's* other duties took less than 2 hours.

# Chapter 9

# Digital Nurse Scenario

## 9.1 Overview of Scenario

This second scenario was developed to verify more complex human-agent teamwork. The idea behind this scenario was, again, to invent a case study based on a possible real life scenario, which demonstrates a slightly higher level of teamwork involving humans and agents. This scenario was derived with the assistance of researchers at the Palo Alto Research Center (PARC), who are working on projects to develop devices for use inside a hospital. This scenario was also incrementally built adding functionality along with specifications to verify, to ensure the correctness of the verification and of the scenario. The level of teamwork in this scenario was increased to involve joint activities and to achieve joint goals, e.g., turning a patient is a joint goal between a robot and a nurse, and both are required to achieve the goal. The non-determinism in this scenario was also presented through an outside event, i.e., the event of a heart attack. This, again, is because we want to program the agents with deterministic protocols, so we want to test how they manage in the event of an emergency.

The Digital Nurse scenario was created to have multiple agents (some virtual) and humans working together as part of a team. The scenario involves 2 nurses, 1 doctor, 3 digital nurses, 1 robot and 1 agent to monitor the patients. The goal of the scenario is to take sufficient care of 5 patients. The five patients have certain needs such as food, water, medication and turning in their beds. Also certain patients are at risk of a heart attack, which provides an emergency for the scenario. The nurses have the duties of looking after the patients, however only one nurse at a time looks after the patients. The other nurse continues duties which are not considered as part of the simulation; this nurse covers the active nurse in the simulation during the nurse's break. The nurse will have a schedule to work to: turning the patients when needed, administering medication, responding to emergencies and feeding the patients. The digital nurse is essentially a scheduler for the nurses, reminding them when to perform their duties and informing them of emergencies. The doctor in the scenario has minimal responsibilities: check the patients, prescribe medication and respond to emergencies. The robot is a helping hand for the doctor and nurses: it aids the nurses in turning the patients, refills patient's water jugs, fetches the patient's medication, fetches the patient's food and responds to emergencies.

## 9.2 Brahms Representation

The scenario is modelled in Brahms using 12 Brahms agents and 2 Brahms objects. The agents are: 2 nurses (modelled as a humans), 1 doctor (modelled as a human), 3 digital nurses (one for each nurse, and one for the doctor, they are modelled as software agents), 5 patients (modelled as a humans) and 1 robot (modelled as a hardware/robotic agent). The objects are: 1 to monitor the patients and 1 for a clock. The patients drink water every so often, make a breakfast choice when prompted and might have a heart attack. The patient will only recover from a heart attack if the doctor, robot and a nurse all respond to resuscitate. The Monitor object keeps track of all the agents. It can detect when an agent's water is low and communicates this to the appropriate digital nurse. The Monitor agent also checks the patients' vitals so knows when one has a heart attack, dies or is no longer having a heart attack. The Monitor agent is also used for counting durations for verification purposes, e.g., it notes when a heart attack occurs and increments a counter until the patient is either dead or resuscitated. One of the nurses (Nurse_1) is assigned the responsibility of turning the patients but cannot do so until the robot is there to assist. To do this two workframes are used: 'wf_turnOne' and 'wf_turnTwo'. The workframe 'wf_turnOne' takes the nurse to the patient and waits for the robot to arrive, a detectable is used to abort this workframe when the robot is at the nurses location. If the robot does not arrive at the location in time then the nurse continues with the workframe and sets a flag to preventing 'wf_turnTwo' from executing. If the robot does arrive in time then workframe 'wf_turnTwo' becomes active which performs a primitive activity to turn the patient followed by belief updates to determine that the patient has been turned. The following is the Brahms code used to do this:

```
workframe wf_turnOne{
  repeat: true;
  priority: 1;
  variables:
    /*assign a variable to a patient that needs turning*/
    forone(Patient) pat;
/*assign a variable to the patient's location*/
    forone(bed) b;

  detectables:
  /*Detect if robot is at the patient's location*/
    detectable waitForRobot {
    when(whenever)
    detect((pat.location = Robot.location), dc:100)
/*if robot at patient location then abort the workframe*/
    then abort;
    }
  /*following guards represent:
       at time point 8
       patient has not been turned
       flag to determine if wf_turnOne or wf_turnTwo
           should be executed
       patient is one which needs turning
       this nurse is responsible for turning patients
```

```
        identify the location of the patient*/
  when(
    knownval(current.perceivedTime = 8) and
    knownval(pat.turned = false) and
    knownval(pat.readyToBeTurned = false) and
    knownval(pat.needTurning = true) and
    knownval(current.turnDuty = true) and
    knownval(pat.location = b))

  do {
   /*move to location identified*/
   moveToBed(b);
   /*flag for wf_turnTwo to true*/
   conclude((pat.readyToBeTurned = true));
   /*inform robot which patient to turn*/
   patientToTurn(pat);
   /*wait for the robot to arrive*/
   waitToTurn();
   /*robot has not arrived so set wf_turnTwo flag
    to false*/
   conclude((pat.readyToBeTurned = false));
  }
}

workframe wf_turnTwo{
  repeat: true;
  priority: 1;
  variables:
    forone(Patient) pat;

  /*Guards check that:
    wf_turnTwo flag is true
this nurse is responsible for turning patients*/
  when(knownval(pat.readyToBeTurned = true) and
    knownval(current.turnDuty = true))

  do {
    /*turn the patients*/
    turnPatient();
/*reset all values and flags*/
    conclude((pat.turned = true));
    conclude((pat.readyToBeTurned = false));
    conclude((pat.timeSinceTurned = 0));
  }
}
```

The nurse responds to resuscitate patients when they have had a heart attack, and also asks the patients what they want for breakfast and makes the order. The doctor visits patients and decides what medication the patient will have, for simplicity the medication is just a number, e.g., medication 1, 2 and 3, etc. The doctor also resuscitates a patient when having a heart attack. The digital nurse is responsible for informing the nurses and doctor of events and passing messages. The digital nurse has workframes to:

1. remind the nurse when it is time for breakfast;
2. send breakfast orders to the robot;
3. inform the robot of medications the doctor has prescribed;
4. inform the nurse when it is break time;
5. arrange another nurse to cover a nurse's break; and
6. inform the nurses and doctor of when a patient has a heart attack.

All the agents have thoughtframes to manage time. The clock object uses a *collectall* variable to send the current time to all the agents. Each agent is a member of a *TimeKeepers* group which means they will all receive messages from the clock. Non-determinism was added to the scenario via the patients having a heart attack. Thoughtframes in the patient's code were added to achieve this. These thoughtframes become active at certain times (times due to the agent's belief of time, not simulation time). When these thoughtframes become active they then decide whether or not a heart attack will happen. A Boolean is set for each patient to decide whether or not they can use these thoughtframes, i.e., whether they are at risk of a heart attack or not. A single agent was put at risk of a heart attack and gradually these thoughtframes were added to test how much non-determinism there could be; three of these thoughtframes were added before all memory was exhausted.

## 9.3 "Digital Nurse" Verification

### 9.3.1 Desirable Properties to Prove

Again we use a range of logical properties for the scenario; recall that in *temporal logic*, $\Diamond\phi$ means that "$\phi$ will be true at *some* moment in the future", while $\Box\phi$ means that "$\phi$ will be true at *all* future moments". We describe the properties verified and classify these just by the core aspect they represent, i.e., properties labelled P$n$ relate to the patient; N$n$ relate to the nurses; R$n$ relate to the robot and DN$n$ relate to the digital nurses. The axioms used in the properties are all based on the beliefs of the agents or facts in the system.

**PATIENT.**

P1: This property was designed to test if an emergency will actually occur in the scenario, i.e., a patient has a heart attack. The property, explained in English, is: Eventually a patient will have a heart attack. The property, as a temporal logic formula is: $\Diamond a$ where

a = Patient_one has a heart attack

P2: When a patient has a heart attack they can only be revived if all members of the team arrive to perform their job, when they do we consider the resuscitation rate to be 100%. If not all members of the team arrive within a certain time frame then the patient dies. We construct the scenario this way for simplicity, so we can easily evaluate whether or not all the agents perform their required task. The property, explained in English, is: The patients are always alive. The property, as a temporal logic formula is $\Box(a \wedge b \wedge c \wedge d \wedge e)$ where

a = Patient_one is alive

b = Patient_two is alive

c = Patient_three is alive

d = Patient_four is alive

e = Patient_five is alive

P3: One of the roles of the nurse is to turn patients that need turning. This task requires the joint effort of the robot and the nurse, where the nurse must wait for the assistance of the robot before turning the patient. The team aspect of this task means it is possible that the patients end up waiting too long to be turned. To evaluate this we assign each patient a counter, where they start to count-up when they are due to be turned. This is not considered to not be part of the scenario, more of an addition to aid verification. The property, explained in English, is: The patients don't wait more than 1 hour when they need to be turned. The property, as a temporal logic formula is: $\Box(a \wedge b \wedge c \wedge d \wedge e)$ where

a = Patient_one's time for waiting to be turned is less than 1 hour

b = Patient_two's time for waiting to be turned is less than 1 hour

c = Patient_three's time for waiting to be turned is less than 1 hour

d = Patient_four's time for waiting to be turned is less than 1 hour

e = Patient_five's time for waiting to be turned is less than 1 hour

**NURSE.**

N1: One of the requirements of the scenario is that the nurse on duty (i.e., Nurse_one) needs to take a break, this break needs to then be covered by the nurse not on duty (Nurse_two). The digital nurse is used to notify the nurse of when to take a break. When the nurse is on a break a simple flag is used to indicate this. The property, explained in English, is: Eventually the nurse will have a break. The property, as a temporal logic formula is: $\Diamond a$ where

a = Nurse_one has a break

N2: In the scenario there is only one nurse performing the duties relevant to the scenario, the other performs 'other duties'. However, when the primary nurse in the simulation takes a break the other nurse must cover this nurse's duties. This relies on the communication between the digital nurses and the nurses, so that the secondary nurse takes over before the primary takes a break. The property, explained in English, is: There is always a nurse on duty. The property, as a temporal logic formula is: $\Box(a \vee b)$

a = Nurse_one is on duty

b = Nurse_two is on duty

N3: When the patient has a heart attack all members of the medical team are called to resuscitate the patient. We have already specified the property that the patient does not die in the simulation, but as a sanity check we wish to ensure this is because the team perform their resuscitation duties within the required time frame. Again the patient is given a counter to count simulation time to aid verification, this counter counts the duration since the patient has had a heart attack. Note that in the simulation it was only Patient_one that was put at risk of a heart attack. The property, explained in English, is: If a heart attack occurs then eventually the nurse, the doctor and the robot will resuscitate the patient within 4 minutes. The property, as a temporal logic formula is: $\Box(a \Rightarrow \Diamond(b \wedge c \wedge d \wedge e))$ where

a = Patient_one has a heart attack

b = nurse performs resuscitation workframe

c = doctor performs resuscitation workframe

d = robot performs resuscitation workframe

e = time since heart attack is less than 4 minutes

**ROBOT**

R1: One of the robot's duties is to ensure the patient's water jugs always have water. In the scenario we have a sensor attached to the object monitoring the patients which informs the robot of when a patient's water is low and the robot should then refill this patient's water jug. This is a low priority task but it is still essential so we need to ensure this task is still performed, and in a timely fashion. To verify this property the sensor counts the duration since it flagged the water as low, if this duration exceeds an hour then an alarm is sounded. The property, explained in English, is: the low water alarm is never sounded. The property, as a temporal logic formula is: $\Box(\neg a)$ where

a = low water alarm is sounded

R2: The doctor's only duty in the simulation is to examine each patient and prescribe medication. Once a patient is prescribed a medication then the robot has the job of retrieving this medication for the nurse to administer. The doctor will visit each patient in turn, tell the digital nurse the prescription who in turn notifies the robot. The requirement we wish to verify is that at no point does the robot have the wrong prescription for the patient. The property, explained in English, is: The robot either has no belief of the patient's medication requirement or it matches what the doctor has prescribed. The property, as a temporal logic formula is: $\Box((a \vee b) \wedge (c \vee d) \wedge (e \vee f) \wedge (g \vee h) \wedge (i \vee j))$ where

a = robot has no belief about Patient_one's medication

b = robot's belief about Patient_one's medication matches the doctor's belief

c = robot has no belief about Patient_two's medication

d = robot's belief about Patient_two's medication matches the doctor's belief e = robot has no belief about Patient_three's medication

f = robot's belief about Patient_three's medication matches the doctor's belief

g = robot has no belief about Patient_four's medication

h = robot's belief about Patient_four's medication matches the doctor's belief

i = robot has no belief about Patient_five's medication

j = robot's belief about Patient_five's medication matches the doctor's belief

**DIGITAL NURSE**

DN1: The digital nurse has the job of informing the nurse of their duties and when to perform them. When the clock announces that 8 hours have passed in the simulation the digital nurse then has the responsibility to inform the nurse that it is breakfast time. The property, explained in English, is is: The digital nurse will notify the nurse it is breakfast time within 1 hour. The property, as a temporal logic formula is: $\lozenge(a \wedge b)$ where

a = breakfast has been announced

b = time is at least 8 but less than 9

DN2: This property is to check the reaction time of the digital nurses, ensuring that when an emergency occurs then the digital nurses inform the doctors, etc. in a timely fashion. The emergency in this case is a heart attack, again a counter is started by the patient to measure the duration since the heart attack occurred, which is used for verification purposes. For this property we use the beliefs of the agents to test whether the communications have been sent by the digital nurses. The property, explained in English, is: The digital nurse will notify the doctor, nurse and robot of a heart attack in less than 2 minutes of when the heart attack occurred. The property, as a temporal logic formula is: $\Box(a \Rightarrow (b \wedge c \wedge d \wedge e))$ where

a = patient_one has a heart attack

b = nurse_one believes patient_one has had a heart attack

c = robot believes patient_one has had a heart attack

d = doctor believes patient_one has had a heart attack

e = time since heart attack is less than 2

DN3: This property again checks the communication of the digital nurses, this time ensuring the nurse is informed to take a break. In the scenario the nurse is due for a break 10 hours into the simulation, so the property needs to check at the time is at least 10 but does not become 11. The property, explained in English, is: The digital nurse will send a notification to the nurse_one to take a break within 1 hour of when it is due. The property, as a temporal logic formula is is: $\lozenge(a \wedge b)$ where

a = the digital nurse believes the nurse has gone on a break

b = the time is at least 10 but not 11

### 9.3.2 Verification Results

All the properties above were verified using our tool, however R1 could initially not be verified. When analysing the results it was found that there was a small error in the implementation of the scenario. The robot would be notified that the patient's water was low, remove the water jug and execute the workframe to fill the jug. However, the patient were not recognising when the water jug was missing and would carry on drinking. The robot would set the fact for the jugs water level as full but immediately afterwards the patient would change the fact to match its belief about the water level - which is very low since the patient didn't know the water had been filled. The scenario was adjusted so that this would not happen but again the verification would fail. This was found to be because the the Monitor object was performing the count and could only count a single patient's jug at a time, but at one point more than one jug was becoming empty. To fix this we changed the Brahms model so that the patients would count how much time had passed since their own jug was low. Once this fix was made the verification of R1 was successful.

# Chapter 10

# Case Study Conclusions

Two case studies were chosen to demonstrate the verification of differing levels of human-agent teamwork. The two scenarios are:

- a home helper scenario; a person with dementia being aided by a robot, an intelligent house and a human careworker
- a hospital scenario; a doctor, 2 nurses, a robot and 3 digital assistants are working together to treat 5 patients

The home helper scenario was intended to be small and relatively deterministic for the purpose of analysing the results of the verification. The hospital scenario, labelled digital nurse, was intended to push the boundaries of the Brahms translation and the verification with multiple agents, more in-depth teamwork and more non-determinism. When designing these scenarios we had to take into account the Brahms functions utilised in these scenarios, since we will want to test as broad a range of functions as possible. The Venn diagrams in Figure 10.1 and Figure 10.2 demonstrate what Brahms constructs have been used in each scenario and all together. Out of all the Brahms constructs described in the operational semantics the only functions not represented in these case studies are: the impasse detectable and the foreach variable.

The verification results of these scenarios were promising, with our tool correctly verifying the properties that should be true and finding an error trace in properties that should not be verifiable. While conducting the verification there were instances where errors in the model were identified (such as R1 in the digital nurse scenario) and adjustments needed to be made so that the property could be verified. During verification there were some memory issues, where the verification process would consume all available memory rendering a property unverifiable. These issues were resolved but in the case of the digital nurse scenario the level of non-determinism had to be kept to a minimal amount. Overall the verification of these case studies demonstrated that accurate models of the Brahms scenarios were being created and that properties could be correctly verified. However, these case studies did highlight that, because of memory issues future work will be required in order to practically verify any Brahms models which are larger than the digital nurse scenario.

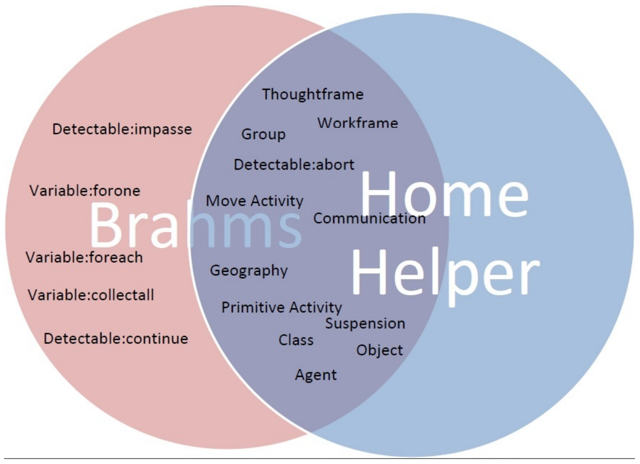


Figure 10.1: Venn diagram showing Brahms functions used in the home helper scenario

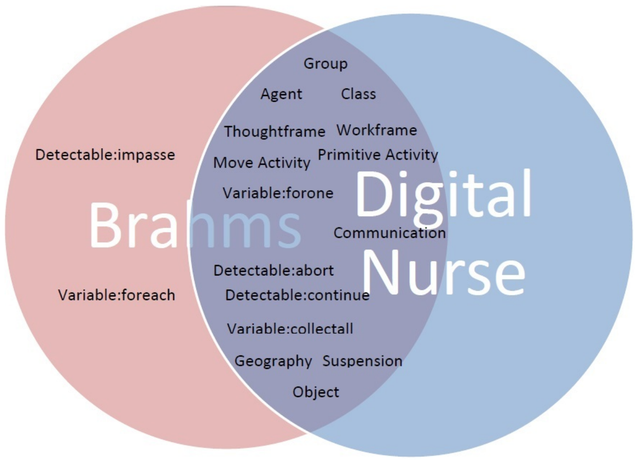


Figure 10.2: Venn diagram showing Brahms functions used in the digital nurse scenario

# Part IV

# Evaluation and Conclusions

# Chapter 11

# Evaluation of the Verification Performance

Next we consider the effect of our translation from Brahms models into the input language for Spin in terms of numbers of states generated in the model and verification times. Since this is the first tool for verifying Brahms models of human-agent teamwork we had no expectations on the performance. However, a performance issue became apparent when only minimal non-determinism could be added to the Digital Nurse example before all memory was consumed. The verification of the Robot Helper scenario also had memory issues, even on deterministic runs. We managed to identify that the cause of this issue was surplus states generated from the semantics, i.e., new states generated when the only variables changed are those holding states for loop counters, stack pointers, temporary variables, etc. We consider these states to be surplus because they do not provide any additional details about the state of the Brahms model. This issue was partially resolved by adding deterministic sections inside the *PROMELA*. These deterministic sections tell the Spin model checker to condense the code into a single state, with any non-determinism dealt with by random selections. Restrictions on the use of these deterministic wrappers meant that not all of the surplus states could be removed. Sections of code containing 'goto' statements could not be inside a wrap neither could sections containing the halting of a processes, this has increased the number of states generated and therefore increased the verification time and amount of memory used.

While the translation of the Brahms models into *PROMELA* is arguably correct there may be better ways to structure the code to improve the size of the state space. To further explore this we conducted some performance testing for agents carrying out very simple and scalable tasks. The main focus of the testing was on the number of agents, the number of actions the agents perform, workframes, thoughtframes, activities and communication. Our hypothesis is that increasing the size of the *PROMELA* code (by adding more Brahms constructs) and increasing the iterations through the semantics (number of time steps) will create surplus states. To explore these issues we focused on simple counting agents.

## 11.1 Single Agent

We performed tests on a single agent scenario to analyse how it is affected by various Brahms constructs. The tests were all based on counting to see the effect on per-

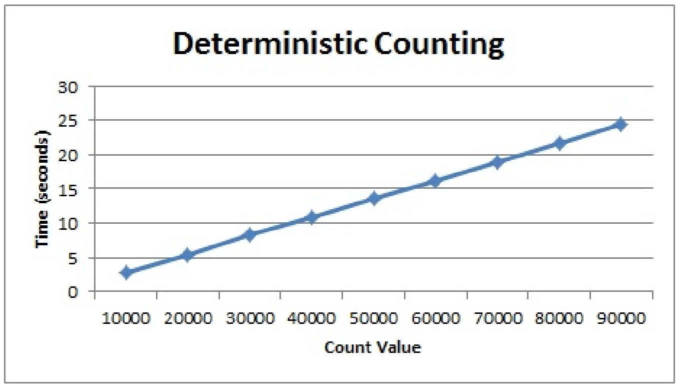


Figure 11.1: Graph showing verification time against size of count

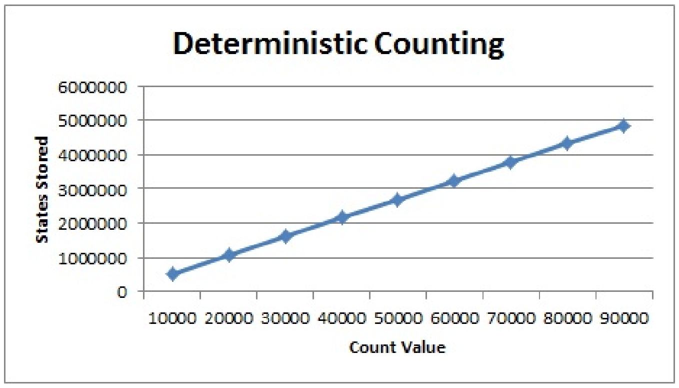


Figure 11.2: Graph showing the number of states stored against size of count

formance when: a count is increased, the number of workframes (or thoughtframes) increases, the number of activities increases and the amount of non-determinism is increased.

### 11.1.1 Deterministic Counting

The first test was intended as a bench mark to judge limitations of a simple agent example. The idea was to see what value a single agent could count to using a single workframe before all memory would be consumed during verification, and to see what the effect of increasing the count would have; the property verified was that the agent's count would never exceed the count assigned, e.g., if the count was 1,000 we would verify that it never reaches 1,001. An activity was also present in the workframe: to wait one time unit before incrementing, this was to incorporate as many semantic rules visited on each count. The graphs representing the test results can be found in Figure 11.1 showing verification time and Figure 11.2 showing the number of states stored. As the results show this is a linear increase in verification time and states stored. This was expected because for each count the agent cycles through its semantics, changing only one variable every time.

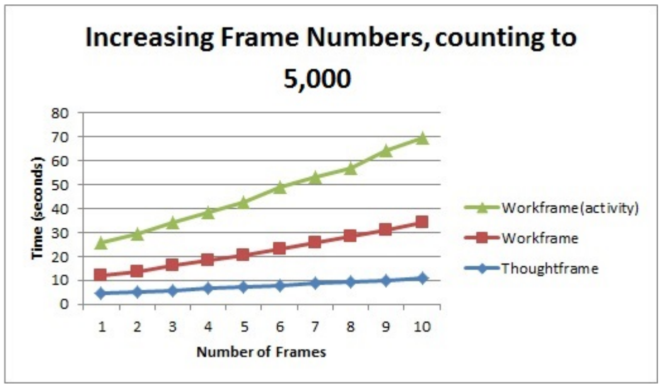


Figure 11.3: Graph showing the verification time of a count via different frame types

### 11.1.2 Frame Types and Number of Frames

The next set of tests was used to judge the effect of the frame type and the number of frames used to count to 5,000. To evaluate the effect of the frame type we verified that the agent would not count to 5,001, using either thoughtframes or workframes. The test was performed by breaking up the count and distributing it between differing numbers of frames, e.g., 1 frame with a count from 0-5,000, 2 frames with counts 0-2,500 and 2,501-5,000, and so on. The hypothesis is that: each frame would have a slight overhead, creating more states for each additional frame; workframes will have a very slightly higher overhead due to more available constructs; and adding an activity will have the largest overhead as it will move time forward causing the agent and scheduler to utilise more of their semantic rules. The results of the tests are shown in Figure 11.3 showing verification time and Figure 11.4 showing states stored. The results showed a linear increase in verification time and states stored when frame numbers increased, as expected. For a count of 5,000 there were found to be approximately 50,000 additional states per additional thoughtframe, which results in approximately 10 extra states per iteration of the semantics (calculated by 50,000 divided by 5,000). Workframes were found to be an extra 30 states per additional workframe. The results also showed that, as expected, thoughtframes performed better on verification and adding an activity to the workframe slows down the verification. However, the difference in performance is surprising, only a marginal performance difference between thoughtframes and workframes was expected yet the difference was similar to adding an activity to the workframe. On closer examination of these results it was found that *PROMELA* code using workframes was much larger than those using thoughtframes, one example being 478 lines with 4 thoughtframes against 748 for 4 workframes. The additional lines of code with the agents using workframes are to check if the workframe needs to be suspended. This could be considered an error because the workframes in these tests do not have activities, which means they will never be suspended. However, the general idea of workframes is that they will contain activities, a workframe without any activities should be represented using a thoughtframe.

### 11.1.3 Non-Deterministic Counting

The next test was to set a benchmark for adding non-determinism to Brahms models for verification. Non-determinism in Brahms can be added through various means,

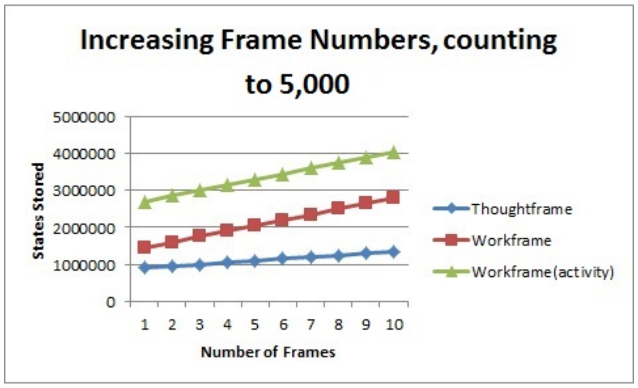


Figure 11.4: Graph showing the number of states stored when verifying a count via different frame types

however we only consider two possible methods for adding non-determinism:

1. belief and fact certainties, there are probabilities assigned to belief and fact updates stating the probability that the update will occur; and
2. thoughtframe and workframe selection, thoughtframes and workframes that are active and have equal priority will be randomly selected for execution;

The test was performed using 2 counters; one which deterministically counts up to a desired stop condition and the second counter which may or may not increase. For example, using two counters $i$ and $j$ we deterministically increase counter $i$ 800 times and every time we increase $i$ we have the option of increasing $j$, resulting in a non-deterministic value of $j$ ranging from 0-800. The results of the tests are shown in Figure 11.5 showing verification time and Figure 11.6 showing states stored. The results show an exponential increase until a count of 800 when memory has been completely consumed. This demonstrates that if the task is small then there are no problems when adding non-determinism. To compare the performance we built a simple *PROMELA* model to do the exact same process. The *PROMELA* code has two variables $i$ and $j$, both set to 0, and a do-while loop with a guard condition that $i < 800$. Inside this loop there is an increment for variable $i$ and an if-statement with two possible options: increment $j$, or do nothing. The do-while loop iterates 800 times, i.e., $i = 800$ on termination, but on each iteration $j$ may or may not be incremented resulting in a termination value varying between 0 and 800. The code is as follows:

```
/*i increases to 800 and on each count
j has an opportunity to increase */
int i = 0;
int j = 0;

active proctype proc_Environment(){
/*Loop to a count of 800*/
  do
  ::(i < 800)->    i = i+1;
  /*Non-deterministic if-statement with two
  possible options: count up or skip count*/
    if
    ::(true)->      j = j+1;
    ::(true)->      skip;
    fi;
  :: else->    break;
  od;
}
```

We performed the same tests using this simple *PROMELA* code and plotted the results onto the same graph shown in Figure 11.7. The results show that the simple *PROMELA* model is much more efficient, since the Brahms implementation has a much steeper exponential curve. This is possibly due to the *PROMELA* model generating states for the Brahms semantics, where the simple model only generates states relating to the model. Figure 11.8 shows the performance difference between these two tests, which surprisingly shows an exponential increase. This was an unexpected result since the number of branches in the model should be identical except the depth of the Brahms model should be much deeper. To help explain our reasoning consider Figure 11.9 and Figure 11.10. Figure 11.9 shows a deterministic example of counting for both a Brahms implementation and the simple *PROMELA* example. This shows that in the simple example only one state is required to increase the counter, where the Brahms implementation will require a set number of additional states in-between these two. These additional states are used to execute the Brahms operational semantics (loop counters, temporary variables, if-statements, etc.) Figure 11.10 shows what happens when branching occurs due to non-determinism; it shows that branching occurs at the same points but for the Brahms implementation there are extra states required between each branching point. Figure 11.9 and Figure 11.10 are hypothetical and use an arbitrary value of n states to represent the additional states used by loop counters, etc. in the implementation of the Brahms operational semantics. This simplified *PROMELA* example was used as a basis to show how our implementation compares to a solution which could be considered as optimal. To achieve similar verification results to this simplified example would be extremely difficult because it would require us to remove all the surplus states from the model. There is also a possibly that it will be a time cost for identifying and removing these states during verification.

## 11.2 Multiple Agents

The next set of tests are similar to the single agent tests but with an emphasis on showing the effect of adding more agents into the scenario. The agents are all synchronised using a Brahms scheduler which prevents any interleaving of the agents, giving a clearly defined order of which agent runs next. Our hypothesis was that adding additional agents should not change the verification performance much, except for an overhead on initialising variables and additional work for the scheduler.

### 11.2.1 Increasing Agents Counting 1,500

This experiment was designed to see the effect of increasing the number of agents all doing the same task. The hypothesis was that since there is no interleaving of agents then there should be a linear increase in verification time, e.g., having one

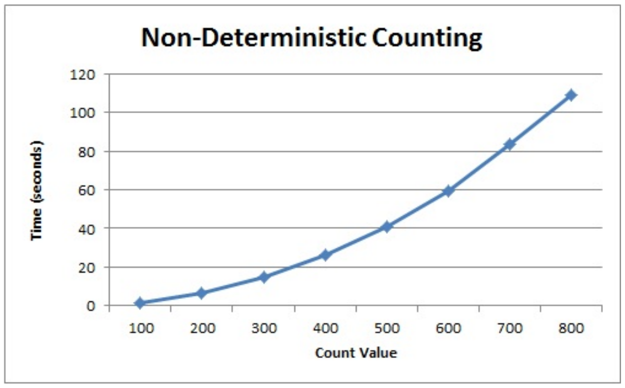


Figure 11.5: Graph showing the verification time of a single agent incrementing two counters, one deterministic and the other non-deterministic

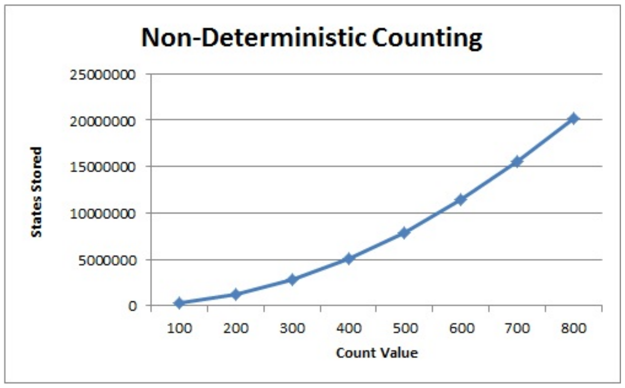


Figure 11.6: Graph showing the number of states used when verifying a single agent incrementing two counters, one deterministic and the other non-deterministic

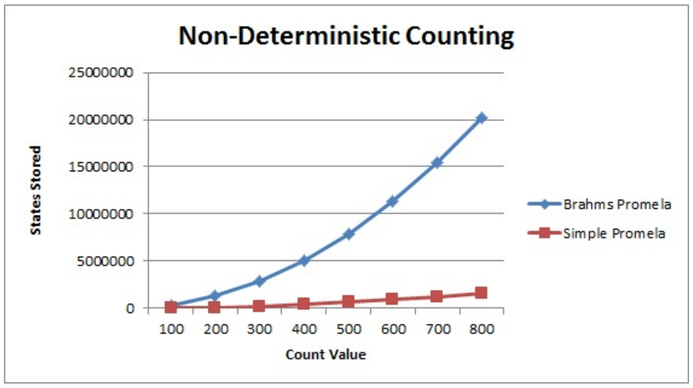


Figure 11.7: Graph showing a simple *PROMELA* representation vs the Brahms implementation, each showing the number of states used when verifying a single agent incrementing two counters, one deterministic and the other non-deterministic

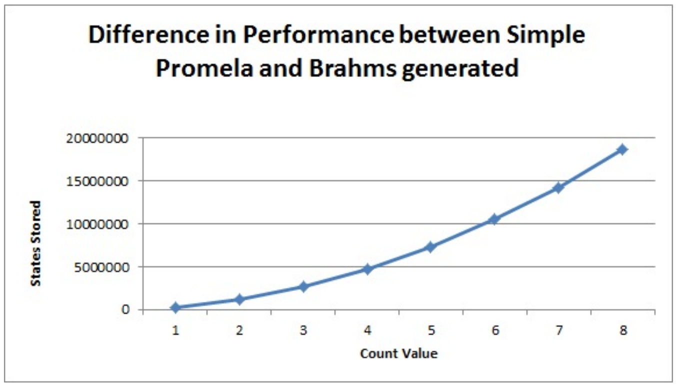


Figure 11.8: Graph showing a single line representing the difference in number of states used between a simple *PROMELA* implementation and the Brahms implementation when counting non-deterministically

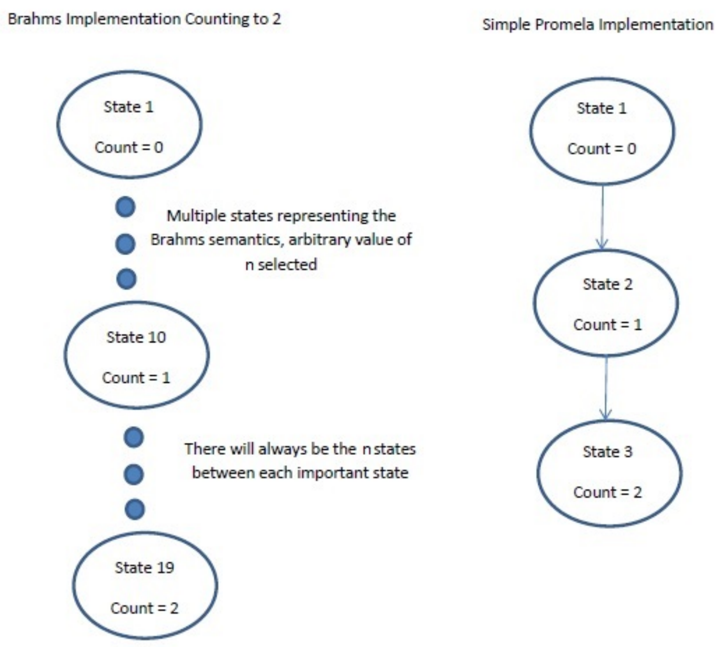

Brahms Implementation Counting to 2
Simple Promela Implementation
State 1
Count = 0
Multiple states representing the Brahms semantics, arbitrary value of n selected
State 10
Count = 1
There will always be the n states between each important state
State 19
Count = 2
State 1
Count = 0
State 2
Count = 1
State 3
Count = 2

Figure 11.9: Simple Deterministic Counting

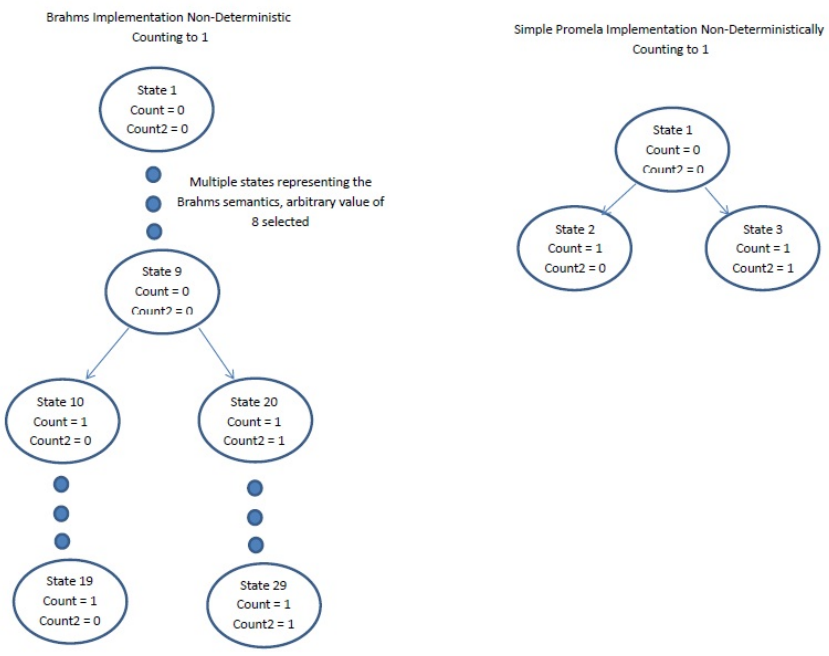

Brahms Implementation Non-Deterministic Counting to 1
State 1 Count = 0 Count2 = 0
Multiple states representing the Brahms semantics, arbitrary value of 8 selected
State 9 Count = 0 Count2 = 0
State 10 Count = 1 Count2 = 0
State 20 Count = 1 Count2 = 1
State 19 Count = 1 Count2 = 0
State 29 Count = 1 Count2 = 1
Simple Promela Implementation Non-Deterministically Counting to 1
State 1 Count = 0 Count2 = 0
State 2 Count = 1 Count2 = 0
State 3 Count = 1 Count2 = 1

Figure 11.10: Simple Non-Determinisitic Counting

agent counting to 1,500 then adding another agent to count to 1,500 should double the number of states and therefore double the verification time. However, the scheduler's work needs to be taken into account, an extra agent will produce more work for the scheduler. When the tests were performed we found that the number of states (graph found in Figure 11.12) did increase linearly but the verification time rose exponentially (graph found in Figure 11.11). The initial hypothesis to explain this result was the property being checked (such as the number of operators and depth of nesting) increased with the number of agents (i.e., Agent_1's and Agent_2's and Agent_3's count does not exceed 1,500) causing a rise in verification time. The tests were rerun but with all the tests checking the same property; that Agent_1's count did not exceed 1,500. The second test produced almost identical results as previously. Since the property had little effect on the verification time this meant the issue must lie with the amount of memory consumed by each state. Each agent has a belief about every attribute of every agent and every object, even if this belief is null. This means that the amount of memory used by each state increases with every additional agent, even if the agent is inactive for the whole simulation.

### 11.2.2 Increasing Agents sharing a Count of 10,000

This set of tests was run to verify a hypothesis arising from the results of the previous test (multiple agents counting to 1,500); the size of the state increases when more agents are added. The idea behind this test was to split a work load between multiple agents to demonstrate how this affects the state space and verification time. The tests would involve a single agent counting to 10,000, 2 agents each counting to 5,000, 3 agents counting to 3,333 and so on. The hypothesis was that the verification time and state space should actually decrease (at an inversely exponential rate) with more agents

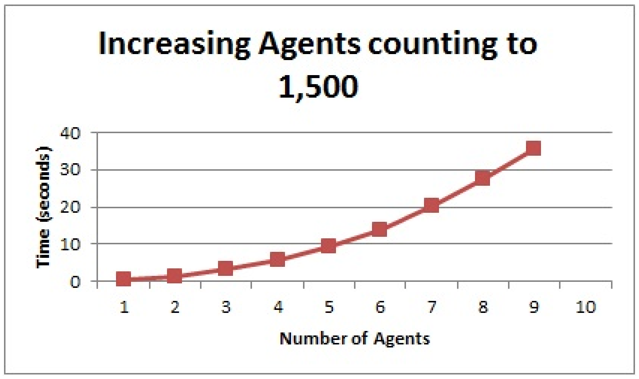


Figure 11.11: Graph showing the verification time of increasing agents counting to 1,500

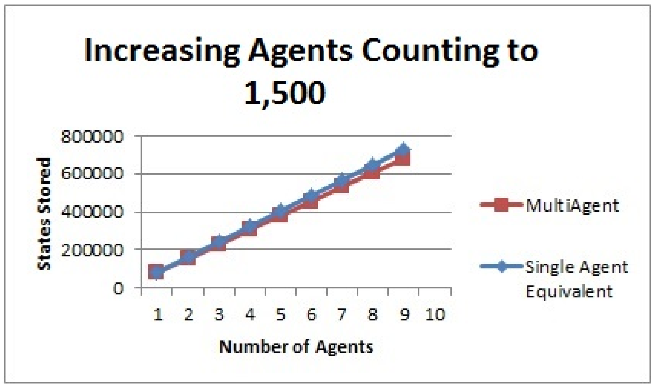


Figure 11.12: Graph showing the the number of states used when increasing agents counting to 1,500. Including a single agent representation

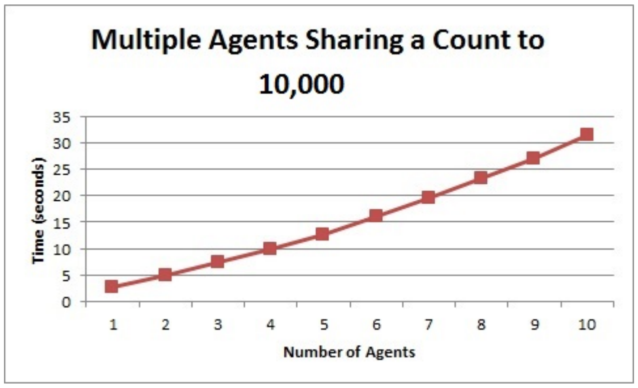


Figure 11.13: Graph showing the verification time when sharing a count to 10,000 between an increasing number of agents

added. The reasoning behind this is that 1 agent executing a cycle of the operational semantics for a Brahms model twice is computationally the same as 2 agents executing the operational semantics of the same Brahms model once. However with 1 agent the scheduler cycles through a set of operational semantic rules for the model twice and with 2 agents it cycles through these semantics once. Figure 11.15 helps explain this idea, showing the difference between the cycles of the semantics for a single agent counting to 10,000 and 2 agents counting to 5,000. With a single agent the scheduler only has to coordinate the one agent but has to iterate through its semantics 10,000 times. With two agents the scheduler needs to coordinate 2 agents but only has to iterate through the operational semantics for the model 5,000 times. The results of the tests showed the hypothesis to be correct with respect to the state space, shown in Figure 11.14 for states and Figure 11.13 for time. However, the time to verify the property rose linearly with the number of agents. This reinforces the previous hypothesis that the memory consumed by the states is slowing the verification: In the single agent example all the variables and counters of just 1 agent are stored, but with 2 agents this is doubled to accommodate the second agent. It is an exponential increase because if a third agent is added then agent 1 and 2 will need beliefs about agent 3's attributes and agent 3 will need beliefs about agent 1 and 2's attributes, also since agent 3 is identical to 1 and 2 then it will have the same attributes.

### 11.2.3 Broadcasting Count to 1,000

These next set of tests were designed to analyse the effect of communication on verification time. The test involved a single object counting to 1,000 then communicating its count to the agents. As it was previously discovered that increasing the number of agents increases the verification time we decided to keep the number of agents the same but change the number receiving the communication. Communication is carried out in Brahms by one agent changing another agent's beliefs, meaning that communication is little more than a belief update, so in theory only a few additional states would be needed. When performing this test we used two different methods for sending a single message to multiple agents: one would perform a communication for each agent, one after the other; and the other would use a *collectall* variable to decide who would receive the communication and then update the beliefs of all these agents at once. What we are essentially trying to do here is a 'broadcast', Brahms does have a broadcast

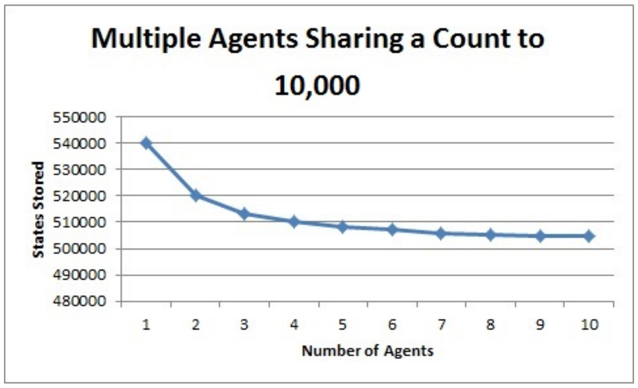


Figure 11.14: Graph showing the the number of states used when sharing a count to 10,000 between an increasing number of agents

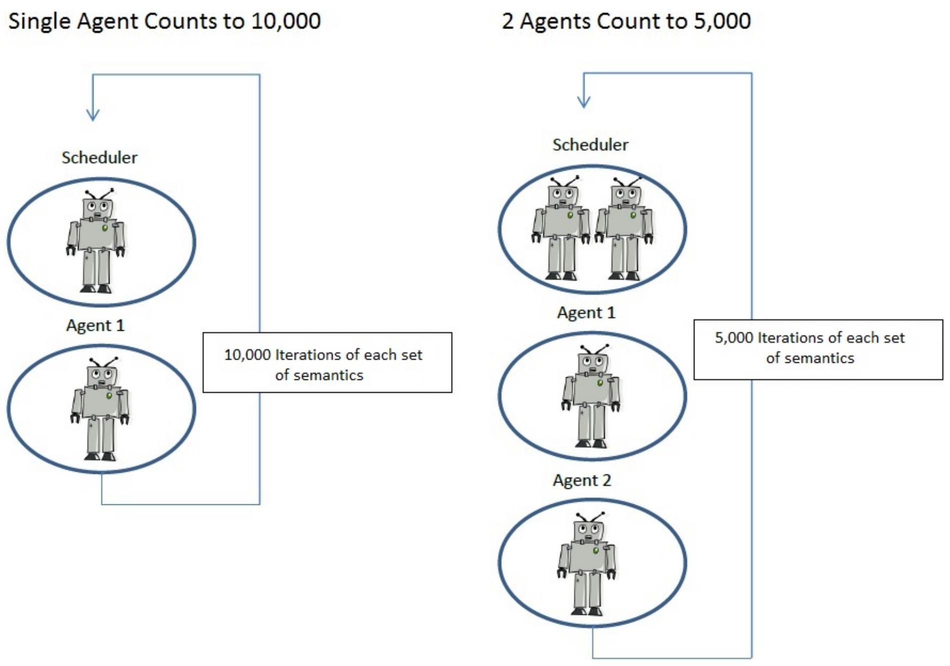

Single Agent Counts to 10,000
Scheduler
Agent 1
10,000 Iterations of each set of semantics
2 Agents Count to 5,000
Scheduler
Agent 1
Agent 2
5,000 Iterations of each set of semantics

Figure 11.15: Semantic iterations comparrison of single agent vs two agents

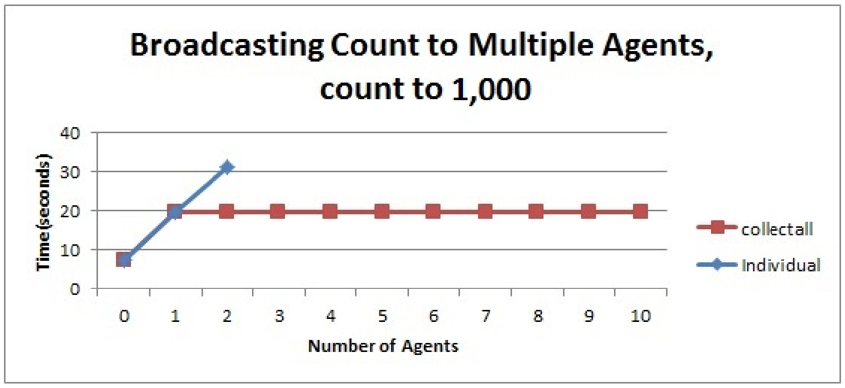


Figure 11.16: Graph showing the verification time of an object counting to 1,000 and broadcasts this count to an increasing number of agents

function but this was not implemented in the operational semantics or the *PROMELA* translation because it could be represented using other constructs. This test was of particular interest because it would demonstrate the verification performance of these two different methods. Where the broadcast using a *collectall* variable will require the use of loops and if-statements to identify the agents it is communicating to but will perform all the communications very quickly, whereas the individual communications method will bypass these loops and if-statements but need to execute the communication semantic rules multiple times. The results of the two experiments were overlapped onto the same graphs (Figure 11.16 showing the verification time and Figure 11.17 showing the states used) to show the difference in performance.

The predicted result for the *collectall* test was a high initial number states and verification time for communication to a single agent with a very slight increase for every additional agent. The individual communications prediction was a lower initial number of states and verification time with a steeper increase for every additional agent. The results for the verification time are shown in Figure 11.16 which shows that the prediction for the *collectall* method was correct. However, it was surprising to see how little adding agents to the communication affected the performance, Figure 11.17 shows that only 2 additional states are created for every agent added.

In Figure 11.16 and Figure 11.17 there are only 3 points shown on the graph for individual communications to the agents, but only 1 is visible as the first two overlap with the *collectall*. This was because the maximum amount of memory was exceeded when trying to verify the communications to more than 2 agents. This signifies an issue with communication suggesting further testing needs to be performed to ascertain the reason behind this state space explosion.

### 11.2.4 Multi-Agent Non-Deterministic Counting

The next set of tests was aimed at analysing the effect of non-determinism when the number of agents increases. Similarly to the deterministic test, non-determinism was added to the scenario with two counters, one deterministic to act as a stop condition and the other which non-deterministically increments.

The agents were asked to count to 5, the results are shown in Figure 11.18 showing time and Figure 11.19 showing the number of states. As expected, the rise in time and the number of states increases at an exponential rate until memory was exhausted after

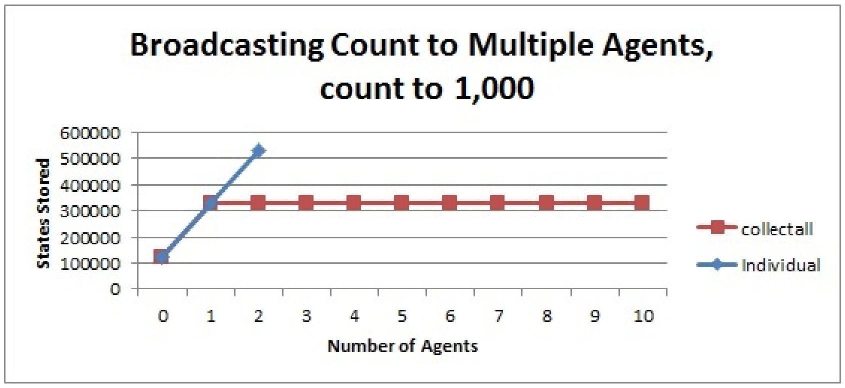


Figure 11.17: Graph showing the the number of states used when an object counts to 1,000 and broadcasts this count to an increasing number of agents

5 agents non-deterministically counting to 5.

The expected results for these tests were that the number of agents would exponentially increase the depth of the model (number of states in a trace run through the model) and the size of the count would exponentially increase the amount of branching that occurs in the model. Based on these assumptions it was decided to keep the count small, since adding more agents will also add more non-determinism, i.e., 1 agent counting to 10 would have much less non-determinism than 2 agents counting to 10. The rise in number of states was however far greater than anticipated. We identified in Figure 11.19 that if a task is distributed across multiple agents then the number of states required reduces. This posed a question of whether this would still be the case with multiple agents, e.g., if one agent counts to 800 with a non-deterministic choice at each count, could 5 agents count to 160 with the same non-deterministic choices using fewer states?

To answer this question another simple *PROMELA* example was created, based on the one used in Section 11.1.3. The simple *PROMELA* example uses 3 variables: 1 acts as a deterministic count (variable $a$) and a variable for each agent (variables $b$ and $c$) to non-deterministically increment on each count. There is a process for each agent and a process for the scheduler to synchronise the agents. The scheduler loops 6 times, each time incrementing counter $a$. On each iteration of this loop the scheduler hands control of the cpu using a variable *turn* to each agent, where they have a choice to increment their counter. Below is *PROMELA* code for a scheduler of an example model:

```
int a = 0; /*deterministic counter*/
int b = 0; /*agent 1's non-deterministic counter*/
int c = 0; /*agent 2's non-deterministic counter*/

mtype = {Environment, ag1, ag2}
mtype turn = Environment;

/*The scheduler*/
active proctype proc_Environment(){
  /*Loop through all agents*/
  do
  ::(a < 6)-> /*Loop for 5 counts, 6th for termination*/
```

```
    /*Agent 1*/
    if
    ::(turn == Environment)->
      /*if at start, start agent*/
      if
      ::(a == 0)->        run proc_agent1();
      ::else->        skip;
      fi;
      /*Give agent 1 control of cpu*/
      turn = ag1;
    fi;
    /*Agent 2*/
    if
    ::(turn == Environment)->
      /*if at start, start agent*/
      if
      ::(a == 0)->        run proc_agent2();
      ::else->       skip;
      fi;
      /*Give agent 2 control of cpu*/
      turn = ag2;
    fi;
  :: else->    break;
  od;


}

proctype proc_agent1(){
  do
  ::(a < 5)->
    if
/*Increment counter*/
    ::(turn == ag1)->      b = b+1;
/*Do not increment counter*/
    ::(turn == ag1)->      skip;
    fi;
    /*hand control back to environment*/
    turn = Environment;
  :: else->    break;
od;
}

proctype proc_agent2(){
  do
  ::(a < 5)->
    if
/*Increment counter*/
    ::(turn == ag2)->      c = c+1;
```

```
/*Do not increment counter*/
    ::(turn == ag2)->      skip;
    fi;
    /*hand control back to environment*/
    turn = Environment;
  :: else->    break;
od;
}
```

Through using this simplified version of the Brahms *PROMELA* implementation we were able to test if it was a problem with the Brahms implementation or whether *PROMELA* had state reduction techniques for handling large non-determinism on single variables. The results of this *PROMELA* test was that the memory was still being exhausted for small counts, e.g., counting to 5. To further test this hypothesis another simpler *PROMELA* example was created that did not reflect the Brahms semantics but performed the same task. This simplified version compressed the agents and scheduler into a single process. This was done by using a do-while loop to increment the deterministic counter with nested if-statements to represent the agents' choice to increment their counters. The code is as follows:

```
int a = 0; /*deterministic counter*/
int b = 0; /*agent 1's non-deterministic counter*/
int c = 0; /*agent 2's non-deterministic counter*/

active proctype proc_Environment(){
  do
  ::(a < 10)->
    /*Agent 1*/
    if
/*Increment counter*/
    ::(true)->      b = b+1;
/*Do not increment counter*/
    ::(true)->      skip;
    fi;
    /*Agent 2*/
    if
/*Increment Counter*/
    ::(true)->      c = c+1;
/*Do not increment counter*/
    ::(true)->      skip;
    fi;
    /*Increase deterministic counter*/
    a = a+1;
  :: else->    break;
od;
}
```

This simplified version of the multi-agent non-deterministic count showed that 5 agents could count to 10 before all available memory would be, compared to 5 for the previous version. Overall the results of these tests show that the performance of the Brahms

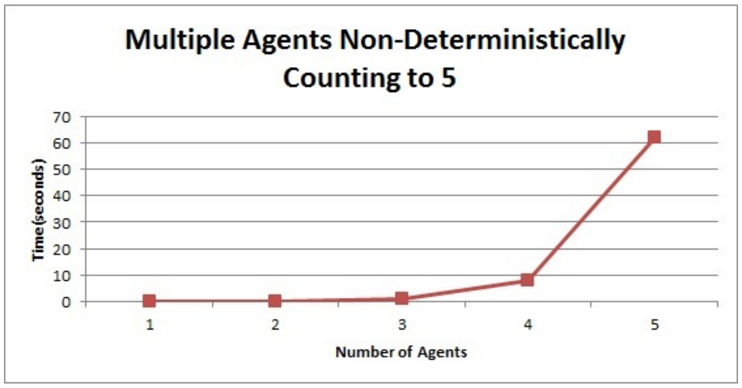


Figure 11.18: Graph showing the verification time of an increasing number of agents non-deterministically counting to 5

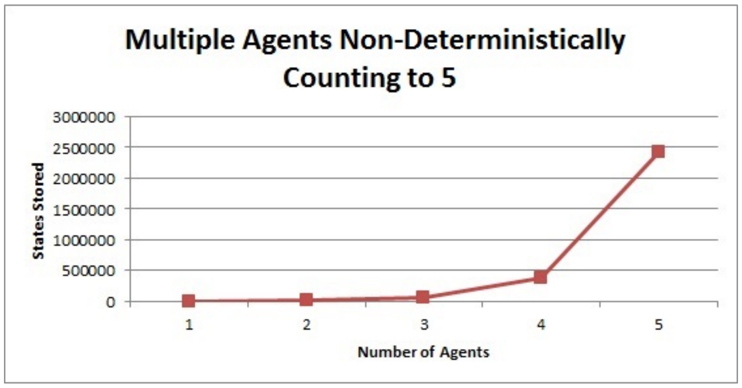


Figure 11.19: Graph showing the number of states used when an increasing number of agents non-deterministically count to 5

implementation is not as poor as first thought but that restructuring the agents into a single process (which is possible) could provide a much needed performance boost.

### 11.2.5 Performance Conclusion

The performance of the *PROMELA* implementation of the Brahms semantics is affected by many factors which increase the number of states produced and therefore the verification time. The main issues identified were that adding agents, adding frames, workframes with no activities, communication and how many activities/concludes the agent has increases the number of states.

We identified that adding additional agents increased the size of the states thus extending verification time. Interestingly, adding inactive agents did not increase the number of states as significantly as expected. This is believed to be because most of the agents' semantic rules are bypassed when inactive; only rules requiring the agent to check for active thoughtframes and workframes are processed.

The addition of workframes and thoughtframes of an agent were found to have an approximate increase of 30 and 15 (respectively) additional states per single iteration of the semantics. The reason why so many additional states are created is unclear especially since the check on whether a frame is active or not is placed in a deterministic wrapper (a function which compresses multiple states into a single state, removing any non-determinism). A possible reason for these states is the increased length of the arrays to hold the frames; these arrays are checked from head to tail to find the frame matching an identification number to set it as active or inactive. However, this array search is also enclosed in a deterministic wrapper. The only other additional code created is from the belief updates contained within the frames. These belief updates are not contained within any deterministic wrappers however only one of the belief updates will be available for selection (the one matching the identification number retrieved from the frame's stack of deeds). One issue that was noticed was the number of deterministic wrappers used, since each one wraps a number of states into single state it would mean multiple wrappers still mean multiple states. Therefore a re-organisation of the code to reduce the number of wrappers used would reduce the number of states created. It was also noticed that the wrappers used did not fully enclose all aspects of the code, aspects such as print lines (for debugging purposes) were omitted and some simple counters. Because of this it would be beneficial to make these small adjustments to the deterministic wrappers and note the effect on the performance. If there is a marked increase in performance then justification could be made to restructure the implementation to reduce the number of deterministic wrappers used and thereby improve performance.

A performance issue was identified with the use of workframes making workframes far more expensive than thoughtframes. A possible reason for this was executing a semantic rule to check whether or not to suspend the current workframe. This check needs to be performed but it was found that this check was possibly being unnecessarily performed, notably when a workframe had no activities.

Communication provided the worst and most unexpected performance result, especially with the individual communication to multiple agents. The use of the *collectall* was expected to give a rise in number of the states, however when a single communication is made the *collectall* only required 2 more states than the individual communication. However the addition of a communication (from 0 communications to 1) in both cases more than doubled the number of states used (124,271 to 330,272 for the communication without a *collectall*). Since this is a simple belief update then such an increase should not happen. Future work will need to be conducted to identify how to fix this issue.

The addition of non-determinism to the tests proved to have surprisingly better performance than expected with a single agent but with poorer performance with multiple agents. The single agent model managed a count to 800 before running into memory issues whereas the tests with 5 agents counting to 5 pushed the memory limits. Additional tests were run on simplified *PROMELA* models to help understand this; they suggested that this outcome should have been predicted. However, these tests also highlighted that implementing the Brahms semantics in a single process could improve the performance when adding non-determinism. Implementing the Brahms semantics in a single process would also remove blocking statements (where an agent's process is halted until another agent has finished) and allow easier use of deterministic wrappers.

## 11.3 Modified Brahms Implementation

As a result of the performance testing it was identified that the *PROMELA* implementation of the Brahms semantics could be improved by adding more deterministic

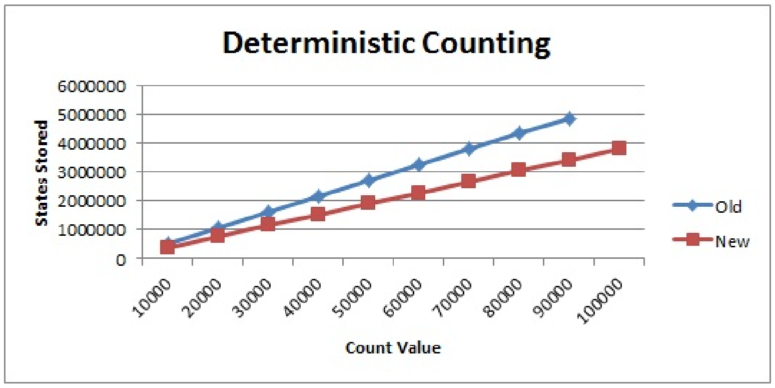


Figure 11.20: Graph showing the number of states stored when verifying an increasing deterministic count

wrappers. The performance testing highlighted that Spin generates a state for every line of code that it encounters; including print lines. This led to some small modifications to extend the wrappers to include every line of code they could encapsulate and add wrappers for others. The performance testing was then performed once again using exactly the same Brahms models to demonstrate what kind of performance increase would be achieved from adding some extra deterministic wrappers. More importantly, any increase in performance achieved would justify future work to alter the implementation to accommodate and condense the use of deterministic wrappers.

The sections below will explain the tests performed before discussing the results. For simplicity we only consider the number of states used for verification to analyse the performance difference. The graphs will show both the performance of the previous implementation of Brahms semantics (labelled as Old) and the updated implementation (labelled as New).

## 11.4 Single Agent

The single agent tests were to analyse the performance of Brahms functions. These individual tests show how the performance has been affected for individual Brahms functions as some functions may have more deterministic wrappers added than others.

### 11.4.1 Deterministic Counting

This test was to see what value a single agent could count to using workframes with a primitive activity of 1 time unit to ensure full cycling of the semantics. The results can be seen in Figure 11.20. The new implementation shows a marked improvement with a growth rate of 380,000 states per additional count of 10,000 compared to 540,000 of the previous implementation, approximately 30% decrease in the number of states.

### 11.4.2 Frame Types and Number of Frames

In this test a count was broken down into sections, for a single frame the whole count was represented in a single frame but for 2 frames the count would be split into 2 frames (one counting from 0 to 2,500, the other counting from 2,500 to 5,000). This test was performed 3 times, once for each of the following: workframes, thoughtframes

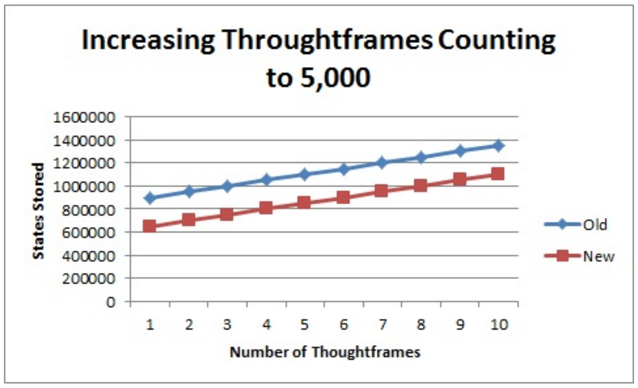


Figure 11.21: Graph showing the number of states stored when verifying a count distributed among different thoughtframes

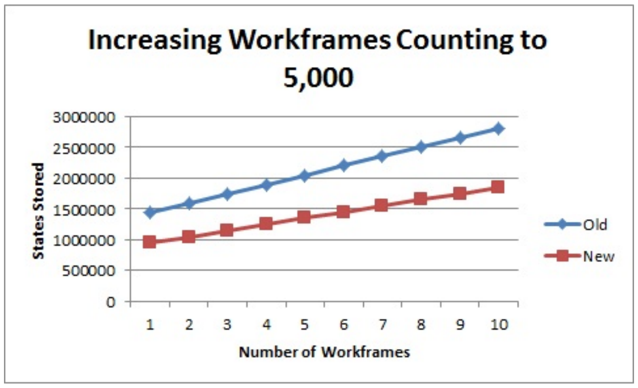


Figure 11.22: Graph showing the number of states stored when verifying a count distributed among different workframes

and workframes with an activity. A graph was used to display each test with a comparison against the previous implementation: thoughtframes in Figure 11.21; workframes in Figure 11.22; and workframes with an activity in Figure 11.23. The results shows an improvement on all frame types, however the number of states required per thoughtframe does not change so both lines on the graph are parallel to each other. There is an initial performance improvement but every time another thoughtframe is added both the old and new implementation need an additional 50,000 states. It can be noted that 50,000 additional states over a count of 5,000 mean an addition of 10 states per thoughtframe per count (50,000/5,000). This implies that most aspects which can be in a deterministic wrapper are covered. Workframes on the other hand showed a real performance improvement; requiring less on single workframes and showing a smaller state increase when new workframes are added. For workframes (both with and without an activity) the number of states increase at a rate of 150,000 per additional workframe compared with the new implementation's 100,000; approximately a 33% improvement. The workframes also have an initial improvement when using a single workframe. This improvement is due to workframes requiring additional code for suspending the workframe, processing detectables and processing activities. These additional lines of code provide more possible applications of deterministic wrappers.

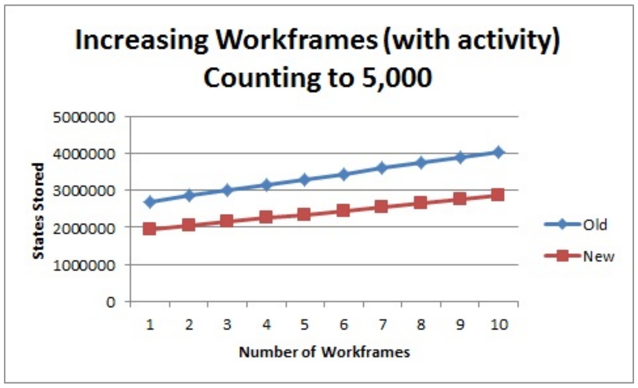


Figure 11.23: Graph showing the number of states stored when verifying a count distributed among different workframes with an activity

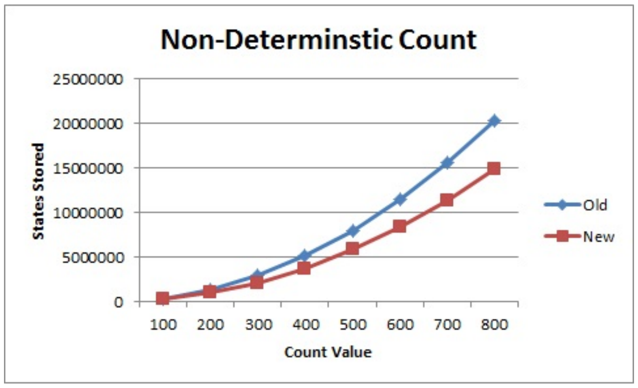


Figure 11.24: Graph showing the number of states stored when verifying an increasing non-deterministic count

### 11.4.3 Non-Deterministic Counting

To test non-determinism 2 counters were used; one which would deterministically count setting a desired stop-condition and a second counter which may or may not increase with the first counter. This test also showed a performance improvement for the new implementation. This improvement is harder to quantify since the graphs are exponential, however the new implementation required approximately 27% less states than the old implementation. The results are shown in Figure 11.24.

## 11.5 Multiple Agents

The following sets of tests will show the change in performance with the addition of multiple agents. There has however already been a significant performance increase when using Brahms functions in single agent scenarios, taking this into account we will analyse the performance in comparison to the already known increases in performance.

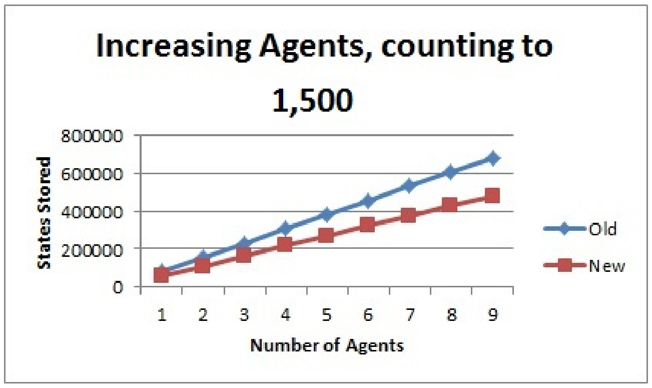


Figure 11.25: Graph showing the number of states stored with increasing agents counting to 1,500

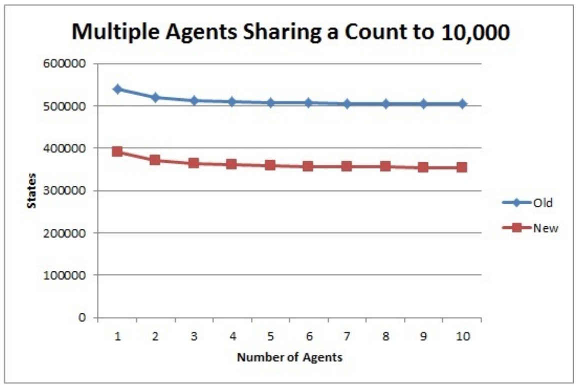


Figure 11.26: Graph showing the number of states stored with a count to 10,000 split between an increasing numbers of agents

### 11.5.1 Increasing Agents Counting to 1,500

This test was designed to analyse the effect of multiple agents performing the same task, i.e., counting to 1,500. The test results are shown in Figure 11.25. The results show a 30% reduction in states as compared to the previous implementation, considering the single agent version, in Figure 11.20, also had a 30% reduction this indicates that there is no additional performance increase (or decrease) from adding new agents.

### 11.5.2 Increasing Agents sharing a Count of 10,000

This test was to split a work load between multiple agents to demonstrate how this affects the state space. This test also showed a 30% increase in performance, which was to be expected given the previous test. The results are shown in Figure 11.26

### 11.5.3 Broadcasting Count to 1,000

This test is a single object counting to 1,000 and communicating its count to an increasing number of agents. There were two versions of this test; one for individual communications and another for using a *collectall* variable to simulate a broadcast.

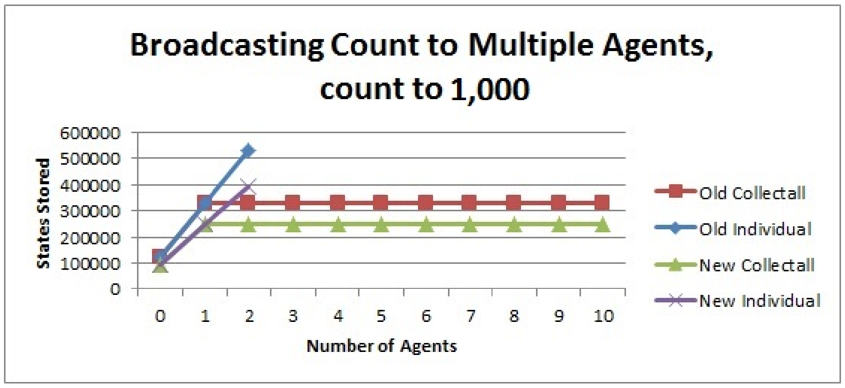


Figure 11.27: Graph showing the number of states stored when an object communicates its count to 1,000 to an increasing number of agents

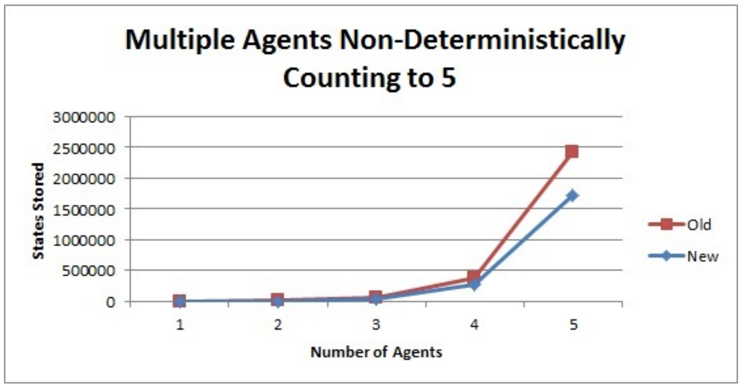


Figure 11.28: Graph showing the number of states stored when an increasing number of agents non-deterministically count to 5

This could not be tested in a single agent environment for obvious reasons, so only a comparison to the previous implementation can be made. The results are all overlaid onto a single graph: old, new, individual and *collectall*. The results (Figure 11.27) show a performance increase but still show that individual communications are an issue with memory becoming an issue after only 2 communications. This indicates that deterministic wrappers will not solve this issue. Overall the communication shows a 25% state reduction; slightly less than other Brahms functions.

### 11.5.4 Multi-Agent Non-Deterministic Counting

This experiment tested non-determinism with multiple agents. An increasing number of agents was used with two counters: one deterministic to act as a stop condition (counting to 5) and the other which non-deterministically increments with the first counter. The results of this test shows a performance increase of approximately 28%, shown in Figure 11.28, which is consistent with the single agent example showing a 27% improvement.

## 11.6 Performance Conclusion

The addition of more deterministic wrappers has significantly increased the performance with state reductions varying from 25-33%. This change was a very minor change, simply compressing some print line states and counter states into a single state. This demonstrates that a serious performance gain can be made by maximising the use of deterministic wrappers within the implementation, thereby justifying further work to break down the implementation and re-implement it so that deterministic wrappers can be more easily used. To further analyse the improvement to the verification we verified the properties of the Digital Nurse and Home Helper scenarios to analyse the performance difference.

Table 11.1: Home Helper Verification

| Property | Old Version(states) | New Version(states) | Percentage |
|---|---|---|---|
| F1 | 70938 | 70483 | 0.65% |
| F2 | 69794 | 69980 | -0.02% |
| F3 | 78641 | 78650 | -0.01% |
| T1 | 33955 | 34122 | -0.05% |
| T2 | 69954 | 69927 | 0.01% |
| H1 | 137545 | 137176 | 0.03% |
| H2 | 137545 | 137176 | 0.03% |
| M1 | 80333 | 80189 | 0.02% |
| M2 | 70938 | 70483 | 0.07% |
| M3 | 69308 | 69281 | 0.01% |

Table 11.2: Digital Nurse Verification

| Property | Old Version(states) | New Version(states) | Percentage |
|---|---|---|---|
| P1 | 103618 | 85644 | 17.3% |
| P2 | 230864 | 172190 | 25.4% |
| P3 | 230864 | 172190 | 25.4% |
| N1 | 101594 | 75905 | 25.2% |
| N2 | 230864 | 172190 | 25.4% |
| N3 | 230864 | 172190 | 25.4% |
| R1 | 230864 | 172190 | 25.4% |
| R2 | 230864 | 172190 | 25.4% |
| D1 | 103618 | 85644 | 17.3% |
| D2 | 230864 | 172190 | 25.4% |
| D3 | 100062 | 75575 | 24.4% |

The results show a marked improvement for the Digital Nurse, shown in Table 11.2 with approximately +25% state reduction which is comparable to the results of the tests performed on the counting agents. However, for the Home Helper scenario there was only a slight improvement of less than 1%, with F2, F3, T1 showing very minor increases in the number of states stored; results are shown in Table 11.1.

# Chapter 12

# Impact on Computer Science

In this chapter we discuss how the work produced from this thesis has aided others in their research. To remind the reader of the contributions this thesis makes to computer science:

- the first formal operational semantics for Brahms
- the first formal verification tool specifically for human-agent teamwork
- the first formal verification tool for the Brahms simulation framework
- aided construction of another formal verification tool for the Brahms simulation framework

The formal operational semantics we produced for Brahms is the first of its kind, these semantics will allow an insight into the workings of Brahms for any other researcher who wishes to develop their own verification techniques or tools for Brahms. It also gives prospective Brahms users an idea of how a Brahms simulation is processed, which may help them to decide whether or not Brahms is the system they need for their research.

The verification tool we present in this thesis doubles up as two contributions; one for verifying human-agent teamwork and the other for verifying Brahms models. This is the first tool of its kind for both human-agent teamwork and Brahms. Using this tool other researchers will be able to verify simulations involving humans and agents, and other Brahms simulations. This tool also operates as a prototype, giving ideas and insight into how the verification of Brahms can be performed and also details possible problems a researcher may encounter.

The final contribution bundles all these contributions together and presents a real example of where our work has contributed to computer science. This real example is another model checking tool developed at NASA to verify Brahms models. To create their tool NASA used our operational semantics for Brahms and requested our expertise on how to implement the semantics in Java. To aid them in this implementation we produced a simplified implementation of the operational semantics of Brahms and presented them to NASA, this code can be found in Appendix E. A copy of our tool was presented to NASA, the issue where *PROMELA* generates states for changes in counters, variable, etc. that arent needed for verification was explained. The process used by NASA as a result took great care in preventing the creation of these unnecessary states. The model checker they created is explained below.

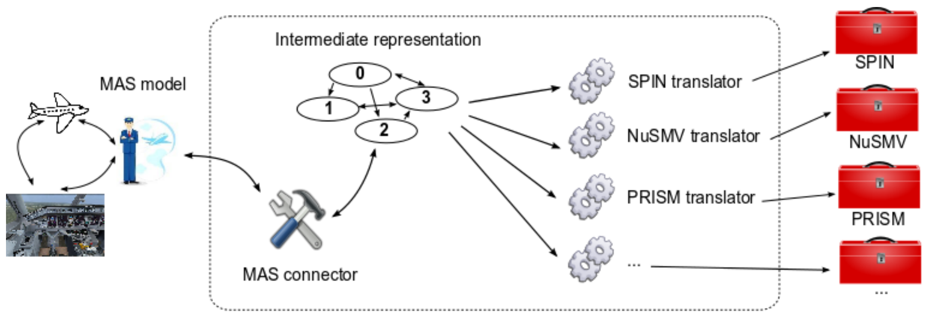


Figure 12.1: An extensible architecture that leverages state of the art technologies to verify MAS models.

## 12.1 NASA's Translation via Java Pathfinder to Multiple Model Checkers

With our collaboration Neha Rungta and Franco Raimondi developed a different tool for verifying Brahms models using the semantics we present in this thesis in Chapter 6. Figure 12.1 presents a high level overview of this framework. The input to the framework is a Brahms model representing a simulation of the desired MAS (Multi-Agent System). The *MAS connector* shown in 12.1 executes the Brahms semantics and generates an intermediate representation of the MAS model described in the inputted Brahms code. This model holds all the relevant states and transitions of all the agents in the MAS, discarding anything which is not part of the model e.g., counters and variables used when executing the Brahms semantics. This was done by extending the Java Pathfinder model checker to gain control of the execution of the Brahms semantics. Extensible plugins in JPF, such as customised choice points, allowed the efficient reduction of the state space producing the model in the intermediate representation. This intermediate representation is essentially an explicit state model representing all the possible states and actions of the MAS. Additionally the *MAS connector* gathers and stores information such as transition probabilities, temporal and epistemic relations between states. This allows for additional search and exploration strategies for verification purposes which are re-useable for different verifiers such as probabilistic and on-the-fly safety properties.

Currently the intermediate representation can be converted into formats for mainstream verification tools such as Spin, NuSMV and PRISM, allowing verification of LTL/CTL properties, probabilities, time bounds and cost.

### 12.1.1 Case study: Air France 447 Model

The following is Neha Rungta's [71] description of the case study to demonstrate the efficacy of this technique:

> On June 1, 2009 Air France Flight 447 between Rio de Janeiro and Paris crashed in the equatorial Atlantic. The final BEA accident report[1] states

[1]`http://www.bea.aero/en/enquetes/`

that the pitot tubes sensors used to detect the airspeed of the aircraft were reporting incorrect airspeed values. The weather conditions caused icing to build up on the pitot tubes resulting in inaccurate airspeed readings. The inexperience of the pilot was determined to be the cause of the crash. The pilot in charge misjudged the airspeed of the plane and increased the altitude of the plane without realizing the plane was in a stall which eventually led to its crash. According to the report the pilot was presented with several chances to recover, but, was unable to do so.

A model of the conditions during the flight of AirFrance 447 was developed to validate that the conditions or hardware failures did not lead to the crash[2]. This flight model only includes the important scenarios (i.e. those that have been determined as catalysts in the cause of the crash). In the model, there are several components that interact with one another, namely, the pilot, the controls, the airplane itself, two pitot tube sensors, the weather, and the stall level. The pilot uses the controls (throttle, elevators) to manipulate the speed, altitude and attitude of the plane. In the model, the pilot relies on the airspeed reading, which is provided by the pitot sensors. Pitot sensors monitor the speed of the plane and relay that information back to the pilot through the gauges. The `Weather` object in the model that can simulate stormy conditions and that results in ice to form over the pitot tubes. The model assumes that when the icing over the of the Pitot tube's sensors exceeds a certain threshold the airspeed readings become inaccurate. If the pilot notices that the airspeed readings from each of the two pitot sensors do not match, the pilot attempts to estimate the airspeed using measures described in aviation procedures. In the model the pilot can use other values to determine that the airspeed values reported by the pitot sensors is incorrect and then tries to guess the correct speed. The stall level may increase depending on the combination of airspeed, altitude, and attitude. Once the stall level goes above some threshold, through different mechanisms the pilot becomes aware of the situation and can adjust the controls accordingly.

**Verification Results**

The results showed that the model checker was able to verify that hardware failure was not at fault and that the pilot had plenty of opportunities to correct the stall. JPF took 2.5 minutes to generate approximately $28,000$ to prove the property (which was not made available to us) representing the above statement.

### 12.1.2 Comparing results

A comparison was made to analyse the performance between the JPF with an *intermediate representation* and the direct translation to *PROMELA* presented in this thesis. The comparison was on the model of the home helper shown in Chapter 8 which models an elderly person, a helper robot, a care provider who is human, and another automated agent. Ten different properties were verified using both frameworks in order to

`flight.af.447/flight.af.447.php`

[2]Statistical Analysis of Flight Procedures, by Adrian Agogino and Guillaume Brat at NASA Ames Research Center, CA.

empirically compare the effectiveness of each approach.

The JPF framework has two tasks to perform; generate the *intermediate representation* and perform the verification. It takes less than one minute to generate the intermediate model, with 511 intermediate states and 690 transitions. The verification of the 10 properties then only takes 2 seconds, because the model is so refined. Giving an overall result of:

```
elapsed time:       00:00:54
JPF states:         7792
search depth:       maxDepth=1173
bytecode executed   77169747
max memory:         110MB
loaded code:        classes=168, methods=2054
```

The direct translation to *PROMELA*, the technique noted in this thesis, does not have an *intermediate representation* so Spin is required to build the model itself. This allows for surplus states required for execution of the Brahms semantics. Overall it takes approx 5 minutes to verify all 10 properties using this technique, about 5 times slower than the JPF technique. An example result for verifying a single property is:

```
State-vector:        18144 byte
depth reached:       126049
states, stored:      137545
states, matched:     11574
transitions
(= stored+matched):  149119
total actual
memory usage:        1010.894 Megabytes
```

It is interesting to note that in the JPF verification over 77 million bytecode instructions are executed while 168 classes and over 2000 methods are analysed. This demonstrates that the program analysed is of a significant size. However, a mere 7792 JPF states are generated and only 511 intermediate states are produced as output. This demonstrates how much refinement is done to eliminate surplus states which would only be used to calculate what values the next state would have i.e., the execution of the Brahms semantics. The Spin verification is then extremely fast because it does not need to generate a model since it has already been inputted into it.

The direct translation to *PROMELA* takes longer due to the structure of the example and semantics. It is worth noting however that the direct translation to *PROMELA* was a prototype to the JPF verification framework. The identification of the requirement of these surplus states during the model checking process inspired the JPF framework to create this *intermediate representation* and thereby increase the efficiency of the model checking.

# Chapter 13

# Conclusion

The work in this thesis is directed towards the verification of human-agent teamwork using the Spin model checker and the Brahms multi-agent environment. Brahms enables the description of human-agent teamwork scenarios where the defining factors are the actions taken, their timing, duration and results. It has proven useful in the analysis of such scenarios via simulation. By adding verification to Brahms we hope to extend its usefulness by allowing all possible simulations to be explored, thus ensuring that undesirable outcomes cannot arise within the model.

In this thesis we have presented the first formal operational semantics for the Brahms framework and the first tool for the formal verification of Brahms models involving human-agent teamwork. The formal semantics we produced provides us with a route towards the formal verification of Brahms applications. Using these operational semantics we can devise model checking procedures and can either invoke standard model checkers, such as Spin [40] or agent model checkers such as AJPF [7]. Using the operational semantics we were able to identify the core data structures of Brahms and develop a parser to parse a model into Java data structures. Parsing a Brahms model into Java data structures allowed for easier implementation of the Brahms semantics in the input languages of various model checkers. We implemented the Brahms semantics in the input language for the Spin model checker, *PROMELA*, for Spin verification.

Two scenarios were developed to demonstrate and analyse the verification produced by our tool. The first scenario produced was a home helper robot to aid a person with dementia. The scenario involved a robot performing duties such as fetching food and medication, an intelligent house to monitor the person and a care worker who would assist in situations the robots were incapable of handling. This was the simplest scenario but still demonstrates much of the semantics of Brahms, including the most important aspects: selection of workframes and thoughtframes; suspension of workframes when a more important (higher priority) workframe becomes active; detection of facts; performance of concludes', primitive activities, move' activities and communication activities; and the use of the scheduler. We verified properties of this scenario, such as: if fire alarm has been going for $> T$ seconds and the person has not yet left the house, then the robot informs the person to leave.

The second scenario relates to a hospital, this involved: 2 nurses, 5 patients, a doctor, a robot to monitor the patient's, a helper robot and digital assistants for the nurses and doctor. One nurse has the role of looking after the patients, turning them, etc. The other nurse performs 'other duties' not relevant to the simulation but is there to cover the nurse looking after the patients during that nurse's break. The doctor is there for emergencies and the prescription of medication. The helper robot is there to aid the nurse; fetching medication and turning patients, etc. The digital assistants are for reminding/informing the nurses and the doctor of their most current duties and also act as autonomous communication devices to keep everyone up to date on the patients. This scenario was more complex than the home helper scenario demonstrating the same key features of the Brahms semantics but also required the use of Brahms variables. The main difference with this scenario was the larger number of agents and the level of teamwork; agents and humans actually perform tasks together. An example of a property verified for this scenario is: The patients don't wait more than 1 hour when they need to be turned.

To further evaluate our tool we carried out performance evaluating tests to measure the efficiency of the verification. These tests were not based on human-agent teamwork but on the use of Brahms functions. The tests performed were very simple and based on agents counting. The tests involved measuring the effect of increasing the count, increasing the number of agents counting, adding non-determinism to the count, communicating the counts and dividing the count between multiple agents. The tests revealed some issues such as communication adding states exponentially and that more deterministic wrappers could be added to improve efficiency. After the performance testing was conducted a new slightly improved version was created which had better use of deterministic wrappers, this new version was then ran through the same performance tests and showed a 30% reduction in the number of states produced. Verification of the hospital and home helper scenarios was performed using the newer version and showed an improvement of approximately 25% state reduction for the hospital scenario and 4% for the homer helper.

The work performed in this thesis contributed to the development of another verification tool developed by Neha Rungta and Franco Raimondi. With our help Neha and Franco used the semantics produced for Brahms in this thesis to develop a Java implementation of the Brahms semantics. With this Java instantiation of the semantics they are able to use Java Pathfinder to create a refined model of the Brahms simulation, this refined model is then output into the input language of a model checker, such as Spin, for verification. Generating the refined model using Java Pathfinder proves to be the most time expensive part of the verification, but once the model is generated the verification is quick and efficient.

## Future Work

The tool created in this thesis for the verification of human-agent teamwork in Brahms models was the first of its kind, a prototype. As with all prototypes there are always areas for improvement, such as increased efficiency and functionality. The tool we created in this thesis is no different, verification efficiency can be improved on and more Brahms functions could be added to the *PROMELA* translation. The possibility of increasing the efficiency of our tool was highlighted in the performance evaluation section which showed areas where we could improve the tool. Such ways of increasing efficiency included:

1. restructuring the Brahms translation into a single *PROMELA* process to allow for better use of deterministic wrappers

2. identifying the communication issue which resulted in an exponential rise in states when performing multiple communications

3. structuring the Brahms translation for efficient use of deterministic wrappers to limit the number of surplus states

Where functionality is concerned the translation to Brahms could be expanded to include Brahms functions such as group activities and possibly to handle activities with durations within a minimum and maximum range (where currently only a single value is allowed). Functionality of the tool itself could also be improved as currently it is only executable from the command line. Spin never-claims (used to express specifications) currently need to be created manually requiring expertise and knowledge of the translation. The implementation of a graphical interface which can aid the user in generating the never-claims of the user's specification would make the system much easier to use. The graphical interface could also be integrated into the Brahms Composer (the Brahms graphical interface) giving the user the option to either generate a random Brahms simulation or verify whether a property holds or not. By adding verification to the Brahms Composer we would provide easy access to verification, which will allow all possible simulations (with fixed time granularities) to be explored, thus ensuring that undesirable outcomes cannot arise within the model.

# Appendix A

# Brahms Syntax

This is the syntax of Brahms, which was referred to in Chapter 7. It is represented in Backus-Naur Form (BNF).

## Identifiers

```
name    ::= [ letter ][ letter | digit | - ]*
letter ::= a | b || z | A | B || Z | _
digit ::= 0 | 1 || 9
blank-character ::=   | \t | \n | \f | \r
number ::= [ integer | long | double ]
integer ::= { + | - } unsigned
long ::= { + | - } unsigned { l | L }
unsigned ::= [ digit ]+
double ::= [ integer.unsigned ]
truth-value ::= true | false | unknown
literal-string ::= " [ letter | digit | - | : | ; | . ] "
literal-symbol ::= name
```

## Compilation Unit

```
compilation-unit ::=
    [ PCK.package-declaration ]*
    [ IMP.import-declaration ]*
    [ GRP.group |
    AGT.agent |
    CLS.class |
    OBJ.object |
    COC.conceptual-class |
    COB.conceptual-object |
    ADF.areadef |
    ARE.area |
    PAT.path ]*
```

## Package Declaration

```
package-declaration ::= package package-name ;
package-name ::= ID.name |
package-name . ID.name
```

## Import Declaration

```
import-declaration ::= [ brahms-import-declaration
    | java-import-declaration ]
brahms-import-declaration ::= [ import brahms-single-type-import ;
    | import brahms-multi-type-import ; ]
brahms-single-type-import ::= concept-name |
    PCK.package-name . concept-name
concept-name ::= ID.name
brahms-multi-type-import ::= * |
    PCK.package-name . *
java-import-declaration ::= [ jimport java-single-type-import ;
    | jimport java-type-import-on-demand ; ]
java-single-type-import ::= [ ID.name
    | PCK.package-name . ID.name ]
java-type-import-on-demand ::= PCK.package-name . *
```

## Group

```
group ::= group group-name { group-membership }
{
    { display : ID.literal-string ; }
    { cost : ID.number ; }
    { time_unit : ID.number ; }
    { icon : ID.literal-string ; }
    { attributes }
    { relations }
    { initial-beliefs }
    { initial-facts }
    { activities }
    { workframes }
    { thoughtframes }
}
group-name ::= ID.name
group-membership ::= memberof group-name [ , group-name ]*
attributes ::= attributes : [ ATT.attribute ]*
relations ::= relations : [ REL.relation ]*
initial-beliefs ::= initial_beliefs : [ BEL.initial-belief ]*
initial-facts ::= initial_facts : [ FCT.initial-fact ]*
activities ::= activities : [ activity ]*
```

```
activity ::= [ CAC.composite-activity |
    PAC.primitive-activity |
    MOV.move-activity |
    CAA.create-agent-activity |
    COA.create-object-activity |
    COM.communicate-activity |
    BCT.broadcast-activity |
    JAC.java-activity |
    GET.get-activity |
    PUT.put-activity ]
workframes ::= workframes : [ WFR.workframe ]*
thoughtframes ::= thoughtframes : [ TFR.thoughtframe ]*
```

## Agent

```
agent ::= agent agent-name { GRP.group-membership }
{
    { display : ID.literal-string ; }
    { cost : ID.number ; }
    { time_unit : ID.number ; }
    { location : ARE.area-name ; }
    { icon : ID.literal-string ; }
    { GRP.attributes }
    { GRP.relations }
    { GRP.initial-beliefs }
    { GRP.initial-facts }
    { GRP.activities }
    { GRP.workframes }
    { GRP.thoughtframes }
}
externalagt ::= external agent agent-name ;
agent-name ::= ID.name
```

## Class

```
class ::= class class-name { class-inheritance }
{
    { display : ID.literal-string ; }
    { cost : ID.number ; }
    { time_unit : ID.number ; }
    { resource : ID.truth-value ; }
    { icon : ID.literal-string ; }
    { GRP.attributes }
    { GRP.relations }
    { GRP.initial-beliefs }
    { GRP.initial-facts }
    { GRP.activities }
    { GRP.workframes }
    { GRP.thoughtframes }
}
class-name ::= ID.name
class-inheritance ::= extends class-name [ , class-name ]*
```

## Object

```
object ::= object object-name
    instanceof CLS.class-name
{ COB.conceptual-object-membership }
{
    { display : ID.literal-string ; }
    { cost : ID.number ; }
    { time_unit : ID.number ; }
    { resource : ID.truth-value ; }
    { location : ARE.area-name ; }
    { icon : ID.literal-string ; }
    { GRP.attributes }
    { GRP.relations }
    { GRP.initial-beliefs }
    { GRP.initial-facts }
    { GRP.activities }
    { GRP.workframes }
    { GRP.thoughtframes }
}
object-name ::= ID.name
```

## Conceptual Class

```
conceptual-class ::= conceptual_class conceptual-class-name
    { conceptual-class-inheritance }
{
    { display : ID.literal-string ; }
    { icon : ID.literal-string ; }
    { GRP.attributes }
    { GRP.relations }
}
conceptual-class-name ::= ID.name
conceptual-class-inheritance ::= extends conceptual-class-name
    [ , conceptual-class-name ]*
```

## Conceptual Object

```
conceptual-object ::= conceptual_object conceptual-object-name
instanceof COC.conceptual-class-name
{ conceptual-object-membership }
{
    { display : ID.literal-string ; }
    { icon : ID.literal-string ; }
    { GRP.attributes }
    { GRP.relations }
}
conceptual-object-name ::= ID.name
conceptual-object-membership ::= partof conceptual-object-name
[ , conceptual-object-name ]*
```

## Area Definition

```
areadef ::= areadef areadef-name { areadef-inheritance }
{
    { display : ID.literal-string ; }
    { icon : ID.literal-string ; }
    { GRP.attributes }
    { GRP.relations }
    { GRP.initial-facts }
}
areadef-name ::= ID.name
areadef-inheritance ::= extends areadef-name [ , areadef-name ]*
```

## Area

```
area ::= area area-name
instanceof ADF.areadef-name
{ partof area-name }
{
    { display : ID.literal-string ; }
    { icon : ID.literal-string ; }
    { GRP.attributes }
    { GRP.relations }
    { GRP.initial-facts }
}
area-name ::= ID.name
```

## Path

```
path ::= path path-name
{
    { display : ID.literal-string ; }
    area1 : ARE.area-name ;
    area2 : ARE.area-name ;
    { distance : ID.unsigned ; }
}
path-name ::= ID.name
```

## Attribute

```
attribute ::= { private | protected | public }
attribute-type-def
attribute-name
{ attrib-body }
;
attribute-name ::= location | ID.name
attribute-type-def ::= [ type-def
    | collection-type-def
    | relation-type-def ]
type-def ::= [ class-type-def
    | value-type-def
    | java-type-def ]
class-type-def ::= [ Agent |
    Group |
    Class |
    Object |
    ActiveClass |
    ActiveInstance |
    ActiveConcept |
    ConceptualClass |
    ConceptualObject |
    ConceptualConcept |
    AreaDef |
    Area |
    GeographyConcept |
    Concept |
    GRP.group-name |
    CLS.class-name |
    COC.conceptual-class-name |
    ADF.areadef-name ]
value-type-def ::= [ int | long | double | symbol | string | boolean ]
collection-type-def ::= [ map ]
relation-type-def ::= relation ( type-def )
java-type-def ::= java ( java-ref-type-def )
```

```
java-ref-type-def ::=
    java-class-or-interface-type-def [ [ ] ]*
java-class-or-interface-type-def ::=
    java-type-decl-specifier { java-type-arguments }
java-type-decl-specifier ::= [ ID.name [ . ID.name ]*
    | java-class-or-interface-type-def . ID.name ]
java-type-arguments ::= < java-type-argument
    [ , java-type-argument ]* >
java-type-argument ::= [ java-ref-type-def
| ? { java-wildcard-bounds } ]
java-wildcard-bounds ::= [ extends java-ref-type-def
| super java-ref-type-def ]
attrib-body ::= {
{ display : ID.literal-string ; }
}
```

## Relation

```
relation ::= { private | protected | public }
ATT.class-type-def
relation-name
{ ATT.attrib-body }
;
relation-name ::= ID.name
```

## Variable

```
variable ::=  [ collectall | foreach | forone ]
( ATT.type-def )
variable-name
{ variable-body }
;
variable-name ::= ID.name
variable-body ::= {
    { display : ID.literal-string ; }
}
```

## Initial Belief

```
initial-belief ::= ( [ value-expression
    | relational-expression ] ) ;
value-expression ::= obj-attr
    equality-operator value |
    obj-attr equality-operator
```

```
        sgl-object-ref
equality-operator ::= = | !=
evaluation-operator ::= equality-operator
    | > | >= | < | <=
obj-attr ::= tuple-object-ref . ATT.attribute-name
{ ( collection-index ) }
tuple-object-ref ::= AGT.agent-name |
    OBJ.object-name |
    COB.conceptual-object-name |
    ARE.area-name |
    VAR.variable-name |
    PAC.param-name |
    current
collection-index ::= ID.literal-string |
    ID.unsigned |
    VAR.variable-name |
    PAC.param-name
sgl-object-ref ::= AGT.agent-name |
    OBJ.object-name |
    COB.conceptual-object-name |
    ARE.area-name |
    VAR.variable-name |
    PAC.param-name |
    unknown |
    current
value ::= ID.literal-string | ID.number |
    PAC.param-name | unknown
relational-expression ::= tuple-object-ref
    REL.relation-name sgl-object-ref { is ID.truth-value }
```

## Initial Fact

```
initial-fact ::= ( [ BEL.value-expression |
    BEL.relational-expression ] ) ;
```

## Primitive Activitiy

```
primitive-activity ::= primitive_activity activity-name(
    { param-decl [ , param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned | param-name ] ; }
    { random : [ ID.truth-value | param-name ] ; }
    { min_duration : [ ID.unsigned | param-name ] ; }
    { max_duration : [ ID.unsigned | param-name ] ; }
    { resources }
}
```

```
activity-name ::= ID.name
param-decl ::= param-type param-name
param-type ::= ATT.type-def
param-name ::= ID.name
resources ::= resources : [ param-name | OBJ.object-name ]
    [ , [ param-name | OBJ.object-name ]*;
activity-ref ::= activity-name
    ( { param-expr [ , param-expr ]* } ) ;
param-expr ::= GRP.group-name |
    AGT.agent-name |
    CLS.class-name |
    OBJ.object-name |
    COC.conceptual-class-name |
    COB.conceptual-object-name |
    ARE.area-name |
    VAR.variable-name |
    ID.number |
    ID.literal-symbol |
    ID.literal-string |
    ID.truth-value
```

## Move Activity

```
move-activity ::= move PAC.activity-name (
    { PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned | PAC.param-name ] ; }
    { random : [ ID.truth-value | PAC.param-name ] ; }
    { min_duration : [ ID.unsigned | PAC.param-name ] ; }
    { max_duration : [ ID.unsigned | PAC.param-name ] ; }
    { PAC.resources }
    location : [ ARE.area-name | PAC.param-name ] ;
    { detectDepartureIn : [ ARE.area-name | PAC.param-name ]
        [ , [ ARE.area-name | PAC.param-name ] ]* ; }
    { detectDepartureInSubAreas : [ ID.truth-value |
        PAC.param-name ] ; }
    { detectArrivalIn : [ ARE.area-name | PAC.param-name ]
        [ , [ ARE.area-name | PAC.param-name ] ]* ; }
    { detectArrivalInSubAreas : [ ID.truth-value |
        PAC.param-name ] ; }
}
```

## Create Agent

```
create-agent-activity ::= create_agent
    PAC.activity-name (
    { PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned |
        PAC.param-name ] ; }
    { random : [ ID.truth-value |
        PAC.param-name ] ; }
    { min_duration : [ ID.unsigned |
        PAC.param-name ] ; }
    { max_duration : [ ID.unsigned |
        PAC.param-name ] ; }
    { PAC.resources }
    { memberof : [ GRP.group-name | PAC.param-name ]
    [ , [ GRP.group-name | PAC.param-name ]* ] ; }
    { quantity : [ ID.unsigned | PAC.param-name ] ; }
    { destination : [PAC.param-name ] ; }
    { destination_name : [ ID.literal-symbol |
        PAC.param-name ] ; }
    { location : [ ARE.area-name | PAC.param-name ] ; }
    { when : [ start | end | PAC.param-name ] ; }
}
```

## Create Area

```
create-area-activity ::= create_area
    PAC.activity-name (
    { PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned | PAC.param-name ] ; }
    { random : [ ID.truth-value | PAC.param-name ] ; }
    { min_duration : [ ID.unsigned | PAC.param-name ] ; }
    { max_duration : [ ID.unsigned | PAC.param-name ] ; }
    { PAC.resources }
    { instanceof : [ ADF.areadef-name | PAC.param-name ]
    [ , [ ADF.areadef-name | PAC.param-name ]* ] ; }
    { partof : [ ARE.area-name | PAC.param-name ] ; }
    { inhabitants : [ AGT.agent-name | OBJ.object-name
        |  PAC.param-name ] [ , [ AGT.agent-name |
            OBJ.object-name | PAC.param-name ]* ] ; }
    { destination : [PAC.param-name ] ; }
    { destination_name : [ ID.literal-symbol |
        PAC.param-name ] ; }
    { when : [ start | end | PAC.param-name ] ; }
}
```

## Create Object

```
create-object-activity ::= create_object
    PAC.activity-name (
    { PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned |
        PAC.param-name ] ; }
    { random : [ ID.truth-value |
        PAC.param-name ] ; }
    { min_duration : [ ID.unsigned |
        PAC.param-name ] ; }
    { max_duration : [ ID.unsigned |
        PAC.param-name ] ; }
    { PAC.resources }
    action : [ new | copy | PAC.param-name ] ;
    source : [ CLS.class-name |
    OBJ.object-name |
    COC.conceptual-object-name |
    COB.conceptual-object-name |
    PAC.param-name ] ;
    destination : [PAC.param-name ] ;
    { destination_name : [ ID.literal-symbol |
        PAC.param-name ] ; }
    { location : [ ARE.area-name |
        PAC.param-name ] ; }
    { conceptual_object :
    [ COB.conceptual-object-name | PAC.param-name ]
    [ , [ COB.conceptual-object-name |
        PAC.param-name ] ]* ; }
    { when : [ start | end | PAC.param-name ] ; }
}
```

## Communicate

```
communicate-activity ::= communicate
    PAC.activity-name (
{ PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned |
        PAC.param-name ] ; }
    { random : [ ID.truth-value |
        PAC.param-name ] ; }
    { min_duration : [ ID.unsigned |
        PAC.param-name ] ; }
    { max_duration : [ ID.unsigned |
        PAC.param-name ] ; }
    { PAC.resources }
    { type : [ phone | fax | email |
        face2face | terminal |
    pager | none | PAC.param-name ] ; }
    with : [ [ AGT.agent-name |OBJ.object-name |
    PAC.param-name ] [ , [ AGT.agent-name |
    OBJ.object-name | PAC.param-name ] ]* ;
    about : TDF.transfer-definition
        [ , TDF.transfer-definition ]* ;
    { when : [ start | end | PAC.param-name ] ; }
}
```

## Broadcast

```
broadcast-activity ::= broadcast
    PAC.activity-name (
    { PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned |
        PAC.param-name ] ; }
    { random : [ ID.truth-value |
        PAC.param-name ] ; }
    { min_duration : [ ID.unsigned |
        PAC.param-name ] ; }
    { max_duration : [ ID.unsigned |
        PAC.param-name ] ; }
    { PAC.resources }
    { type : [ phone | fax | email |
        face2face | terminal |
    pager | none | PAC.param-name ] ; }
    { to : [ ARE.area-name | PAC.param-name ]
        [ , [ ARE.area-name | PAC.param-name ] ]* ; }
    { toSubAreas : [ ID.truth-value | PAC.param-name ] ; }
    about : TDF.transfer-definition
        [ , TDF.transfer-definition ]* ;
    { when : [ start | end | PAC.param-name ] ; }
}
```

## Get

```
get-activity ::= get PAC.activity-name (
{ PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned | PAC.param-name ] ; }
    { random : [ ID.truth-value | PAC.param-name ] ; }
    { min_duration : [ ID.unsigned | PAC.param-name ] ; }
    { max_duration : [ ID.unsigned | PAC.param-name ] ; }
    { PAC.resources }
    items
    { source : [OBJ.object-name | AGT.agent-name |
        ARE.area-name | PAC.param-name ] ; }
    { when : [ start | end | PAC.param-name ] ; }
}
items ::= items : [ PAC.param-name | OBJ.object-name |
    AGT.agent-name ] [ , [ PAC.param-name |
    OBJ.object-name | AGT.agent-name ]* ;
```

## Put

```
put-activity ::= put PAC.activity-name (
{ PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned | PAC.param-name ] ; }
    { random : [ ID.truth-value | PAC.param-name ] ; }
    { min_duration : [ ID.unsigned | PAC.param-name ] ; }
    { max_duration : [ ID.unsigned | PAC.param-name ] ; }
    { PAC.resources }
    items
    { destination : [OBJ.object-name | AGT.agent-name |
        ARE.area-name | PAC.param-name ] ; }
    { when : [ start | end | PAC.param-name ] ; }
}
items ::= items : [ PAC.param-name | OBJ.object-name |
    AGT.agent-name ] [ , [ PAC.param-name | OBJ.object-name |
    AGT.agent-name ]* ;
```

## Gesture

```
gesture-activity ::= gesture PAC.activity-name (
{ PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned | PAC.param-name ] ; }
    { random : [ ID.truth-value | PAC.param-name ] ; }
    { min_duration : [ ID.unsigned | PAC.param-name ] ; }
    { max_duration : [ ID.unsigned | PAC.param-name ] ; }
    { PAC.resources }
    gesture : [ ID.literal-symbol | PAC.param-name ] ; }
}
```

## Java

```
java-activity ::= java PAC.activity-name (
{ PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned | PAC.param-name ] ; }
    { random : [ ID.truth-value | PAC.param-name ] ; }
    { min_duration : [ ID.unsigned | PAC.param-name ] ; }
    { max_duration : [ ID.unsigned | PAC.param-name ] ; }
    { PAC.resources }
    class : [ ID.literal-string | PAC.param-name ] ;
    { when : [ start | end | PAC.param-name ] ; }
}
```

## Composite

```
composite-activity ::= composite-activity PAC.activity-name (
{ PAC.param-decl [ , PAC.param-decl ]* } )
{
    { display : ID.literal-string ; }
    { priority : [ ID.unsigned | PAC.param-name ] ; }
    { end_condition : [ detectable | nowork ] ; }
    { WFR.detectable-decl }
    { GRP.activities }
    { GRP.workframes }
    { GRP.thoughtframes }
}
```

## Workframe

```
workframe ::= workframe workframe-name
{
    { display : ID.literal-string ; }
    { type : factframe | dataframe ; }
    { repeat : ID.truth-value ; }
    { priority : ID.unsigned ; }
    { variable-decl }
    { detectable-decl }
    { [ precondition-decl workframe-body-decl ] |
    workframe-body-decl }
}
workframe-name ::= ID.name
variable-decl ::= variables : [ VAR.variable ]*
detectable-decl ::= detectables : [ DET.detectable ]*
precondition-decl ::= when ( {
    [ PRE.precondition ] [ and PRE.precondition ]* } )
workframe-body-decl ::= do {
    [ workframe-body-element ]* }
workframe-body-element ::= [ PAC.activity-ref |
    CON.consequence | DEL.delete-operation ]
```

## Thoughtframe

```
thoughtframe ::= thoughtframe thoughtframe-name
{
{ display : ID.literal-string ; }
{ repeat : ID.truth-value ; }
{ priority : ID.unsigned ; }
{ WFR.variable-decl }
{ [ WFR.precondition-decl thoughtframe-body-decl ] |
thoughtframe-body-decl }
}
thoughtframe-name ::= ID.name
thoughtframe-body-decl ::= do {
    [ thoughtframe-body-element ; ]* }
thoughtframe-body-element ::= CON.consequence
```

## Precondition

```
precondition ::= { [ known | unknown ] } ( novalcomparison ) |
    { [ knownval | not ] } ( evalcomparison )
    novalcomparison ::= BEL.obj-attr |
    BEL.obj-attr REL.relation-name |
    BEL.tuple-object-ref REL.relation-name
    evalcomparison ::= eval-val-comp | rel-comp
    eval-val-comp ::= expression BEL.evaluation-operator expression |
    BEL.obj-attr BEL.equality-operator ID.literal-symbol |
    BEL.obj-attr BEL.equality-operator ID.literal-string |
    BEL.obj-attr BEL.equality-operator BEL.sgl-object-ref |
    BEL.sgl-object-ref BEL.equality-operator BEL.sgl-object-ref
    rel-comp ::= BEL.obj-attr REL.relation-name BEL.obj-attr
    { is ID.truth-value } |
    BEL.obj-attr REL.relation-name BEL.sgl-object-ref
    { is ID.truth-value } |
    BEL.tuple-object-ref REL.relation-name BEL.sgl-object-ref
    { is ID.truth-value }
expression ::= term | expression [ + | - ] term
term ::= factor | term [ * | / | div | mod ] factor
factor ::= primary | factor ^ primary
primary ::= - primary | element
element ::= ID.number |
BEL.obj-attr |
VAR.variable-name |
unknown
```

## Consequence

```
consequence ::= conclude ( ( resultcomparison )
    { , fact-certainty }
    { , belief-certainty } ) ;
resultcomparison ::= [ result-val-comp | PRE.rel-comp ]
result-val-comp ::=  BEL.obj-attr BEL.equality-operator
    PRE.expression |
    BEL.obj-attr BEL.equality-operator ID.literal-symbol |
    BEL.obj-attr BEL.equality-operator ID.literal-string |
    BEL.obj-attr BEL.equality-operator BEL.sgl-object-ref |
    BEL.tuple-object-ref BEL.equality-operator
    BEL.sgl-object-ref
fact-certainty ::= fc : ID.unsigned
belief-certainty ::= bc : ID.unsigned
```

## Detectable

```
detectable ::= detectable detectable-name {
    { when ( [ whenever | ID.unsigned ] ) }
    detect ( ( resultcomparison ) { ,
        detect-certainty } )
    { then detectable-action } ;
}
detectable-name ::= ID.name
```

```
resultcomparison ::= [ detect-val-comp |
    detect-rel-comp ]
detect-val-comp ::= obj-attr |
    obj-attr BEL.evaluation-operator PRE.expression |
    obj-attr BEL.evaluation-operator obj-attr |
    obj-attr BEL.equality-operator ID.literal-symbol |
    obj-attr BEL.equality-operator ID.literal-string |
    obj-attr BEL.equality-operator sgl-object-ref
detect-rel-comp ::=  detectable-object REL.relation-name |
detectable-object REL.relation-name sgl-object-ref
{ is ID.truth-value }
obj-attr ::=  detectable-tuple
    { ( BEL.collection-index ) }
detectable-tuple ::=  detectable-object .
    ATT.attribute-name
detectable-object ::= BEL.tuple-object-ref |
    < ID.name >
sql-object-ref ::= BEL.sql-object-ref |
    < ID.name > | ?
detect-certainty ::= dc : ID.unsigned
detectable-action ::= continue | impasse | abort |
    complete | end_activity
```

## Transfer Definition

```
transfer-definition ::= transfer-action ( communicative-act |
DET.resultcomparison )
    transfer-action ::= send | receive
communicative-act ::= OBJ.object-name | PAC.param-name
```

## Delete

```
delete-operation ::= delete [ VAR.variable-name |
    PAC.param-name ]
```

# Appendix B

# ANTLR Parser

The following code is detailed in the appendix to show how Brahms code is read in and parsed for storage in Java data structures. It is coded using the BNF language defined for the ANTLR parser.

```
grammar Brahms;

@header {
import java.util.HashSet;
import java.util.Stack;
import java.util.Set;
import java.util.Iterator;
import java.io.*;
}

@members {
//  Geography
String area;
String area1;
String area2;
int distance;
String instance;
String part;
int locID = 0;
int agentObjectID = 0;


String areaDef;
String ex;

Set<locations> locs = new HashSet<locations>();
Set<areaDefs> areaDefSet = new HashSet<areaDefs>();
Set<path> paths = new HashSet<path>();

//  Store all agents, objects, classes and groups
Set<agent> mas = new HashSet<agent>(); // Store all agents
Set<object> mos = new HashSet<object>();
```

```
Set<b_class> mcs = new HashSet<b_class>();
Set<group> mgs = new HashSet<group>();

//  Store all Facts
Set facAbout = new HashSet();
Set facName  = new HashSet();
Set facValue = new HashSet();

//agent
String name;
Set<String> memberOf = new HashSet<String>();
String display;
String cost;
String timeUnit;
String location;
int wfNumber = 0;
int tfNumber = 0;

// Relations
String relPriv;
String relTo;
String relName;

//Attributes
String attName;
String attPrivacy;
String attType;
int attDuration;

// Beliefs
String about; // Also acts as a variable for guard
String attributeName; // Also acts as a variable for guard
String belMath; // Math in a belief, = or a relation
String belValue; // value of the belief

//Frames
String f_name; // name of workframe or thoughtframe
String f_repeat = "false";
int f_priority = 0;
String f_event; //  States if next even needs to be popped off
//  conclude or activity stack

//Detectables
String det_name; // detectables name
String det_type; // abort, impasse etc.
String det_Math; // = or a relation
int det_id; // ID number for detectable (mainly for promela)
int dc = 100; // condition on detection
```

```
//Variables
String f_varName; // name of the variable
String f_varType; // forone, foreach, collectall etc
String f_varAssoc; //  Which class of objects or group of agents
int varNo = 0; // ID number for variable (promela)

String valueOwned; // Owner of a value to be assigned to a belief
String valueAttr; // the attribute
String valueOwned2; // incase a 2nd is needed
String valueAttr2; // incase a 2nd is needed
String valueOperator; // +, - *, /

//Guards
String f_guardType; // known, knownval etc.
String leftAssoc; // Agent on left side of equation
String leftAttr; // attribute on left side of equation
String f_guardMathSymb; // math symbol, = or a relation
String rightAssoc; // Agent on right hand side
String rightAttr; // attribute on right hand side
String value; // If not an attribute but an integar or string

//Activity
String actName;  //  Name of activity
eventType eveType;
int duration = -1;
String paramType;
String paramType2;
String parameter;
String parameter2;
eventType TypeEvent; // evenType declared in event.java
String whom_where;
String whom_where2;
String messAbout;
String messAtt;
String messAbout2;
String messAtt2;
Set<messages> mess = new HashSet<messages>();

//Concludes
String f_concAssoc;  //  Who the conclude belief belongs to
String f_concBelief; // name of the attribute
String f_concValue; // new value Needs changing
int concID = 0;
int bc = 100;
int fc = 100;

// Stacks
// Instances of the class workframes
Set<workframe> workframes = new HashSet<workframe>();
```

```
Set<thoughtframe> thoughtframes = new HashSet<thoughtframe>();
Set<relation> relations = new HashSet<relation>();
Set<variable> variables = new HashSet<variable>();
Set<detectable> detectables = new HashSet<detectable>();
Set<activity> activities = new HashSet<activity>();
Set<attribute> attributes = new HashSet<attribute>();
Set<belief> beliefs = new HashSet<belief>();
Set<fact> facts = new HashSet<fact>();
Set<guard> guards = new HashSet<guard>();
List<event> events = new ArrayList<event>();

//  Temp variables
Set<workframe> tempWFset = new HashSet<workframe>();
Set<variable> tempVarset = new HashSet<variable>();
Set<guard> tempGuardset = new HashSet<guard>();
List<event> tempEventList = new ArrayList<event>();

int cou = 0;
}
// Start rule
prog:(((geography NEWLINE*)
|(agent NEWLINE* )
|(object NEWLINE*)
|(class1 NEWLINE*)
|group NEWLINE*)
{
cou++;
name = "";
display = "";
cost = "";
timeUnit = "";
location = "";
relations.clear();
memberOf.clear();
activities.clear();
attributes.clear();
beliefs.clear();
facts.clear();
workframes.clear();
thoughtframes.clear();})+

{
Set<agent> TempMAS = new HashSet<agent>(mas);
Set<object> TempMOS = new HashSet<object>(mos);
Set<b_class> TempMCS = new HashSet<b_class>(mcs);
Set<group> TempMGS = new HashSet<group>(mgs);
Set<locations> TempLocs = new HashSet<locations>(locs);
Set<areaDefs> TempAreaDefs = new HashSet<areaDefs>(areaDefSet);
Set<path> TempAreaPaths = new HashSet<path>(paths);
```

```
Set<fact> TempFacts = new HashSet<fact>(facts);
MultiAgentSystem multi = new MultiAgentSystem(TempMAS, TempMOS,
TempMCS, TempMGS, TempLocs, TempAreaDefs, TempAreaPaths,
TempFacts);}
;

geography
:
area
| areadef
| path
;

area
: 'area' areaName = ID {area = $areaName.text;}
('instanceof' instanceOf = ID {instance =
$instanceOf.text;})?
('partof' partOf = ID {part = $partOf.text;})? '{' '}'
{locations loca =  new locations(area, instance, part, locID);
locs.add(loca);
locID++;
}
;

areadef
: 'areadef' areaDefinition = ID {areaDef = $areaDefinition.text;}
('extends' extension = ID {ex = $extension.text;})? '{' '}'
{areaDefs ad = new areaDefs(areaDef, ex); areaDefSet.add(ad);}
;

path
: 'path' ID NEWLINE* '{' NEWLINE*
'area1:' a1 = ID {area1 = $a1.text;}';' NEWLINE*
'area2:' a2 = ID {area2 = $a2.text;}';' NEWLINE*
'distance:' dist = INT {distance = Integer.parseInt($dist.text);}';'
NEWLINE*
{path path1 = new path(area1, area2, distance); paths.add(path1);}
'}'
;

class1  : 'class' agentName=ID {name = $agentName.text;}
NEWLINE*
('instanceof' (mem = ID {memberOf.add($mem.text);} (',')?)+)?
'{' NEWLINE*
(displayName = display {display = $displayName.text;} NEWLINE*)?
(theCost = cost {cost = $theCost.text;} NEWLINE*)?
( theTimeUnit = timeUnit {timeUnit = $theTimeUnit.text;} NEWLINE*)?
( location NEWLINE*)?
( attributes NEWLINE*)?
( relations NEWLINE*)?   /* SKIPPED RELATIONS SO FAR! */
( beliefs NEWLINE*)?
( facts NEWLINE*)?
( activities NEWLINE*)?
( workframes NEWLINE*)?
( thoughtframes NEWLINE*)?
'}'
{
Set<relation> TempRelations1 = new HashSet
<relation>(relations);
Set<activity> TempActivities1 = new HashSet
<activity>(activities);
Set<attribute> TempAttributes1 = new HashSet
<attribute>(attributes);
Set<belief> TempBeliefs1 = new HashSet<belief>(beliefs);
Set<fact> TempFacts1 = new HashSet<fact>(facts);
Set<workframe> TempWorkframes1 = new HashSet
<workframe>(workframes);
Set<thoughtframe> TempThoughtframes1 = new HashSet
<thoughtframe>(thoughtframes);
Set<String> TempMemberOf1 = new HashSet<String>(memberOf);

mcs.add(new b_class(
name,
display,
cost,
timeUnit,
location,
TempMemberOf1,
TempRelations1,
TempActivities1,
TempAttributes1,
TempBeliefs1,
TempFacts1,
TempWorkframes1,
TempThoughtframes1));
wfNumber = 0;
}
;

group  : 'group' agentName=ID {name = $agentName.text;} NEWLINE*
('memberof' (mem = ID {memberOf.add($mem.text);} (',')?)+)?
'{' NEWLINE*
(displayName = display {display = $displayName.text;} NEWLINE*)?
(theCost = cost {cost = $theCost.text;} NEWLINE*)?
( theTimeUnit = timeUnit {timeUnit = $theTimeUnit.text;} NEWLINE*)?
( location NEWLINE*)?
( attributes NEWLINE*)?
( relations NEWLINE*)?   /* SKIPPED RELATIONS SO FAR! */
( beliefs NEWLINE*)?
( facts NEWLINE*)?
( activities NEWLINE*)?
( workframes NEWLINE*)?
( thoughtframes NEWLINE*)?
'}'
{
Set<relation> TempRelations2 = new HashSet
<relation>(relations);
Set<activity> TempActivities2 = new HashSet
<activity>(activities);
Set<attribute> TempAttributes2 = new HashSet
<attribute>(attributes);
Set<belief> TempBeliefs2 = new HashSet
<belief>(beliefs);
Set<fact> TempFacts2 = new HashSet<fact>(facts);
Set<workframe> TempWorkframes2 = new HashSet
<workframe>(workframes);
Set<thoughtframe> TempThoughtframes2 = new HashSet
<thoughtframe>(thoughtframes);
Set<String> TempMemberOf2 = new HashSet<String>(memberOf);

mgs.add(new group(
name,
display,
cost,
timeUnit,
location,
TempMemberOf2,
TempRelations2,
TempActivities2,
TempAttributes2,
TempBeliefs2,
TempFacts2,
TempWorkframes2,
TempThoughtframes2));
wfNumber = 0;
} ;


agent : 'agent' agentName=ID {name = $agentName.text;}
NEWLINE* ('memberof' (mem = ID {memberOf.add($mem.text);}
(',')?)+)? '{' NEWLINE*
(displayName = display {display = $displayName.text;} NEWLINE*)?
(theCost = cost {cost = $theCost.text;} NEWLINE*)?
( theTimeUnit = timeUnit {timeUnit = $theTimeUnit.text;} NEWLINE*)?
( location NEWLINE*)?
( attributes NEWLINE*)?
( relations NEWLINE*)?
( beliefs NEWLINE*)?
( facts NEWLINE*)?
( activities NEWLINE*)?
( workframes NEWLINE*)?
( thoughtframes NEWLINE*)?
'}'
{
Set<relation> TempRelations3 = new HashSet
<relation>(relations);
Set<activity> TempActivities3 = new HashSet
<activity>(activities);
Set<attribute> TempAttributes3 = new HashSet
<attribute>(attributes);
Set<belief> TempBeliefs3 = new HashSet<belief>(beliefs);
Set<fact> TempFacts3 = new HashSet<fact>(facts);
Set<workframe> TempWorkframes3 = new HashSet
<workframe>(workframes);
Set<thoughtframe> TempThoughtframes3 = new HashSet
<thoughtframe>(thoughtframes);
Set<String> TempMemberOf3 = new HashSet<String>(memberOf);

mas.add(new agent(
name,
agentObjectID,
TempMemberOf3,
display,
cost,
timeUnit,
location,
TempRelations3,
TempActivities3,
TempAttributes3,
TempBeliefs3,
TempFacts3,
TempWorkframes3,
TempThoughtframes3));
wfNumber = 0;
agentObjectID++;
}
;

object : 'object' agentName=ID {name = $agentName.text;}
NEWLINE* ('instanceof' (mem = ID {memberOf.add($mem.text);}
(',')?)+)? '{' NEWLINE*
(displayName = display {display = $displayName.text;}
NEWLINE*)?
(theCost = cost {cost = $theCost.text;} NEWLINE*)?
( theTimeUnit = timeUnit {timeUnit = $theTimeUnit.text;} NEWLINE*)?
( location NEWLINE*)?
( attributes NEWLINE*)?
( relations NEWLINE*)?   /* SKIPPED RELATIONS SO FAR! */
( beliefs NEWLINE*)?
( facts NEWLINE*)?
( activities NEWLINE*)?
( workframes NEWLINE*)?
( thoughtframes NEWLINE*)?
'}'
{
Set<relation> TempRelations4 = new HashSet
<relation>(relations);
Set<activity> TempActivities4 = new HashSet
<activity>(activities);
Set<attribute> TempAttributes4 = new HashSet
<attribute>(attributes);
Set<belief> TempBeliefs4 = new HashSet<belief>(beliefs);
Set<fact> TempFacts4 = new HashSet<fact>(facts);
Set<workframe> TempWorkframes4 = new HashSet
<workframe>(workframes);
Set<thoughtframe> TempThoughtframes4 = new HashSet
<thoughtframe>(thoughtframes);
Set<String> TempMemberOf4 = new HashSet<String>(memberOf);

mos.add(new object(
name,
agentObjectID,
TempMemberOf4,
display,
cost,
timeUnit,
location,
TempRelations4,
TempActivities4,
TempAttributes4,
TempBeliefs4,
TempFacts4,
TempWorkframes4,
TempThoughtframes4));
wfNumber = 0;
agentObjectID++;
}
;

display : 'display:' '"' ID '"' ';'
;

cost : 'cost:' INT ';' /* Need to make this a double somehow */
;
```

```
timeUnit: 'time:' INT ';'
;

location: 'location:' theLocation = ID {location =
$theLocation.text;}
';'
;

attributes
: 'attributes:' (NEWLINE* attribute{attributes.add
(new attribute(attPrivacy,attType,attName));})*
;

relations
: 'relations:' (NEWLINE* relation{relations.add
(new relation(relPriv,relName,relTo));})*
;

relation
: privacy = ('public' | 'private') {relPriv = $privacy.text;}
to = ID {relTo = $to.text;} name = ID
{relName = $name.text;}';'
;

attribute
: privacy = ('public' | 'private')? {attPrivacy = $privacy.text;}
type = ID/*('int'| 'String'| 'boolean'|'double'|)*/
{attType = $type.text;} name = ID {attName = $name.text;} ';'
;


beliefs : 'initial_beliefs:' (NEWLINE* belief ';')*;

facts : 'initial_facts:' (NEWLINE* fact ';')* ;

belief  :  ('('d=attRef math = (MATH|ID)
{belMath = $math.text;} (valID = (ID|'current')
{belValue = $valID.text;}|valInt = INT
{belValue = $valInt.text;}| valTrue = 'true'
{belValue = $valTrue.text;}| valFalse = 'false'
{belValue = $valFalse.text;})  ')')
{beliefs.add(new belief(about, attributeName,
belMath, belValue)); belMath = null;
attributeName = null; about = null; belValue = null;};

fact  :  ('('d=attRef math = (MATH|ID) {belMath = $math.text;}
(valID = ID {belValue = $valID.text;}|valInt = INT
{belValue = $valInt.text;}| valTrue = 'true'
{belValue = $valTrue.text;}| valFalse = 'false'
```

```
{belValue = $valFalse.text;})  ')')
{facts.add(new fact(about, attributeName, belMath, belValue));
belMath = null; attributeName = null; about = null;
belValue = null;};


activities
: 'activities:' (NEWLINE* {actName = null; paramType = null;
paramType2 = null; parameter = null; parameter2 = null;
duration = -1; messAbout = null; messAtt = null;
messAbout2 = null; messAtt2 = null;} activity)*;


: ('move' {eveType = eventType.Move;} name = ID
{actName = $name.text;} '(' ((pType = ID param = ID)
{paramType = $pType.text; parameter = $param.text;})?
')' NEWLINE* '{' NEWLINE* 'location:' where = ID
{whom_where = $where.text; messAbout = null;
messAtt = null;} ';' NEWLINE*  | 'primitive_activity'
{eveType = eventType.PrimAct;} name = ID
{actName = $name.text;} '(' ((pType = ID param = ID)
{paramType = $pType.text; parameter = $param.text;})?
 ')' NEWLINE* '{' NEWLINE* 'max_duration:'
 (dur = INT{duration = Integer.parseInt ($dur.text);}|
 ID{duration = -1;}) {whom_where = null; messAbout = null;
messAtt = null;} ';' NEWLINE* |'communicate'
{eveType = eventType.CommAct;} name = ID
{actName = $name.text;} '(' ((pType = ID param = ID)
{paramType = $pType.text; parameter = $param.text;})?
((',' pType2 = ID param2 = ID){paramType2 = $pType2.text;
parameter2 = $param2.text;})? ')' NEWLINE* '{'
NEWLINE* 'max_duration:'
(dur = INT{duration = Integer.parseInt($dur.text);}|
ID{duration = -1;}) ';'
NEWLINE* 'with:' where = ID {whom_where = $where.text;} ';'
NEWLINE* 'about:'
NEWLINE* ('send(' about = (ID|'current')
{messAbout = $about.text;} '.' attrib = ID
{messAtt = $attrib.text;} MATH about2 = (ID|'current')
{messAbout2 = $about2.text;} '.' attrib2 = ID {messAtt2 =
$attrib2.text;} ')' (',' NEWLINE*)? {mess.add(
new messages(messAbout, messAbout2, messAtt, messAtt2));
messAbout = null; messAbout2 = null; messAtt2 = null;
messAtt2 = null;})+ ';' NEWLINE* ('when: end;'
 NEWLINE*)?) {Set<messages> TempMess = new HashSet
 <messages>(mess); activities.add(new activity(eveType,
 actName, paramType, parameter,
 paramType2, parameter2, duration, whom_where, TempMess));
 duration = 0; mess.clear();}
```

```
'}' ;

workframes
:'workframes:'  (NEWLINE* workframe
{

Set<variable> TempVariables = new HashSet
<variable>(variables);
Set<detectable> TempDetectables = new HashSet
<detectable>(detectables);
Set<guard> TempGuards = new HashSet
<guard>(guards);
List<event> TempEvents = new ArrayList
<event>(events);
Set<agent> TempMas = new HashSet<agent>(mas);
workframes.add(new workframe(TempMas, wfNumber, name, f_name,
f_repeat, f_priority, TempVariables, TempDetectables,
TempGuards, TempEvents));
wfNumber = wfNumber+1;
f_name = null;
f_repeat = "false";
f_priority = 0;
variables.clear();
detectables.clear();
guards.clear();
events.clear();
det_id = 0;
}
)*;

workframe
:

'workframe' wfname=ID {f_name = $wfname.text;} NEWLINE* '{' NEWLINE*
(repeat NEWLINE*)?
(priority NEWLINE*)?
('variables:' (NEWLINE* variable {variables.add(new variable(varNo,
f_varType, f_varAssoc, f_varName)); varNo++;})* {varNo = 0;}
NEWLINE*)? ('detectables:' (NEWLINE* detectable)* NEWLINE*)?
'when' '(' condition   NEWLINE* ('and' NEWLINE* condition NEWLINE*)*
')' NEWLINE*
'do' NEWLINE* '{' NEWLINE*
(event NEWLINE*)*
'}' NEWLINE*
'}'

;

repeat
```

```
: 'repeat:' rep = ('true'|'false'|'once') {f_repeat = $rep.text;}';'
;

priority
: 'priority:' pri = INT {f_priority = Integer.parseInt($pri.text);}
';';

variable:
type = 'forone' {f_varType = $type.text;} '(' assoc = ID {
f_varAssoc = $assoc.text;} ')' name = ID
{f_varName = $name.text;} ';' | type = 'foreach'
{f_varType = $type.text;} '(' assoc = ID {f_varAssoc =
$assoc.text;} ')' name = ID {f_varName = $name.text;} ';'
| type = 'collectall' {f_varType = $type.text;} '('
assoc = ID {f_varAssoc = $assoc.text;} ')' name = ID
{f_varName = $name.text;} ';'
;

detectable
: {dc = 100; about = null; attributeName = null; valueOwned = null;
value = null; valueAttr = null;}
'detectable' detname = ID {det_name = $detname.text;} NEWLINE* '{'
 NEWLINE* ('when' '(' 'whenever' ')'
|'when' '(' INT ')')
NEWLINE* 'detect' '(' '(' attRef detMath = MATH {det_Math =
$detMath.text;} leftExpr  ')' (',' 'dc:' detdc = INT
{dc = Integer.parseInt($detdc.text);} )?
')' NEWLINE* 'then' dettype = ('complete' |
 'abort' | 'continue'|'impasse')
{det_type = $dettype.text;} ';' NEWLINE* '}'
{detectables.add(new detectable
(det_id, det_name, about, attributeName, valueOwned, valueAttr,
value, det_Math, det_type, dc, wfNumber));}
{about = null; attributeName = null; valueOwned = null;
value = null; dc = 100; valueAttr = null; det_id++;}
;

event : conclude
| action
{TypeEvent = null; about = null; attributeName = null;
valueOwned = null; whom_where = null; whom_where2 = null;
value = null; bc = 100; fc = 100;
actName = null; duration = 0;};

conclude: {TypeEvent = eventType.Conc; about = null;
attributeName = null; valueOwned = null; valueOwned2 = null;
value = null; bc = 100; fc = 100; valueAttr = null;
valueAttr2 = null; valueOperator = null;} 'conclude' '(' '('
attRef  MATH multiexpr ')' (',' 'bc:' belCon = INT
```

```
{bc = Integer.parseInt($belCon.text);})? (',' 'fc:'
facCon = INT {fc = Integer.parseInt($facCon.text);})? ')' ';'
{concID = concID-1; Set<variable> tempVars = new HashSet
<variable>(variables);
events.add(new event(concID, f_name, TypeEvent, about,
attributeName, valueOwned, valueOwned2, value, valueAttr,
valueAttr2, valueOperator, bc, fc, tempVars));};

action : name = ID {actName = $name.text;} '(' (param = ID
{whom_where = $param.text;}| dur = INT {duration =
Integer.parseInt($dur.text);})?
(',' param2 = ID {whom_where2 = $param.text;})? ')' ';'
{Set<activity> tempActivities = new HashSet<activity>
(activities); Set<variable> tempVars = new HashSet<variable>
(variables); concID = concID-1; events.add(new event
(concID, f_name, actName, whom_where, whom_where2,
duration, tempActivities, tempVars)); actName = null;
 duration = 0; whom_where = null; whom_where2 = null;}
;

thoughtframes
: 'thoughtframes:' (NEWLINE* thoughtframe {
Set<variable> TempVariables = new HashSet
<variable>(variables);
Set<guard> TempGuards = new HashSet
<guard>(guards);
List<event> TempEvents = new ArrayList
<event>(events);
Set<agent> TempMas = new HashSet
<agent>(mas);
thoughtframes.add(new thoughtframe(TempMas, tfNumber,
name,f_name, f_repeat, f_priority, TempVariables,
TempGuards, TempEvents));
tfNumber = tfNumber+1;
f_name = null;
f_repeat = "false";
f_priority = 0;
variables.clear();
guards.clear();
events.clear();
}
)* ;

thoughtframe
:
'thoughtframe' tfname = ID {f_name = $tfname.text;} NEWLINE*
'{' NEWLINE* (repeat NEWLINE*)? (priority NEWLINE*)?
('variables:' (NEWLINE* variable {variables.add(new
variable(varNo, f_varType, f_varAssoc, f_varName));
varNo++;})* {varNo = 0;} NEWLINE*)? 'when' '('
condition NEWLINE* ('and' NEWLINE*   condition
NEWLINE*)* ')'NEWLINE* 'do' NEWLINE* '{' NEWLINE*
(conclude NEWLINE*)*
'}' NEWLINE*
'}'
;

// For when refering to an agent and its attribute
attRef : who = (ID|'current') {about = $who.text;}
('.' att = ID {attributeName = $att.text;})?;

// Left side of the equation
leftExpr: valueOwner = (ID|'current')
{valueOwned = $valueOwner.text; } '.'
theValue = ID {valueAttr = $theValue.text;}
| theValue=(ID|'true'|'false'|INT)
{valueOwned = null; valueAttr = null;
value = $theValue.text; }
;
// right side of the equation
rightExpr
: (valueOwner = (ID|'current') {valueOwned2 =
$valueOwner.text; } '.')? theValue = ID
{valueAttr2 = $theValue.text;}
| theValue=('true'|'false'|INT)
{valueOwned2 = null; valueAttr2 = null;
value = $theValue.text; };

// For different types of concludes
condition
: type = ('knownval'
| 'not')? {f_guardType = $type.text;} '('
(leftGuard mathSymb = MATH
{f_guardMathSymb = $mathSymb.text;} rightGuard
|  who = (ID|'current') {/*about*/
leftAssoc = $who.text;
leftAttr = null;} relate =  ID
{f_guardMathSymb = $relate.text;}
att =  (ID|'current') {/*attributeName*/
rightAssoc = $att.text;
rightAttr = null; value = ""; valueOwned = "";})
{guards.add(new guard(f_guardType, f_guardMathSymb,
 leftAssoc, rightAssoc, leftAttr, rightAttr, value));} ')'
| 'known' {f_guardType = $type.text;} '(' attRef ')'
{f_guardType = null; f_guardMathSymb = null;
leftAssoc = null; rightAssoc = null; leftAttr = null;
rightAttr = null; value = null;}
;
```

```
// Left side of the guard
leftGuard
: ((valueOwner = (ID|'current')
{leftAssoc = $valueOwner.text; } '.')
theValue = ID {leftAttr = $theValue.text;}
|theValue = ID {leftAssoc = null;
leftAttr = $theValue.text;});

// Right side of the guard
rightGuard
: {rightAssoc = null; rightAttr = null;value = null;}
(valueOwner = (ID|'current')
{rightAssoc = $valueOwner.text; } '.')
theValue = ID { rightAttr = $theValue.text;}
| theValue = (INT|ID|'true'|'false')
{value = $theValue.text;
rightAssoc = null; rightAttr = null;}
;
// Joins the left side of the expression with the right side
multiexpr
: leftExpr (op = OPERATORS {valueOperator = $op.text;}
theval = rightExpr)? ;


ID  :   ('a'..'z'|'A'..'Z'|'_')+ INT?
;
INT :   '-'?('0'..'9')+ ;

MATH : (('!')?'=')|(('=')?'<')|(('=')?'>');

NEWLINE:'\r'? '\n'
|'\r'
|'\t' ;
OPERATORS
: '+'|'-'|'*'|'/';
WS  :   (' '|'\t')+ {skip();} ;

// Comments
COMMENT
: '/*' .* '*/' {$channel=HIDDEN;}
;
LINE_COMMENT
: '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
;
```

# Appendix C

# Java Intermediary Representation

The following sections details the Java programming code used to store the data of a Brahms and translate it into partial instantiations of the Brahms semantics in *PROMELA* code. This code is detailed in the appendix to help readers understand how the project has been programmed. The code is broken up into the following sections:

- Main Method. This class reads in the Brahms code as a text file, which is accepted as an argument from the command line, and then executes the parser generated by ANTLR to parse the simulation. The Parsing code identifies the main class for translation as 'MultiAgentSystem. When executed this class instantiates all the other classes and begins the translation process.
- The Scheduler. This is the 'MultiAgentSystem class which instantiates all the other classes and translates the scheduler semantic rules in *PROMELA*.
- Agents. This is the agent class, it instantiates the agent and implements the agents semantic rules in *PROMELA*.
- Groups. This class stores all the details of a group, its thoughtframes, workframes, etc. When an agent starts generating its semantics in the agent class it will check which group it is a member of and retrieve any details from this group.
- Classes. This class stores all the details of an object class, its thoughtframes, workframes, etc. When an object starts generating its semantics in the object class it will check which class it is a member of and retrieve any details from this class.
- Attributes. This class stores the details of an attribute, when an agent stores all its attributes it is in a set of instances of this class.
- Relationships. This class stores the details of a relation, when an agent stores all its relations it is in a set of instances of this class.
- Beliefs. This class stores the details of a belief, when an agent stores all its beliefs it is in a set of instances of this class.

- Facts. This class stores the details of a fact, when the system stores all the facts it is in a set of instances of this class.
- Activities. This class stores the details of an activity, when an workframe or a thoughtframe stores all its activities it is in a set of instances of this class.
- Event. This class stores the semantic rules of how to execute an event; it decides what type of event it is and how it is to be handled.
- Concludes. This class stores the details of a conclude, when workframe or a thoughtframe stores its concludes it is in a set of instances of this class.
- Communication Messages. This class stores all the details of a communication message, when a communication event stores all its messages it is in a set of instances of this class.
- Workframes. This class stores the details of a workframe, when an agent stores all its workframes it is in a set of instances of this class.
- Detectables. This class stores the details of a detectable, when workframe stores its detectables it is in a set of instances of this class.
- Thoughtframes. This class stores the details of a workframe, when an agent stores all its workframes it is in a set of instances of this class.
- Guard Conditions. This class stores the details of a guard condition, when a workframe, thoughtframe or a detectable stores its guard conditions it is in a set of instances of this class.
- Variables. This class stores the details of a variable, when workframe or a thoughtframe stores its variables it is in a set of instances of this class.
- Geography: Area Definitions. This class stores the details of area definitions; an instance is created for this class for every area definition and stored in a set in ‘MultiAgentSystem.
- Geography: The Locations. This class stores the details of locations; an instance is created for this class for every location and stored in a set in ‘MultiAgentSystem.
- Geography: Paths between Areas. This class stores the details of the paths between locations; an instance is created for this class for every path and stored in a set in ‘MultiAgentSystem. These paths are then used to create a matrix describing the shortest path to and from every location for every location.
- Geography: Calculating Undefined Routes. This class uses Dijkstras algorithm to calculate the shortest paths between locations and enters them into the path matrix described above.

## C.1 The Main Method

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
```

```
*Date: Dec 2012
**/

/*
The main method which runs the ANTRL parser
*/

import org.antlr.runtime.*;
import java.io.*;

public class Main {
    public static void main(String[] args) throws Exception {
        try{
            FileInputStream file = new FileInputStream(args[0]);
            ANTLRInputStream input = new ANTLRInputStream(file);
            BrahmsLexer lexer = new BrahmsLexer(input);
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            BrahmsParser parser = new BrahmsParser(tokens);
            parser.prog();
        }
        catch(Exception e) {}
    }
}
```

## C.2 The Scheduler

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
This is the main class which forms the multi-agent system.  This
class stores all the agents, objects, groups etc.

This class relates to the System's Tuple in the semantics i.e.
<Agents, currect agent, Beliefs, Facts, Time>

Agents in the tuple refers to all agents and objects
(for simplicity).  Current agent under consideration isn't
included as this only happens during run time.  These data
structures only include what comes from the Brahms code,
not any variables which represent "under the hood"
operations such as time.
*/

import java.util.*;
```

```
class MultiAgentSystem
{
    // All the data about agents/objects
    Set<agent> agents = new HashSet<agent>();
    Set<object> objects = new HashSet<object>();
    Set<b_class> classes = new HashSet<b_class>();
    Set<group> groups = new HashSet<group>();
    Set<relation> relations = new HashSet<relation>();
    Set<activity> activities = new HashSet<activity>();
    Set<attribute> attributes = new HashSet<attribute>();
    Set<attribute> allAttributes = new HashSet<attribute>();
    Set<relation> allRelations = new HashSet<relation>();
    Set<belief> beliefs = new HashSet<belief>();
    Set<fact> facts = new HashSet<fact>();
    Set<guard> guards = new HashSet<guard>();
    Set<conclude> concludes = new HashSet<conclude>();
    Set<workframe> workframes = new HashSet<workframe>();
    Set<thoughtframe> thoughtframes = new HashSet<thoughtframe>();


    /***************
    *Promela fields*
    ****************/

    String name; // Used to name the agent/object under consideration
    // Stores all the identification numbers of agents,
    // objects and locations
    String identificationNumbers[];

    int numberOfAgentsObjects; // Used for array sizes
    int numberOfEverything; // Used for array sizes
    // Work around until I make code to count max depth of a workframe
    //    or thoughtframe
    int maxDepth;
    int maxVar;
    //Counts for max number of work and thoughtframes



    Set<locations> locs = new HashSet<locations>();
    Set<areaDefs> areaDefinitions = new HashSet<areaDefs>();
    Set<path> paths = new HashSet<path>();
    int[][] adjacencyMatrix; // Used for calculating paths


    String promelaCode = "";


    public MultiAgentSystem()
```

```
{ }

public MultiAgentSystem(Set new_agents, Set new_objects, Set
    new_classes, Set new_groups, Set new_locs,
    Set new_areaDefinitions, Set new_paths, Set new_facts)
{
    agents = new_agents;
    objects = new_objects;
    classes = new_classes;
    groups = new_groups;
    locs = new_locs;
    areaDefinitions = new_areaDefinitions;
    paths = new_paths;
    facts = new_facts;
    numberOfAgentsObjects = agents.size() + objects.size()+2;
    numberOfEverything = locs.size() + numberOfAgentsObjects+2;
    adjacencyMatrix = new int[locs.size()][locs.size()];
    identificationNumbers = new String[numberOfEverything];
    //identificationNumbers[0] = "Environment";
    buildAdjacencyMatrix();
    toPromela();
}

public void buildAdjacencyMatrix(){
    for(int i = 0; i < locs.size();i++){
        for(int j = 0; j < locs.size();j++){
            adjacencyMatrix[i][j] = 99999;
        }

    }

    int areaID1 = -1;
    int areaID2 = -1;

    //  Generate initial adjacency matrix for the geography
    for (Iterator<path> pathit = paths.iterator();
            pathit.hasNext(); ){
        path pt = pathit.next();
        for (Iterator<locations> Locit = locs.iterator();
                Locit.hasNext(); ){
            locations l = Locit.next();
            if(l.getName().equals(pt.getArea1())){
                areaID1 = l.getID();
            }
            if(l.getName().equals(pt.getArea2())){
                areaID2 = l.getID();
            }

        }
```

```
            adjacencyMatrix[areaID1][areaID2] = pt.getDist();
            adjacencyMatrix[areaID2][areaID1] = pt.getDist();

        }
    }

    public void toPromela(){
        String thoughtWorkInit = "";
        for (Iterator<agent> agentit = agents.iterator();
                agentit.hasNext(); ){
            agent ag = agentit.next();
            thoughtWorkInit = thoughtWorkInit.concat(
                ag.thoughtWorkInitialisation(groups) + "\n");
            allAttributes.addAll(ag.getAttributes());
            // collect all relations
            allRelations.addAll(ag.getRelations());
        }
        for (Iterator<object> objectit = objects.iterator();
                objectit.hasNext(); ){
            object ob = objectit.next();
            thoughtWorkInit = thoughtWorkInit.concat
                (ob.thoughtWorkInitialisation(classes) + "\n");
            allAttributes.addAll(ob.getAttributes());
            // collect all relations
            allRelations.addAll(ob.getRelations());
        }
        promelaCode = promelaCode.concat(
            "/*Environment's Variables*/\n");
        promelaCode = promelaCode.concat("int choice = 0;\n");
        //If there is more than 1 variable used in an event
        promelaCode = promelaCode.concat("int multiVarOne = 0;\n");
        promelaCode = promelaCode.concat("int multiVarTwo = 0;\n");
        promelaCode = promelaCode.concat("bool EnvironmentActive
            = true;\n");
         // temporary variable
        promelaCode = promelaCode.concat("int tempIndex = -1;\n");
        promelaCode = promelaCode.concat("int timeDeduction;\n");
        // temporary variable
        promelaCode = promelaCode.concat("int new;\n");

        for (Iterator<agent> agentit = agents.iterator();
                agentit.hasNext(); ){
            agent ag = agentit.next();
            promelaCode = promelaCode.concat("bool " + ag.getName()+
                "Active = true;\n");
            promelaCode = promelaCode.concat("int " + ag.getName()+
                "_timeRemaining = -1;\n");
            promelaCode = promelaCode.concat("int cnt" + ag.getName()+
                "= 0;\n");
            // flag to set time remaining at 0 so agents return and
            // process thoughtframes again.  Due to communication
            // updating agent's beliefs after  the cycle
            promelaCode = promelaCode.concat("bool comm" +
                ag.getName()+ "= false;\n");
        }
        for (Iterator<object> objectit = objects.iterator();
                objectit.hasNext(); ){
            object ob = objectit.next();
            promelaCode = promelaCode.concat("bool " + ob.getName()+
                "Active = true;\n");
            promelaCode = promelaCode.concat("int " + ob.getName()+
                "_timeRemaining = -1;\n");
            promelaCode = promelaCode.concat("int cnt" + ob.getName()+
                "= 0;\n");
            promelaCode = promelaCode.concat("bool comm"+ob.getName()+
                "= false;\n");
        }
        promelaCode = promelaCode.concat("int cntEnvironment = 0;\n");
        promelaCode = promelaCode.concat("mtype = {Environment,
            complete, abort, continue, impasse, null\n");

        Set<String> workframeNames = new HashSet<String>();
        Set<String> thoughtframeNames = new HashSet<String>();
        for (Iterator<agent> MASit = agents.iterator();
                MASit.hasNext(); ){
            agent ag = MASit.next();
            name = ag.getName();
            promelaCode = promelaCode.concat("    ,ag_" +name+"\n");
            promelaCode = promelaCode.concat("    ," + name + "\n");
            Set<workframe> tempWorkframes = new HashSet<workframe>
                (ag.getWorkframes());
            Set<thoughtframe> tempThoughtframes = new HashSet
                <thoughtframe>(ag.getThoughtframes());
            for (Iterator<thoughtframe> tfit = tempThoughtframes.
                    iterator(); tfit.hasNext(); ){
                thoughtframe tf = tfit.next();
                thoughtframeNames.add(tf.getName());
            }
            for (Iterator<workframe> wfit = tempWorkframes.iterator();
                    wfit.hasNext(); ){
                workframe wf = wfit.next();
                workframeNames.add(wf.getName());
            }
        }
        for (Iterator<object> MOSit = objects.iterator();
                MOSit.hasNext(); ){
            object ob = MOSit.next();
            name = ob.getName();
```

```
        promelaCode = promelaCode.concat("    ,ob_" +
            name+ "\n");
        promelaCode = promelaCode.concat("    ," + name + "\n");
        Set<workframe> tempWorkframes = new HashSet<workframe>(
            ob.getWorkframes());
        Set<thoughtframe> tempThoughtframes = new HashSet
            <thoughtframe>(ob.getThoughtframes());
        for (Iterator<thoughtframe> tfit = tempThoughtframes.
                iterator(); tfit.hasNext(); ){
            thoughtframe tf = tfit.next();
            thoughtframeNames.add(tf.getName());
        }
        for (Iterator<workframe> wfit = tempWorkframes.iterator();
                wfit.hasNext(); ){
            workframe wf = wfit.next();
            workframeNames.add(wf.getName());
        }
    }
    for (Iterator<locations> locit = locs.iterator();
            locit.hasNext(); ){
        locations l = locit.next();
        name = l.getName();
        promelaCode = promelaCode.concat("    ," + name + "\n");
    }
    for (Iterator<group> groupit = groups.iterator();
            groupit.hasNext(); ){
        group g = groupit.next();
        Set<workframe> tempWorkframes = new HashSet<workframe>
            (g.getWorkframes());
        Set<thoughtframe> tempThoughtframes = new HashSet
            <thoughtframe>(g.getThoughtframes());
        for (Iterator<thoughtframe> tfit = tempThoughtframes.
                iterator(); tfit.hasNext(); ){
            thoughtframe tf = tfit.next();
            thoughtframeNames.add(tf.getName());
        }
        for (Iterator<workframe> wfit = tempWorkframes.iterator();
                wfit.hasNext(); ){
            workframe wf = wfit.next();
            workframeNames.add(wf.getName());
        }
    }
    for (Iterator<b_class> classit = classes.iterator();
            classit.hasNext(); ){
        b_class c = classit.next();
        Set<workframe> tempWorkframes = new HashSet<workframe>
            (c.getWorkframes());
        Set<thoughtframe> tempThoughtframes = new HashSet
                <thoughtframe>(c.getThoughtframes());
        for (Iterator<thoughtframe> tfit = tempThoughtframes.
                iterator(); tfit.hasNext(); ){
            thoughtframe tf = tfit.next();
            thoughtframeNames.add(tf.getName());
        }
        for (Iterator<workframe> wfit = tempWorkframes.
                iterator(); wfit.hasNext(); ){
            workframe wf = wfit.next();
            workframeNames.add(wf.getName());
        }

    }
    for (Iterator<String> tfNameIT = thoughtframeNames.iterator();
            tfNameIT.hasNext(); ){
        String n = tfNameIT.next();
        promelaCode = promelaCode.concat("    ," + n + "\n");

    }
    for (Iterator<String> wfNameIT = workframeNames.iterator();
            wfNameIT.hasNext(); ){
        String n = wfNameIT.next();
        promelaCode = promelaCode.concat("    ," + n + "\n");

    }
    promelaCode = promelaCode.concat("}\n\n");
    promelaCode = promelaCode.concat("mtype activeDetectableType
        = null;\n");
    promelaCode = promelaCode.concat("int activeDetectableID;\n");
    promelaCode = promelaCode.concat("mtype turn=Environment;\n");
    // holds the value of lowest time for each agent
    promelaCode = promelaCode.concat("int lowest = -1;\n");
    // the name of the agent
    promelaCode = promelaCode.concat("mtype theLowest;\n");

    /*if(numberOfEverything < 15)
        maxDepth = 15;
    else
        maxDepth = numberOfEverything+1;*/
    maxDepth = numberOfEverything;
    maxVar = 0;

    /*Counters to count number of workframes and thoughtframes,
    used for ID numbers*/
    int wfNum = 0;
    int tfNum = 0;
    /*End of these counters*/

    for (Iterator<agent> MASit = agents.iterator();
            MASit.hasNext(); ){
```

```
        agent ag = MASit.next();
        Set<workframe> tempWorkframes = new HashSet<workframe>
            (ag.getWorkframes());
        Set<thoughtframe> tempThoughtframes = new HashSet
            <thoughtframe>(ag.getThoughtframes());
        for (Iterator<workframe> wfit = tempWorkframes.iterator();
                wfit.hasNext(); ){
            wfNum++;
            workframe wf = wfit.next();
            Set<event> tempEvents = new HashSet<event>
                (wf.getEvents());
            Set<variable> tempVars = new HashSet<variable>
                (wf.getVariables());
            if(tempVars.size() > maxVar)
                maxVar = tempVars.size();
            if(tempEvents.size() > maxDepth)
                maxDepth = tempEvents.size();
        }
        for (Iterator<thoughtframe> tfit = tempThoughtframes.
                iterator(); tfit.hasNext(); ){
            tfNum++;
            thoughtframe tf = tfit.next();
            Set<variable> tempVars = new HashSet<variable>(tf.
                getVariables());
            if(tempVars.size() > maxVar)
                maxVar = tempVars.size();
            Set<event> tempEvents = new HashSet<event>
                (tf.getEvents());
            if(tempEvents.size() > maxDepth)
                maxDepth = tempEvents.size();
        }
    }
    for (Iterator<object> MOSit = objects.iterator();
            MOSit.hasNext(); ){
        object ob = MOSit.next();
        Set<workframe> tempWorkframes = new HashSet<workframe>
            (ob.getWorkframes());
        Set<thoughtframe> tempThoughtframes = new HashSet
            <thoughtframe>(ob.getThoughtframes());
        for (Iterator<workframe> wfit = tempWorkframes.iterator();
                wfit.hasNext(); ){
            wfNum++;
            workframe wf = wfit.next();
            Set<event> tempEvents = new HashSet<event>
                (wf.getEvents());
            Set<variable> tempVars = new HashSet<variable>
                (wf.getVariables());
            if(tempVars.size() > maxVar)
                maxVar = tempVars.size();
            if(tempEvents.size() > maxDepth)
                maxDepth = tempEvents.size();
        }
        for (Iterator<thoughtframe> tfit =
                tempThoughtframes.iterator();
                tfit.hasNext(); ){
            tfNum++;
            thoughtframe tf = tfit.next();
            Set<variable> tempVars = new HashSet<variable>
                (tf.getVariables());
            if(tempVars.size() > maxVar)
                maxVar = tempVars.size();
            Set<event> tempEvents = new HashSet<event>
                (tf.getEvents());
            if(tempEvents.size() > maxDepth)
                maxDepth = tempEvents.size();
        }
    }
    if(wfNum > 0){
        promelaCode = promelaCode.concat
            ("mtype WorkframeIDs["+wfNum+"];\n");
    }
    if(tfNum > 0){
        promelaCode = promelaCode.concat("
            mtype ThoughtframeIDs["+tfNum+"];\n");
    }
    //Add 6 to cater for header data
    promelaCode = promelaCode.concat("typedef array {int
        elements["+(maxDepth+7)+"]};\n");
    if(maxVar > 0)
        promelaCode = promelaCode.concat("typedef array1
            {mtype var_elements["+maxVar+"]};\n");

    promelaCode = promelaCode.concat("int index2;\n\n");
    if(locs.size() >0){
        promelaCode = promelaCode.concat("/*
            Variables for calculating shortest path*/\n");
        promelaCode = promelaCode.concat("typedef matrix
            {int edges["+ locs.size() +"]};\n");
        promelaCode = promelaCode.concat("matrix adjacency[" +
            locs.size() +"];\n");
        promelaCode = promelaCode.concat("int minDist = 0;\n");
        promelaCode = promelaCode.concat("int dist[" +
            locs.size() +"];\n");
        promelaCode = promelaCode.concat("int visit[" +
            locs.size() +"];\n");
        promelaCode = promelaCode.concat("int current;\n");
        promelaCode = promelaCode.concat("int currentLoc;\n");
        promelaCode = promelaCode.concat("int targetLoc;\n");
        promelaCode = promelaCode.concat("int source;\n");
        promelaCode = promelaCode.concat("int temp;\n");
        promelaCode = promelaCode.concat("int distance;\n\n");
    }
    //  To identify who is in what class/group
    for(Iterator<group> groupit = groups.iterator();
            groupit.hasNext();){
        group g = groupit.next();
        promelaCode = promelaCode.concat("int " + g.getName()+
            "members["+numberOfEverything+"];/*this is it*/\n");
    }
    for(Iterator<b_class> classit = classes.iterator();
            classit.hasNext();){
        b_class c = classit.next();
        promelaCode = promelaCode.concat("int " + c.getName()+
            "members["+numberOfEverything+"];\n");
    }
    for(Iterator<areaDefs> areait = areaDefinitions.iterator();
            areait.hasNext();){
        areaDefs ad = areait.next();
        promelaCode = promelaCode.concat("int " + ad.getName()+
            "members["+numberOfEverything+"];\n");
    }

    //  Loop through all agents and assign them an ID number
    promelaCode = promelaCode.concat("/*Agent ID numbers*/\n");
    for (Iterator<agent> MASit = agents.iterator();
            MASit.hasNext(); ){
        agent ag = MASit.next();
        name = ag.getName();
        identificationNumbers[ag.getID()] = name;
        promelaCode = promelaCode.concat("int " + name+"ID  =
            " + ag.getID() + ";\n");
    }

    promelaCode = promelaCode.concat("\n/*Object ID numbers*/\n");
    //  Loop through all objects and assign them an ID number
    for (Iterator<object> MOSit = objects.iterator();
            MOSit.hasNext(); ){
        object ob = MOSit.next();
        name = ob.getName();
        identificationNumbers[ob.getID()] = name;
        promelaCode = promelaCode.concat("int " + name+"ID  = "
            + ob.getID() + ";\n");
    }

    //int numberOfAgentsObjects = i;
    promelaCode = promelaCode.concat("int numberOfAgentsObjects ="
        + numberOfAgentsObjects + ";\n");
```

```
promelaCode = promelaCode.concat(
    "\n/*Locations ID numbers*/\n");
//  Loop through all Locations and assign them an ID number
for (Iterator<locations> locit = locs.iterator();
        locit.hasNext(); ){
    locations l = locit.next();
    name = l.getName();
    int theID = l.getID() + numberOfAgentsObjects;
    identificationNumbers[theID] = name;
    promelaCode = promelaCode.concat("#define " +
        name+"ID " + theID + "\n");
}

promelaCode = promelaCode.concat("#define locationCount "
    + locs.size() +" \n");
promelaCode = promelaCode.concat("#define numberOfEverything
    locationCount+numberOfAgentsObjects+1\n");
promelaCode = promelaCode.concat("int searchID;
    /*Used to find the ID numbers*/\n");
promelaCode = promelaCode.concat("mtype agentsObjectsIDs[
    "+numberOfEverything+"];/*Array which is searched to
    find ID numbers*/\n\n");

/* Workframe/Thoughtframe variables for agents */
promelaCode = promelaCode.concat("/* Workframe/Thoughtframe
    variables for agents */\n");
for (Iterator<agent> MASit = agents.iterator();
        MASit.hasNext(); ){
    agent ag = MASit.next();
    Set<workframe> tempWorkframes = new HashSet<workframe>
        (ag.getWorkframes());
    Set<thoughtframe> tempThoughtframes = new HashSet
        <thoughtframe>(ag.getThoughtframes());
    relations.addAll(ag.getRelations());
    String tempAg = ag.getName();
    // space needed for all workframes
    wfNum = tempWorkframes.size();
    // space needed for all thoughtframes
    tfNum = tempThoughtframes.size();
    //  Add 3 to allow for up to 3 suspended,
    // should really be *2 but trying to save space!!!
    promelaCode = promelaCode.concat("/* Workframe/Thoughtframe
        variables for " + tempAg + "*/\n");
    promelaCode = promelaCode.concat("array wf"+tempAg +
        "["+(wfNum+8)+"];\n");
    promelaCode = promelaCode.concat("int wfTop"+tempAg +
        "["+(wfNum+8)+"];\n");
    promelaCode = promelaCode.concat("
        int wf"+tempAg + "Index = -1;\n");
```

```
promelaCode = promelaCode.concat("\n");
promelaCode = promelaCode.concat("
    array tf"+tempAg + "["+(tfNum+1)+"];\n");
promelaCode = promelaCode.concat("
    int tfTop"+tempAg + "["+(tfNum+1)+"];\n");
promelaCode = promelaCode.concat("
    int tf"+tempAg + "Index = -1;\n\n");
promelaCode = promelaCode.concat("
    /*Agent "+tempAg+"'s relations*/\n");
Set<relation> tempRelations = new HashSet
    <relation>(ag.getRelations());
for (Iterator<workframe> workit=tempWorkframes.iterator();
        workit.hasNext(); ){
    workframe tempWorkframe = workit.next();
    String tempWF = tempWorkframe.getName();
    promelaCode = promelaCode.concat("/* Workframe "
        + tempWF + "*/\n");
    if(maxVar > 0){
        promelaCode=promelaCode.concat("array1 "+tempAg+"_
        wf_"+tempWF+"_var["+numberOfAgentsObjects+"];\n");
    }
    promelaCode = promelaCode.concat("int "+tempAg+"_wf_
        "+tempWF + "_index = -1;\n");
    promelaCode = promelaCode.concat("\n");
}

for (Iterator<thoughtframe> thoughtit = tempThoughtframes.
        iterator(); thoughtit.hasNext(); ){
    thoughtframe tempThoughtframe = thoughtit.next();
    String tempTF = tempThoughtframe.getName();
    promelaCode = promelaCode.concat("/* Thoughtframe "
        + tempTF + "*/\n");
    if(maxVar > 0)
        promelaCode = promelaCode.concat("array1 "+tempAg
            +"_tf_"+tempTF + "_var["+numberOfAgentsObjects
            +"];\n");
    promelaCode = promelaCode.concat("int "+tempAg+"_tf_"
        +tempTF + "_index = -1; /*Look here!*/\n\n");
}
int maxWfDepth = 0;
int maxTfDepth = 0;
for (Iterator<workframe> wfit = tempWorkframes.iterator();
        wfit.hasNext(); ){
    workframe wf = wfit.next();
    List<event> tempEvents = wf.getEvents();
    if(tempEvents.size() > maxWfDepth){
        maxWfDepth = tempEvents.size();
    }
}
```

```
        for (Iterator<thoughtframe> tfit = tempThoughtframes.
                iterator(); tfit.hasNext(); ){
            thoughtframe tf = tfit.next();
            List<event> tempEvents = tf.getEvents();
            if(tempEvents.size() > maxTfDepth){
                maxTfDepth = tempEvents.size();
            }
        }
        // Add 7 to depth, 1 in case a move/comm activity at
        // bottow and 6 to hold header data
        promelaCode = promelaCode.concat("/* Variables for current
            thoughtframe */\n");
        promelaCode = promelaCode.concat("int tf_stack"+tempAg +
            "["+(maxTfDepth+7)+"];\n");
        promelaCode = promelaCode.concat("int tf_"+tempAg + "
            Top = -1;\n\n");
        promelaCode = promelaCode.concat("/* Variables for
            current workframe */\n");
        promelaCode = promelaCode.concat("int wf_stack"+tempAg +
            "["+(maxWfDepth+7)+"];\n");
        promelaCode = promelaCode.concat("int wf_"+tempAg +
            "Top = -1;\n\n");
    }
    promelaCode =promelaCode.concat("\n/****Workframe/Thoughtframe
        variables for objects ****/\n");
    /* Workframe/Thoughtframe variables for objects */
    for (Iterator<object> MOSit = objects.iterator();
            MOSit.hasNext(); ){
        object ob = MOSit.next();
        Set<workframe> tempWorkframes = new HashSet
            <workframe>(ob.getWorkframes());
        Set<thoughtframe> tempThoughtframes = new HashSet
            <thoughtframe>(ob.getThoughtframes());
        String tempAg = ob.getName();
        // space needed for all workframes
        wfNum = tempWorkframes.size();
        // space needed for all thoughtframes
        tfNum = tempThoughtframes.size();
        promelaCode = promelaCode.concat("/* Workframe/Thoughtframe
            variables for " + tempAg + "*/\n");
        promelaCode = promelaCode.concat("array wf"+tempAg +
            "["+(wfNum+3)+"];\n");
        promelaCode = promelaCode.concat("int wfTop"+tempAg +
            "["+(wfNum+3)+"];\n");
        promelaCode = promelaCode.concat("int wf"+tempAg +
            "Index = -1;\n");
        promelaCode = promelaCode.concat("\n");
        promelaCode = promelaCode.concat("array tf"+tempAg +
            "["+(tfNum+3)+"];\n");
```

```
promelaCode = promelaCode.concat("int tfTop"+tempAg +
    "["+(tfNum+3)+"];\n");
promelaCode = promelaCode.concat("int tf"+tempAg +
    "Index = -1;\n\n");
for (Iterator<workframe> workit=tempWorkframes.iterator();
        workit.hasNext(); ){
    workframe tempWorkframe = workit.next();
    String tempWF = tempWorkframe.getName();
    promelaCode = promelaCode.concat("/*
        Workframe " + tempWF + "*/\n");
    if(maxVar > 0){
        promelaCode = promelaCode.concat("array1 "+tempAg
            +"_wf_"+tempWF + "_var["+maxDepth+"];\n");
    }
    promelaCode = promelaCode.concat("int "+tempAg+"_wf_
        "+tempWF + "_index = -1;\n\n");
}
promelaCode = promelaCode.concat("\n");
for (Iterator<thoughtframe> thoughtit =
        tempThoughtframes.iterator();
        thoughtit.hasNext(); ){
    thoughtframe tempThoughtframe = thoughtit.next();
    String tempTF = tempThoughtframe.getName();
    promelaCode = promelaCode.concat("/* Thoughtframe "
        + tempTF + "*/\n");
    if(maxVar > 0)
        promelaCode = promelaCode.concat("array1 tf"+tempTF
            + "_var["+maxDepth+"];\n");
    promelaCode = promelaCode.concat("int tf"+tempTF +
        "_index = -1;\n\n");
}
int maxWfDepth = 0;
int maxTfDepth = 0;
for (Iterator<workframe> wfit = tempWorkframes.iterator();
        wfit.hasNext(); ){
    workframe wf = wfit.next();
    List<event> tempEvents = wf.getEvents();
    if(tempEvents.size() > maxWfDepth)
        maxWfDepth = tempEvents.size();
}
for (Iterator<thoughtframe> tfit = tempThoughtframes.
        iterator(); tfit.hasNext(); ){
    thoughtframe tf = tfit.next();
    List<event> tempEvents = tf.getEvents();
    if(tempEvents.size() > maxWfDepth)
        maxTfDepth = tempEvents.size();
}
// Add 7 to depth, 1 in case a move/comm activity at
// bottow and 6 to hold header data
```

```
        promelaCode = promelaCode.concat("/* Variables for current
            thoughtframe */\n");
        promelaCode = promelaCode.concat("int tf_stack"+tempAg +
            "["+(maxTfDepth+7)+"];\n");
        promelaCode = promelaCode.concat("int tf_"+tempAg +
            "Top = -1;\n\n");

        promelaCode = promelaCode.concat("/* Variables for current
            workframe */\n");
        promelaCode = promelaCode.concat("int wf_stack"+tempAg +
            "["+(maxWfDepth+7)+"];\n");
        promelaCode = promelaCode.concat("int wf_"+tempAg +
            "Top = -1;\n\n");
    }

    /* Variables to hold names of all agents  */
    promelaCode = promelaCode.concat("/*Variables to hold
            names of all agents*/\n");
    for (Iterator<group> groupit = groups.iterator();
            groupit.hasNext(); ){
        group gr = groupit.next();
        // collect all attributes
        allAttributes.addAll(gr.getAttributes());
        // collect all relations
        allRelations.addAll(gr.getRelations());
        String groupsName = gr.getName();
        promelaCode = promelaCode.concat("int "+gr.getName()+
            "Counter = 0;\n");
        int count = 0;
        for (Iterator<agent> MASit2 = agents.iterator();
                MASit2.hasNext(); ){
            agent ag2 = MASit2.next();
            Set memOf = ag2.getMemberOf();
            count = memOf.size();
        }
    }
    promelaCode = promelaCode.concat("\n");
    /* Variables to hold names of all objects  */
    promelaCode = promelaCode.concat("/*Variables to
        hold names of all objects*/\n");
    for (Iterator<b_class> classit = classes.iterator();
            classit.hasNext(); ){
        b_class cl = classit.next();
        // collect all the attributes
        allAttributes.addAll(cl.getAttributes());
        // collect all relations
        allRelations.addAll(cl.getRelations());
        String className = cl.getName();
        promelaCode = promelaCode.concat("int "+cl.getName()+
            "Counter = 0;\n");
        int count = 0;
        for (Iterator<object> MOSit2 = objects.iterator();
                MOSit2.hasNext(); ){
            object ob2 = MOSit2.next();
            Set memOf = ob2.getMemberOf();
            count = memOf.size();
        }
    }
    /* Variables to hold names of all objects  */
    for (Iterator<areaDefs> areaDefit = areaDefinitions.iterator();
            areaDefit.hasNext(); ){
        areaDefs ad = areaDefit.next();
        promelaCode = promelaCode.concat("int "+ad.getName()+
            "["+locs.size()+"]; \n");
        int counterX = 0;
        promelaCode = promelaCode.concat("int "+ad.getName()+
            "Counter = 0;\n");
        for (Iterator<locations> locsit = locs.iterator();
                locsit.hasNext(); ){
            locations lo = locsit.next();
            if(lo.getInstanceOf().equals(ad.getName())){
                counterX++;
            }
        }
    }

    /*Attributes for objects and agents*/
    promelaCode = promelaCode.concat("/*Agent’s and object’s
        attributes - Need to be global for verification purposes*/
        \n");
    // Temp set of attributes to check for duplications
    Set<String> tempAttFacts = new HashSet<String>();
    promelaCode = promelaCode.concat("
        mtype fact_location" + "[" + numberOfAgentsObjects
        + "];\n"); // fact of locations
    //Cycle through all agents to add their attributes
    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        promelaCode = promelaCode.concat("    /*Agent "+
            ag.getName()+"’s Attributes*/\n");
        promelaCode = promelaCode.concat("    mtype " +
            ag.getName()+ "_" + "location" + "[" +
            numberOfAgentsObjects
            + "];\n");  //Belief on locations
        // Temp set of attributes to check for duplications
        Set<String> tempAtt = new HashSet<String>();
        //Loop through all attributes again
        for (Iterator<attribute> attit = a
                llAttributes.iterator(); attit.hasNext(); ){
            attribute at = attit.next();
            String tempA = at.toPromelaString
                (numberOfAgentsObjects, ag.getName())+ "\n";
            String tempB = at.factToPromelaString
                (numberOfAgentsObjects, ag.getName())+ "\n";

            if(!tempAtt.contains(tempA)){
                tempAtt.add(tempA);
                promelaCode = promelaCode.concat(tempA);
            }
            if(!tempAttFacts.contains(tempB)){
                tempAttFacts.add(tempB);
                promelaCode = promelaCode.concat("/*Fact*/\n");
                promelaCode = promelaCode.concat(tempB);
            }
        }
        promelaCode = promelaCode.concat("
            /*Agent "+ag.getName()+"'s Relations*/\n");
        //Temp set of relations to check for duplications
        Set<String> tempRel = new HashSet<String>();
        for (Iterator<relation> relit = allRelations.iterator();
                relit.hasNext(); ){
            relation rel = relit.next();
            String tempR = "    "+rel.toPromelaString(
                numberOfEverything, ag.getName())+ "\n";
            String tempB = rel.factToPromelaString(
                numberOfEverything, ag.getName())+ "\n";

            if(!tempRel.contains(tempR)){
                tempRel.add(tempR);
                promelaCode = promelaCode.concat(tempR);
            }
            if(!tempAttFacts.contains(tempB)){
                tempAttFacts.add(tempB);
                promelaCode = promelaCode.concat("/*Fact*/\n");
                promelaCode = promelaCode.concat(tempB);
            }
        }
    }
    for (Iterator<object> objectit = objects.iterator();
            objectit.hasNext(); ){
        object ob = objectit.next();
        promelaCode = promelaCode.concat("
            /*Object "+ob.getName()+"'s Attributes*/\n");
        //Belief on locations
        promelaCode = promelaCode.concat("    mtype " +
            ob.getName()+ "_" + "location" + "[" +
            numberOfAgentsObjects + "];\n");
            // Temp set of attributes to check for duplications
        Set<String> tempAtt = new HashSet<String>();
        //Loop through all attributes again
        for (Iterator<attribute> attit = allAttributes.iterator();
                attit.hasNext(); ){
            attribute at = attit.next();
            String tempA = at.toPromelaString(
                numberOfAgentsObjects, ob.getName())+ "\n";
            String tempB = at.factToPromelaString(
                numberOfAgentsObjects, ob.getName())+ "\n";

            if(!tempAtt.contains(tempA)){
                tempAtt.add(tempA);
                promelaCode = promelaCode.concat(tempA);
            }
            if(!tempAttFacts.contains(tempB)){
                tempAttFacts.add(tempB);
                promelaCode = promelaCode.concat("/*Fact*/\n");
                promelaCode = promelaCode.concat(tempB);
            }
        }
        promelaCode = promelaCode.concat("    /*Object "+
            ob.getName()+"'s Relations*/\n");
        //Temp set of relations to check for duplications
        Set<String> tempRel = new HashSet<String>();
        for (Iterator<relation> relit = allRelations.iterator();
                relit.hasNext(); ){
            relation rel = relit.next();
            String tempR = "    "+rel.toPromelaString(
                numberOfEverything, ob.getName())+ "\n";
            String tempB = rel.factToPromelaString(
                numberOfEverything, ob.getName())+ "\n";

            if(!tempRel.contains(tempR)){
                tempRel.add(tempR);
                promelaCode = promelaCode.concat(tempR);
            }
            if(!tempAttFacts.contains(tempB)){
                tempAttFacts.add(tempB);
                promelaCode = promelaCode.concat("/*Fact*/\n");
                promelaCode = promelaCode.concat(tempB);
            }
        }
    }

    promelaCode = promelaCode.concat("active
        proctype proc_Environment(){\n");
    promelaCode = promelaCode.concat("    int i;\n ");
```

```
promelaCode = promelaCode.concat("    int j;\n");
promelaCode = promelaCode.concat("    int k;\n");
promelaCode = promelaCode.concat("d_step{\n");
promelaCode = promelaCode.concat("    printf(\"
    Starting system!\\n\");\n");
promelaCode = promelaCode.concat("    i = 0;\n ");
promelaCode = promelaCode.concat("    j = 0;\n");
promelaCode = promelaCode.concat("    k = 0;\n\n");


/*Creates adjacency matrix*/
if(locs.size() > 0){

    Dijkstra dj = new Dijkstra(locs, adjacencyMatrix);
    int[][] shortestPaths = dj.findShortestPaths();

    for(int i = 0;i<locs.size();i++){
        for(int j = 0;j<locs.size();j++){
            promelaCode = promelaCode.concat("
                adjacency"+"["+i+"]"+ ".edges[" + j + "] = "
                + shortestPaths[i][j] +";\n");
        }
    }
}

/* loops to go through all the objects/agents/locations and
    add them to an array */
promelaCode = promelaCode.concat("
    /*agents added to an array*/\n");
promelaCode = promelaCode.concat("
    printf(\"Initialising Agents\\n\");\n");
for (Iterator<agent> MASit2 = agents.iterator();
        MASit2.hasNext(); ){
    agent ag = MASit2.next();
    String tempAg2 = ag.getName();
    promelaCode = promelaCode.concat("    agentsObjectsIDs
        [" + ag.getID() + "] = " + tempAg2 + ";\n");
}
promelaCode = promelaCode.concat("\n");
promelaCode = promelaCode.concat("
    /*objects added to an array*/\n");
promelaCode = promelaCode.concat("
    printf(\"Initialising Objects\\n\");\n");
for (Iterator<object> MOSit = objects.iterator();
        MOSit.hasNext(); ){
    object ob = MOSit.next();
    String tempOb = ob.getName();
    promelaCode = promelaCode.concat("
        agentsObjectsIDs[" + ob.getID()  + "] = " +
        tempOb + ";\n");
}
promelaCode = promelaCode.concat("\n");
promelaCode = promelaCode.concat("    /*locations added
    to an array*/\n");
promelaCode = promelaCode.concat("    printf(
    \"Initialising Locations\\n\");\n");
for (Iterator<locations> locsit = locs.iterator();
        locsit.hasNext(); ){
    locations loc = locsit.next();
    String tempLoc = loc.getName();
    int theID = numberOfAgentsObjects + loc.getID();
    promelaCode = promelaCode.concat("    agentsObjectsIDs[" +
        theID  + "] = " + tempLoc + ";\n");
    for(Iterator<areaDefs> areait = areaDefinitions.iterator();
            areait.hasNext();){
        areaDefs ad = areait.next();
        if(loc.getInstanceOf().equals(ad.getName())){
            promelaCode = promelaCode.concat("    " +
            ad.getName()+"members["+theID+"] = 1;\n");
        }
    }
}
promelaCode = promelaCode.concat("\n");
/* Filling the multiple arrays which individiualise all the
agents, objects and locations according by class/group*/
/* names of all agents  */
for (Iterator<group> groupit = groups.iterator();
        groupit.hasNext(); ){
    group gr = groupit.next();
    String groupsName = gr.getName();
    int count = 0;
    for (Iterator<agent> MASit2 = agents.iterator();
            MASit2.hasNext(); ){
        agent ag2 = MASit2.next();
        Set memOf = ag2.getMemberOf();
        count = memOf.size();
    }

}

/* names of all objects  */
for (Iterator<b_class> classit = classes.iterator();
        classit.hasNext(); ){
    b_class cl = classit.next();
    String className = cl.getName();
    int count = 0;
    for (Iterator<object> MOSit2 = objects.iterator();
            MOSit2.hasNext(); ){
```

```
            object ob2 = MOSit2.next();
            Set memOf = ob2.getMemberOf();
            count = memOf.size();
        }
    }
    /* Names of locations */
    promelaCode = promelaCode.concat("
        /*Store name of all locations in an array*/\n");
    int count = 0;
    for (Iterator<locations> locsit = locs.iterator();
            locsit.hasNext(); ){
        locations lo = locsit.next();
        String locName = lo.getName();

        for (Iterator<areaDefs> areaDefit = areaDefinitions.
                iterator(); areaDefit.hasNext(); ){
            areaDefs ad = areaDefit.next();
            if(lo.getInstanceOf().equals(ad.getName())){
            }
        }

        count++;

    }


    /* Initialise Thoughtframes and Workframes for agents */
    promelaCode = promelaCode.concat("    /*Initialise
        Thoughtframes and Workframes for agents and objects*/\n");
    promelaCode = promelaCode.concat("
        printf(\"Initialising Thoughtframes and workframes
        for agents\\n\");\n");

    promelaCode = promelaCode.concat(thoughtWorkInit);

    promelaCode = promelaCode.concat("    /*Program Loop:
        Initiating Agents/Objects*/\n");
    promelaCode = promelaCode.concat("    printf(\"Starting
        agents and objects\\n\");\n");
    promelaCode = promelaCode.concat("} \n");
    promelaCode = promelaCode.concat("    do\n");
    promelaCode = promelaCode.concat("    ::(turn == Environment
        && EnvironmentActive == true && (");
    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        promelaCode = promelaCode.concat(ag.getName()+
            "Active == true");
        if(agentit.hasNext())
            promelaCode = promelaCode.concat(" || ");
    }
    for (Iterator<object> objectit = objects.iterator();
            objectit.hasNext(); ){
        object ob = objectit.next();
        promelaCode = promelaCode.concat(" || "+
            ob.getName()+"Active == true");
    }
    promelaCode = promelaCode.concat(") && ");
    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        promelaCode = promelaCode.concat(ag.getName()+
            "_timeRemaining == -1");
        if(agentit.hasNext())
            promelaCode = promelaCode.concat(" && ");
    }
    for (Iterator<object> objectit = objects.iterator();
            objectit.hasNext(); ){
        object ob = objectit.next();
        promela Code = promelaCode.concat(" && "+
            ob.getName()+"_timeRemaining == -1");
    }
    promelaCode = promelaCode.concat(") -> \n");
    promelaCode = promelaCode.concat("        d_step{");
    promelaCode = promelaCode.concat("        printf(\"time =
        %d, \", cntEnvironment);\n");
    for (Iterator<agent> agentit = agents.iterator(); agentit.
            hasNext(); ){
        agent ag = agentit.next();
        promelaCode = promelaCode.concat("
            printf(\"cnt"+ag.getName()+" = %d, \", cnt"+
            ag.getName()+");\n");
    }
    for (Iterator<object> objectit = objects.iterator();
            objectit.hasNext(); ){
        object ob = objectit.next();
        promelaCode = promelaCode.concat("
            printf(\"cnt"+ ob.getName()+" =
            %d, \", cnt"+ob.getName()+");\n");
    }
    //Loop through all agents to initiate them
    promelaCode = promelaCode.concat("
        printf(\"\\n\");\n");
    promelaCode = promelaCode.concat("        }");
    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        promelaCode = promelaCode.concat("
            "+ "if\n");
        promelaCode = promelaCode.concat("        "+ "
            ::(turn == Environment) ->\n");
        promelaCode = promelaCode.concat("
            d_step{\n");
        promelaCode = promelaCode.concat("
            "+ ag.getName() + "Active = true;\n");
        promelaCode = promelaCode.concat("
            "+ "turn = ag_" + ag.getName() + ";\n");
        promelaCode = promelaCode.concat("
            printf(\"turn =  ag_"+ag.getName()+"\\n\");\n");
        promelaCode = promelaCode.concat("            }");
        promelaCode = promelaCode.concat("            "+
            "run proc_" + ag.getName() + "();\n");
        promelaCode = promelaCode.concat("        "+
            "fi;\n\n");
    }
    for (Iterator<object> objectit = objects.iterator();
            objectit.hasNext(); ){
        object ob = objectit.next();

        promelaCode = promelaCode.concat("        "+ "if\n");
        promelaCode = promelaCode.concat("        "+
            "::(turn == Environment) ->\n");
        promelaCode = promelaCode.concat("
            d_step{\n");
        promelaCode = promelaCode.concat("
            "+ ob.getName() + "Active = true;\n");
        promelaCode = promelaCode.concat("
            printf(\"turn = ob_"+ob.getName()+"\\n\");\n");
        promelaCode = promelaCode.concat("
            "+ "turn = ob_" + ob.getName() + ";\n");
        promelaCode = promelaCode.concat("            }");
        promelaCode = promelaCode.concat("
            "+ "run proc_" + ob.getName() + "();\n");
        promelaCode = promelaCode.concat("        "+
            "fi;\n\n");
    }
    promelaCode = promelaCode.concat("    ::(turn ==
        Environment && EnvironmentActive == true && (");
    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        promelaCode = promelaCode.concat(ag.getName()+
            "Active == true");
        if(agentit.hasNext())
            promelaCode = promelaCode.concat(" || ");
    }
```

```
for (Iterator<object> objectit = objects.iterator();
        objectit.hasNext(); ){
    object ob = objectit.next();
    promelaCode = promelaCode.concat(" ||
        "+ ob.getName()+"Active == true");
}
promelaCode = promelaCode.concat(") && (");
for (Iterator<agent> agentit = agents.iterator();
        agentit.hasNext(); ){
    agent ag = agentit.next();
    promelaCode = promelaCode.concat(ag.getName()+
        "_timeRemaining != -1");
    if(agentit.hasNext())
        promelaCode = promelaCode.concat(" || ");
}
for (Iterator<object> objectit = objects.iterator();
        objectit.hasNext(); ){
    object ob = objectit.next();
    promelaCode = promelaCode.concat(" || "+
        ob.getName()+"_timeRemaining != -1");
}
promelaCode = promelaCode.concat(")) -> \n");
promelaCode = promelaCode.concat("    d_step{\n");

promelaCode = promelaCode.concat("        " +
    "lowest = -1;\n");
for (Iterator<agent> agentit = agents.iterator();
        agentit.hasNext(); ){
    agent ag = agentit.next();
    promelaCode = promelaCode.concat("       " + "if\n");
    promelaCode = promelaCode.concat("       " +
        "::("+ag.getName() +"_timeRemaining != -1 &&
        (" + ag.getName() + "_timeRemaining < lowest ||
        lowest == -1))->\n");
    promelaCode = promelaCode.concat("       " + "
        lowest = "+ag.getName() +"_timeRemaining;\n");
    promelaCode = promelaCode.concat("       " + "
        theLowest = "+ag.getName() +";\n");
    promelaCode = promelaCode.concat("       " +
        "::else->\n");
    promelaCode = promelaCode.concat("       " + "
        skip;\n");
    promelaCode = promelaCode.concat("       " +
        "fi;\n");
}
for (Iterator<object> objectit = objects.iterator();
        objectit.hasNext(); ){
    object ob = objectit.next();
    promelaCode = promelaCode.concat("       " + "if\n");
    promelaCode = promelaCode.concat("       " +
        "::("+ob.getName() +"_timeRemaining != -1 && (" +
        ob.getName() + "_timeRemaining < lowest ||
        lowest == -1))->\n");
    promelaCode = promelaCode.concat("       " + "
        lowest = "+ob.getName() +"_timeRemaining;\n");
    promelaCode = promelaCode.concat("       " + "
        theLowest = "+ob.getName() +";\n");
    promelaCode = promelaCode.concat("       " +
        "::else->\n");
    promelaCode = promelaCode.concat("       " + "
        skip;\n");
    promelaCode = promelaCode.concat("       " + "fi;\n");
}
promelaCode = promelaCode.concat("       " +
    "printf(\" %e currently has the lowest time remaining
     of %d\\n\", theLowest, lowest);\n");
promelaCode = promelaCode.concat("       " +
    "printf(\" SO the cntEnvironment =  %d + %d\\n\",
     cntEnvironment, lowest);\n");
promelaCode = promelaCode.concat("       " +
    "cntEnvironment = cntEnvironment + lowest;\n");
promelaCode = promelaCode.concat("    }\n");


for (Iterator<agent> agentit = agents.iterator(); a
        gentit.hasNext(); ){
    agent ag = agentit.next();
    promelaCode = promelaCode.concat("        if\n");
    promelaCode = promelaCode.concat("
        ::(turn == Environment)->\n");
    promelaCode = promelaCode.concat("    d_step{\n");
    promelaCode = promelaCode.concat("        "+ "
        i=0;\n");
    promelaCode = promelaCode.concat("            "+
        "/*Check for deleted workframes*/\n");
    promelaCode = promelaCode.concat("            printf(\"
        Checking for deleted workframes for agent "+
        ag.getName()+"\\n\");\n");
    promelaCode = promelaCode.concat("            "
        + "do\n");
    promelaCode = promelaCode.concat("            "
        + "::(i <= wf"+ ag.getName() +"Index) ->\n");
    promelaCode = promelaCode.concat("                "
        + "if\n");
    promelaCode = promelaCode.concat("                "+
        "::(wf"+ ag.getName() +"[i].elements[3] == 0) ->\n");
    promelaCode = promelaCode.concat("
        printf(\"        Deleting workframe at index %d\\n\",
        i);\n");
    promelaCode = promelaCode.concat("
        "+ "j=0;\n");
    promelaCode = promelaCode.concat("
        "+ "do\n");
    promelaCode = promelaCode.concat("                   "+
        "::(j <= wfTop"+ ag.getName() +"[wf" + ag.getName() +
        "Index]) ->\n");
    promelaCode = promelaCode.concat("
        "+ "wf"+ ag.getName() +"[i].elements[j] = wf" +
        ag.getName() +"[wf" + ag.getName() + "Index].
        elements[j];\n");
    promelaCode = promelaCode.concat("
        "+ "j = j+1;\n");
    promelaCode = promelaCode.concat("                  "
        + "::else ->\n");
    promelaCode = promelaCode.concat("                  "
        + "    wfTop"+ag.getName()+"[i] = wfTop"+ag.getName()
        +"[wf" + ag.getName() + "Index];\n");
    promelaCode = promelaCode.concat("                      "
        + "wf"+ag.getName()+"Index = wf"+ag.getName()+"Index-1
        ;\n");
    promelaCode = promelaCode.concat("                      "+
        "break;\n");
    promelaCode = promelaCode.concat("                   "+
        "od;\n");
    promelaCode = promelaCode.concat("               "+
        "::else ->\n");
    promelaCode = promelaCode.concat("                   "+
        "skip;\n");
    promelaCode = promelaCode.concat("               "+
        "fi;\n");
    promelaCode = promelaCode.concat("               "+
        "i=i+1;\n");
    promelaCode = promelaCode.concat("           "+
        "::else -> \n");
    promelaCode = promelaCode.concat("               "+
        "break;\n");
    promelaCode = promelaCode.concat("           "+
        "od;\n");
    promelaCode = promelaCode.concat("           " +
        ag.getName() + "_timeRemaining = -1;\n");
    promelaCode = promelaCode.concat("
        printf(\"turn = ag_" + ag.getName() + "\\n\");\n");
    promelaCode = promelaCode.concat("           "+
        ag.getName()+"Active = true;\n");
    promelaCode = promelaCode.concat("
        turn = ag_" + ag.getName() + ";\n");
    promelaCode = promelaCode.concat("    }\n");
```

```
    promelaCode = promelaCode.concat("        fi;\n");
}
for (Iterator<object> objectit = objects.iterator();
        objectit.hasNext(); ){
    object ob = objectit.next();
    promelaCode = promelaCode.concat("        if\n");
    promelaCode = promelaCode.concat("
        ::(turn == Environment)->\n");
    promelaCode = promelaCode.concat("    d_step{\n");
    promelaCode = promelaCode.concat("
        i=0;\n");
    promelaCode = promelaCode.concat("
        /*Check for deleted workframes*/\n");
    promelaCode = promelaCode.concat("             printf(\"
        Checking for deleted workframes for agent "+
        ob.getName()+"\\n\");\n");
    promelaCode = promelaCode.concat("            do\n");
    promelaCode = promelaCode.concat("            ::(i <= wf
        "+ ob.getName() +"Index) ->\n");
    promelaCode = promelaCode.concat("                if\n");
    promelaCode = promelaCode.concat("                ::(wf"+
        ob.getName() +"[i].elements[3] == 0) ->\n");
    promelaCode = promelaCode.concat("
        printf(\"Deleting workframe at index %d\\n\", i);\n");
    promelaCode = promelaCode.concat("
        j=0;\n");
    promelaCode = promelaCode.concat("
        do\n");
    promelaCode = promelaCode.concat("
        ::(j <= wfTop"+ ob.getName() +"[wf" + ob.getName()
        + "Index]) ->\n");
    promelaCode = promelaCode.concat("
        wf"+ ob.getName() +"[i].elements[j] = wf" +
        ob.getName() +"[wf" + ob.getName() + "Index]
        .elements[j];\n");
    promelaCode = promelaCode.concat("
        j = j+1;\n");
    promelaCode = promelaCode.concat("
        ::else ->\n");
    promelaCode = promelaCode.concat("
        "+ "wfTop"+ob.getName()+"[i] = wfTop"+ob.getName()+
        "[wf" + ob.getName() + "Index];\n");
    promelaCode = promelaCode.concat("
        wf"+ob.getName()+"Index = wf"+ob.getName()+
        "Index - 1;\n");
    promelaCode = promelaCode.concat("
        break;\n");
    promelaCode = promelaCode.concat("
        od;\n");
        promelaCode = promelaCode.concat("
            ::else ->\n");
        promelaCode = promelaCode.concat("
            skip;\n");
        promelaCode = promelaCode.concat("
            fi;\n");
        promelaCode = promelaCode.concat("
            i=i+1;\n");
        promelaCode = promelaCode.concat("
            ::else -> \n");
        promelaCode = promelaCode.concat("
            break;\n");
        promelaCode = promelaCode.concat("
            od;\n");
        promelaCode = promelaCode.concat("            " +
            ob.getName() + "_timeRemaining = -1;\n");
        promelaCode = promelaCode.concat("
            printf(\"turn = ob_" + ob.getName() + "\\n\");\n");
        promelaCode = promelaCode.concat("            "+
            ob.getName()+"Active = true;");
        promelaCode = promelaCode.concat("
            turn = ob_" + ob.getName() + ";\n");
        promelaCode = promelaCode.concat("    }\n");
        promelaCode = promelaCode.concat("        fi;\n");
        }
        promelaCode = promelaCode.concat("        if
            /*Stops Environment jumping ahead*/\n");
        promelaCode = promelaCode.concat("        ::(turn ==
            Environment)->\n");
        promelaCode = promelaCode.concat("            skip;\n");
        promelaCode = promelaCode.concat("        fi;\n");
        promelaCode = promelaCode.concat("    " +
            "::(turn == Environment && EnvironmentActive
            == true");
    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        promelaCode = promelaCode.concat(" && "+ag.getName()
            + "Active == false");
        }
    for (Iterator<object> objectit = objects.iterator();
            objectit.hasNext(); ){
        object ob = objectit.next();
        promelaCode = promelaCode.concat(" && "+ob.getName() +
            "Active == false");
        }
    promelaCode = promelaCode.concat(") ->\n");
    promelaCode = promelaCode.concat("        " + "d_step{\n");
    promelaCode = promelaCode.concat("        " + "printf(
        \"Issuing terminate command\\n\");\n");
    promelaCode = promelaCode.concat("        " +
        "EnvironmentActive = false;\n");
    promelaCode = promelaCode.concat("        " + "}\n");
    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        promelaCode = promelaCode.concat("        if\n");
        promelaCode = promelaCode.concat("
            ::(turn == Environment)->\n");
        promelaCode = promelaCode.concat("        " +
            "d_step{\n");
        promelaCode = promelaCode.concat("
            printf(\"Passing control to ag_"+ag.getName()+"
                so it can terminate\\n\");\n");
        promelaCode = promelaCode.concat("
            turn = ag_"+ag.getName()+";\n");
        promelaCode = promelaCode.concat("        "
            + "}\n");
        promelaCode = promelaCode.concat("        fi;\n");
        }
    for (Iterator<object> objectit = objects.iterator();
            objectit.hasNext(); ){
        object ob = objectit.next();
        promelaCode = promelaCode.concat("        if\n");
        promelaCode = promelaCode.concat("
            ::(turn == Environment)->\n");
        promelaCode = promelaCode.concat("        " +
            "d_step{\n");
        promelaCode = promelaCode.concat("
            printf(\"Passing control to ob_"+ob.getName()+"
            so it can terminate\\n\");\n");
        promelaCode = promelaCode.concat("
            turn = ob_"+ob.getName()+";\n");
        promelaCode = promelaCode.concat("        " + "}\n");
        promelaCode = promelaCode.concat("        fi;\n");
        }
    promelaCode = promelaCode.concat("        if\n");
    promelaCode = promelaCode.concat("
        ::(turn == Environment)->\n");
    promelaCode = promelaCode.concat("        " + "break;\n");
    promelaCode = promelaCode.concat("        fi;\n");
    promelaCode = promelaCode.concat("    " + "od;\n");
    promelaCode = promelaCode.concat("}\n\n");
    promelaCode = promelaCode.concat("inline findID(name)\n");
    promelaCode = promelaCode.concat("{\n");
    promelaCode = promelaCode.concat("    d_step{\n");
    promelaCode = promelaCode.concat("    index2 = 0;\n");
    promelaCode = promelaCode.concat("    do\n");
```

```
        promelaCode = promelaCode.concat("    ::(index2 <
            numberOfEverything) ->\n");
        promelaCode = promelaCode.concat("        if\n");
        promelaCode = promelaCode.concat("
            ::(agentsObjectsIDs[index2] == name)->\n");
        promelaCode = promelaCode.concat("
            searchID = index2;\n");
        promelaCode = promelaCode.concat("
            index2 = numberOfEverything;\n");
        promelaCode = promelaCode.concat("        ::else ->\n");
        promelaCode = promelaCode.concat("            index2++;\n");
        promelaCode = promelaCode.concat("        fi;\n");
        promelaCode = promelaCode.concat("    ::else ->\n");
        promelaCode = promelaCode.concat("        break;\n");
        promelaCode = promelaCode.concat("    od;\n");
        promelaCode = promelaCode.concat("    }\n");
        promelaCode = promelaCode.concat("}\n");
        for (Iterator<agent> agentit = agents.iterator();
                agentit.hasNext(); ){
            agent ag = agentit.next();
            String tempProm = ag.toPromelaString(agents, objects,
                classes, groups, numberOfAgentsObjects,
                numberOfEverything, locs, areaDefinitions, paths,
                identificationNumbers);
            promelaCode = promelaCode.concat("\n
            /*Code for Agent "+ag.getName()+"*/ \n\n" + tempProm);
            }
        for (Iterator<object> objectit = objects.iterator();
                objectit.hasNext(); ){
            object ob = objectit.next();
            String tempProm = ob.toPromelaString(agents, objects,
                classes, groups, numberOfAgentsObjects,
                numberOfEverything, locs, areaDefinitions, paths,
                identificationNumbers);
            promelaCode=promelaCode.concat("\n/*Code for Object "
            +ob.getName()+"*/ \n\n" + tempProm);
            }

        System.out.println(promelaCode);



    }

}
```

## C.3 Agents

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Holds all data about the agents.  Code is almost identical to objects
except during translation object will react on facts and not beliefs.
In the semantics:
Agent's tuple = <agent, Thoughtframe, Workframe, stage, Befliefs,
Facts, Time, Thoughtframes, Workframes>

Thoughtframe(current thoughtframe), Workframe(current workframe),
Stage(which rules to consider) and Time are not covered in these
data structures because they are purely run time only.
*/

import java.util.*;
class agent
{
    String name; // name of the agent
    String display; // Agents display name
    String cost; // cost of agent
    String timeUnit; //
    String location;  // current location of the agent

    // Which group the agent is a member of, will be changed to a
    // set so agent can be member of multiple groups
    Set<String> memberOf;
    // All relations the agent has.
    Set<relation> relations = new HashSet<relation>();
    // All activites agent has
    Set<activity> activities = new HashSet<activity>();
    // Attributes
    Set<attribute> attributes = new HashSet<attribute>();
    // beliefs
    Set<belief> beliefs = new HashSet<belief>();
    // facts
    Set<fact> facts = new HashSet<fact>();
    // workframes
    Set<workframe> workframes = new HashSet<workframe>();
    // thoughtframes
    Set<thoughtframe> thoughtframes = new HashSet<thoughtframe>();

    /****************
    *For Promela use*
```

```
*****************/
int ID; // An ID number assigned to the agent

// Used to count the number of objects/agents,
// for sizing arrays
Set<agent> agents = new HashSet<agent>();
Set<object> objects = new HashSet<object>();

// Available so agents can access details of the groups they
// are members of.  Should be changed to use inheritance.
Set<b_class> classes = new HashSet<b_class>();
Set<group> groups = new HashSet<group>();

 // Number of all objects and agents, mainly used to
 // declare size of arrays
int numberOfAgentsObjects;
// Number of agents, objects and locations.
// Again for array declaration.
int numberOfEverything;
// Holds all the identification numbers for agents
String identificationNumbers[];
// Detectables to be checked
Set<detectable> agentsDetectables = new HashSet<detectable>();

//  Details of all the locations.  Used to size arrays.
Set<locations> locs = new HashSet<locations>();
Set<areaDefs> areaDefs = new HashSet<areaDefs>();
Set<path> paths = new HashSet<path>();

public String initialisePromela = "";
public String toPromela = "";

public agent()
{ }

public agent(
String new_name,
int new_ID,
Set new_memberOf,
String new_display,
String new_cost,
String new_timeUnit,
String new_location,
Set new_relations,
Set new_activities,
Set new_attributes,
Set new_beliefs,
Set new_facts,
Set new_workframes,
Set new_thoughtframes){

    name = new_name;
    ID = new_ID;
    memberOf = new_memberOf;
    display = new_display;
    cost = new_cost;
    timeUnit = new_timeUnit;
    location = new_location;
    relations = new_relations;
    activities = new_activities;
    attributes = new_attributes;
    beliefs = new_beliefs;
    facts = new_facts;
    workframes = new_workframes;
    thoughtframes = new_thoughtframes;

}

public void inheritance(Set groups){
    for(Iterator<group> groupit = groups.iterator();
            groupit.hasNext();){
        group g = groupit.next();
        g.inheritFromMemberOf(groups);
        if(memberOf.contains(g.getName())){
            Set<String> tempMemberOf = new HashSet<String>
                (g.getMembersOf());
            for(Iterator<String> memIt = tempMemberOf.iterator();
                    memIt.hasNext();){
                String m = memIt.next();
                if(!memberOf.contains(m)){
                    memberOf.add(m);
                    inheritance(groups);
                }
            }
            relations.addAll(g.getRelations());
            activities.addAll(g.getActivities());
            attributes.addAll(g.getAttributes());
            beliefs.addAll(g.getBeliefs());
            facts.addAll(g.getFacts());
            workframes.addAll(g.getWorkframes());
            thoughtframes.addAll(g.getThoughtframes());
        }
    }
}

public String thoughtWorkInitialisation(Set groups){
    inheritance(groups);
    //Loop through all groups agent is a member of
```

```
for(Iterator<String> memIt = memberOf.iterator();
        memIt.hasNext();){
    String m = memIt.next();
    //Set in array that this is a member of the group
    initialisePromela = initialisePromela.concat("    "+
        m+"members["+ID+"] = 1;\n");
}
for(Iterator<workframe> workit = workframes.iterator();
        workit.hasNext();){
    workframe w = workit.next();
    agentsDetectables.addAll(w.getDetectables());
}

initialisePromela = initialisePromela.concat("
    /*Thoughtframes*/\n");
for(Iterator<thoughtframe> thoughtit =
        thoughtframes.iterator(); thoughtit.hasNext();){
    thoughtframe t = thoughtit.next();
    initialisePromela = initialisePromela.concat("
        /*Thoughtframe "+t.getName()+"*/\n");
    initialisePromela = initialisePromela.concat("
        ThoughtframeIDs["+t.getID()+"] = "+t.getName()
        +";\n");
    initialisePromela = initialisePromela.concat("
        tf"+name+"Index++;\n");
    List<event> events = t.getEvents();
    initialisePromela = initialisePromela.concat("
        tfTop"+name+"[tf"+name+"Index] = "+(events.size()+5)
        +";\n");
    // ID of thoughtframe
    initialisePromela = initialisePromela.concat("
        tf"+name+"[tf"+name+"Index].elements[0] = "+
        t.getID() +";\n");
    // Guard condition
    initialisePromela = initialisePromela.concat("
        tf"+name+"[tf"+name+"Index].elements[1] = 0;\n");
    int priority = 10*t.getPriority();
    initialisePromela = initialisePromela.concat("
        tf"+name+"[tf"+name+"Index].elements[2] = "+
        priority+";\n"); // Priority
    initialisePromela = initialisePromela.concat("
        tf"+name+"[tf"+name+"Index].elements[3] = "+
        t.getRepeatValue()+";\n"); // Repeat
    // Comm/Move activity started
    initialisePromela = initialisePromela.concat("
        tf"+name+"[tf"+name+"Index].elements[4] = 0;\n");
    // Impassed
    initialisePromela = initialisePromela.concat("
        tf"+name+"[tf"+name+"Index].elements[5] = -1;\n");
    int i = 0;
    for(Iterator<event> eventit = events.iterator();
            eventit.hasNext();){
        event e = eventit.next();
        initialisePromela = initialisePromela.concat("
            tf"+name+"[tf"+name+"Index].elements["+
            (events.size()+5-i)+"] = "+e.getID()+";\n");
        i++;
    }
}
initialisePromela = initialisePromela.concat("
    /*Workframes*/\n");
for(Iterator<workframe> workit = workframes.iterator();
        workit.hasNext();){
    workframe w = workit.next();
    initialisePromela = initialisePromela.concat("
        /*Workframe "+w.getName()+"*/\n");
    initialisePromela = initialisePromela.concat("
        WorkframeIDs["+w.getID()+"] = "+w.getName() +";\n");
    initialisePromela = initialisePromela.concat("
        wf"+name+"Index++;\n");
    List<event> events = w.getEvents();
    initialisePromela = initialisePromela.concat("
        wfTop"+name+"[wf"+name+"Index] = "+(events.size()
        +5)+";\n");
    initialisePromela = initialisePromela.concat("
        wf"+name+"[wf"+name+"Index].elements[0] = "+
        w.getID()+";\n");
    initialisePromela = initialisePromela.concat("
        wf"+name+"[wf"+name+"Index].elements[1] = 0;\n");
    int priority = 10*w.getPriority();
    initialisePromela = initialisePromela.concat("
        wf"+name+"[wf"+name+"Index].elements[2] = "
        +priority+";\n");
    initialisePromela = initialisePromela.concat("
        wf"+name+"[wf"+name+"Index].elements[3] = "
        +w.getRepeatValue()+";\n");
    initialisePromela = initialisePromela.concat("
        wf"+name+"[wf"+name+"Index].elements[4]
        = 0;\n");
    // Impassed
    initialisePromela = initialisePromela.concat("
        wf"+name+"[wf"+name+"Index].elements[5] = -1;\n");
    int i = 0;
    for(Iterator<event> eventit = events.iterator();
            eventit.hasNext();){
        event e = eventit.next();
        if(e.getType() == eventType.Conc || e.getType()
                == eventType.CommAct || e.getType()
```

```
                    == eventType.Move)
                initialisePromela = initialisePromela.concat("
                    wf"+name+"[wf"+name+"Index].elements["+
                    (events.size()+5-i)+"] = "+e.getID()+";\n");
            if(e.getType() == eventType.PrimAct)
                initialisePromela = initialisePromela.concat("
                    wf"+name+"[wf"+name+"Index].elements["+
                    (events.size()+5-i)+"] = "+e.getDuration()+
                    ";\n");
            i++;
        }
    }

    return initialisePromela;
}

public String toPromelaString(Set new_agents, Set new_objects,
Set new_classes, Set new_groups, int new_numberOfAgentsObjects,
int new_numberOfEverything, Set new_locs, Set new_areaDefs,
Set new_paths, String new_identificationNumbers[]) {

    agents = new_agents;
    objects = new_objects;
    classes = new_classes;
    groups = new_groups;
    numberOfAgentsObjects = new_numberOfAgentsObjects;
    numberOfEverything = new_numberOfEverything;
    locs = new_locs;
    areaDefs = new_areaDefs;
    paths = new_paths;
    identificationNumbers = new_identificationNumbers;
    if(workframes.size() > 0){
        toPromela = toPromela.concat("/*Method to set all
            instances as active if guard condition is met*/\n");
        toPromela = toPromela.concat("inline " + name +
            "wfActive(i, ID) {\n");
        toPromela = toPromela.concat("    d_step{\n");
        toPromela = toPromela.concat("    i = 0;\n");
        toPromela = toPromela.concat("    printf(\"
            Setting workframes named %e, with ID = %d active
            and priotiry %d\\n\", WorkframeIDs[ID], ID, wf"+name+"
            [ID].elements[2]);\n");
        toPromela = toPromela.concat("    do\n");
        toPromela = toPromela.concat("    ::(i <= wf"+name+"
            Index)->\n");
        for (Iterator<workframe> workit = workframes.iterator();
                workit.hasNext(); ){
            workframe wf = workit.next();
        }
```

```
toPromela = toPromela.concat("
    if /*workframe has correct ID and repeat
    = true or once*/\n");
toPromela = toPromela.concat("
    ::(wf"+name+"[i].elements[0] == ID &&
    wf"+name+"[i].elements[3] > 0)->\n");
toPromela = toPromela.concat("
    printf(\"        Active workframe found at index %d
    \\n\", i);\n");
if(agentsDetectables.isEmpty() == false){
    toPromela = toPromela.concat("
        printf(\"            Workframe[%d] has detectables,
        checking if in impasse\\n\", i);\n");
    toPromela = toPromela.concat("
        if /*workframe is in impasse*/\n");
    toPromela = toPromela.concat("
        ::(wf"+name+"[i].elements[5] > -1)->\n");
    toPromela = toPromela.concat("
        printf(\"            workframe is in impasse
        \\n\");\n");
    // Loop through all detectables for agent
    for(Iterator<detectable> adit =
            agentsDetectables.iterator();
            adit.hasNext();){
        detectable d = adit.next();
        // if guard condition is met on detectable,
        // checked wf has same wfID and detectableID
        // as the impasse
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::("+d.toPromelaString(identificationNumbers,
            workframes, name)+" && wf"+name+"[i].
            elements[5] == "+d.getID()+" && wf"+
            name+"[i].elements[0] == "+d.getwfNumber()
            +")->\n");
        toPromela = toPromela.concat("
            printf(\"            Workframe %d is in
            impasse and detectable is active, therefore
            not active\\n\", i);\n");
        toPromela = toPromela.concat("
            wf"+name+"[i].elements[1] = 0;\n");
        toPromela = toPromela.concat("
            ::else->\n");
        // Impasse is resolved so workframe can
        // continue exectuting
        toPromela = toPromela.concat("
            wf"+name+"[i].elements[1] = 1;\n");
        // Set detectable inactive
```

```
            toPromela = toPromela.concat("
                wf"+name+"[i].elements[5] = -1;\n");
            // Amend beliefs to match facts
            toPromela = toPromela.concat("
                " + d.getBeliefUpdateToString() +";\n");
            toPromela = toPromela.concat("
                printf(\"            Impasse resolved,
                workframe can now continue\\n\");\n");
            toPromela = toPromela.concat("
                fi;\n");
        }
        toPromela = toPromela.concat("
            ::else ->\n");
        toPromela = toPromela.concat("
            printf(\"            workframe is not in
            impasse, workframe is set active\\n\");\n");
        // Detectable is not active so workframe exectute
        toPromela = toPromela.concat("
            wf"+name+"[i].elements[1] = 1;\n");
        toPromela = toPromela.concat("            fi;\n");
    }
    else{
        toPromela = toPromela.concat("
            wf"+name+"[i].elements[1] = 1;\n");
    }
    //toPromela = toPromela.concat("
        wf"+name+"[i].elements[1] = 1;\n");
    toPromela = toPromela.concat("
        if /*priority is > current highest*/\n");
    toPromela = toPromela.concat("
        ::(pri <= wf"+name+"[i].elements[2] &&
        wf"+name+"[i].elements[1] == 1)->\n");
    toPromela = toPromela.concat("
        pri = wf"+name+"[i].elements[2];\n");
    toPromela = toPromela.concat("
        ::else ->\n");
    toPromela = toPromela.concat("
        skip;\n");
    toPromela = toPromela.concat("
        fi;\n");
    toPromela = toPromela.concat("       ::else->\n");
    toPromela = toPromela.concat("            skip;\n");
    toPromela = toPromela.concat("       fi;\n");
    toPromela = toPromela.concat("       i = i+1;\n");
    toPromela = toPromela.concat("    ::else->\n");
    toPromela = toPromela.concat("       break;\n");
    toPromela = toPromela.concat("    od;\n");
    toPromela = toPromela.concat("    }\n");
    toPromela = toPromela.concat("}\n\n");
```

```
        toPromela = toPromela.concat("inline "+name+"
            wfNotActive(i, ID) {\n");
        toPromela = toPromela.concat("    d_step{\n");
        toPromela = toPromela.concat("    i = 0;\n");
        toPromela = toPromela.concat("    do\n");
        toPromela = toPromela.concat("
            ::(i <= wf"+name+"Index)->\n");
        toPromela = toPromela.concat("        if\n");
        toPromela = toPromela.concat("
            ::(wf"+name+"[i].elements[0] == ID)->\n");
        toPromela = toPromela.concat("
            wf"+name+"[i].elements[1] = 0;\n");
        toPromela = toPromela.concat("        ::else ->\n");
        toPromela = toPromela.concat("            skip;\n");
        toPromela = toPromela.concat("        fi;\n");
        toPromela = toPromela.concat("        i = i+1;\n");
        toPromela = toPromela.concat("    ::else->\n");
        toPromela = toPromela.concat("        break;\n");
        toPromela = toPromela.concat("    od;\n");
        toPromela = toPromela.concat("    }\n");
        toPromela = toPromela.concat("}\n\n");
}
        if(thoughtframes.size() > 0){
        toPromela = toPromela.concat("inline " + name +
            "tfActive(i, ID) {\n");
        toPromela = toPromela.concat("    d_step{\n");
        toPromela = toPromela.concat("    i = 0;\n");
        toPromela = toPromela.concat("    do\n");
        toPromela = toPromela.concat("
            ::(i <= tf"+name+"Index)->\n");
        toPromela = toPromela.concat("
            if /*thoughtframe has correct ID and repeat
            = true or once*/\n");
        toPromela = toPromela.concat("
            ::(tf"+name+"[i].elements[0] == ID && tf"+name+
            "[i].elements[3] > 0)-> \n");
        toPromela = toPromela.concat("
            tf"+name+"[i].elements[1] = 1;
            /*set guard condition to true*/\n");
        toPromela = toPromela.concat("
            if /*priority is > current highest*/\n");
        toPromela = toPromela.concat("
            ::(pri <= tf"+name+"[i].elements[2])->\n");
        toPromela = toPromela.concat("
            pri = tf"+name+"[i].elements[2];\n");
        toPromela = toPromela.concat("            ::else ->\n");
        toPromela = toPromela.concat("                skip;\n");
        toPromela = toPromela.concat("            fi;\n");
        toPromela = toPromela.concat("       ::else->\n");
        toPromela = toPromela.concat("           skip;\n");
        toPromela = toPromela.concat("       fi;\n");
        toPromela = toPromela.concat("       i = i+1;\n");
        toPromela = toPromela.concat("   ::else->\n");
        toPromela = toPromela.concat("       break;\n");
        toPromela = toPromela.concat("    od;\n");
        toPromela = toPromela.concat("    }\n");
        toPromela = toPromela.concat("}\n");
        toPromela = toPromela.concat("
            inline "+name+"tfNotActive(i, ID) {\n");
        toPromela = toPromela.concat("    d_step{\n");
        toPromela = toPromela.concat("    i = 0;\n");
        toPromela = toPromela.concat("    do\n");
        toPromela = toPromela.concat("
            ::(i <= tf"+name+"Index)->\n");
        toPromela = toPromela.concat("        if\n");
        toPromela = toPromela.concat("
            ::(tf"+name+"[i].elements[0] == ID)->\n");
        toPromela = toPromela.concat("
            tf"+name+"[i].elements[1] = 0;\n");
        toPromela = toPromela.concat("        ::else ->\n");
        toPromela = toPromela.concat("            skip;\n");
        toPromela = toPromela.concat("        fi;\n");
        toPromela = toPromela.concat("        i = i+1;\n");
        toPromela = toPromela.concat("    ::else->\n");
        toPromela = toPromela.concat("        break;\n");
        toPromela = toPromela.concat("    od;\n");
        toPromela = toPromela.concat("    }\n");
        toPromela = toPromela.concat("}\n");
    }
    toPromela = toPromela.concat("
        proctype proc_"+name+"() {\n");
    toPromela = toPromela.concat("
        int pri = 0;\n");
    toPromela = toPromela.concat("
        int currentPri = 0;\n");
    toPromela = toPromela.concat("
        bool det_complete = false;\n");
    toPromela = toPromela.concat("
        bool concludes = false;\n");
    toPromela = toPromela.concat("    int i;\n");
    toPromela = toPromela.concat("    int j;\n");
    toPromela = toPromela.concat("    int k;\n\n");
    toPromela = toPromela.concat("\n");
    toPromela = toPromela.concat("    /*Beliefs*/\n");
    if(location != null && !location.equals("")){
        beliefs.add(new belief("current", "location",
            "=", location));
```

```
}
if(beliefs.size() > 0){
    for(Iterator<belief> beliefit = beliefs.iterator();
            beliefit.hasNext();){
        belief b = beliefit.next();
        toPromela = toPromela.concat(
            b.promelaToString(ID, identificationNumbers,
            name));
    }
}
toPromela = toPromela.concat("    /*Facts*/\n");
if(location != null && !location.equals("")){
    facts.add(new fact("current", "location", "=",
        location));
}
if(beliefs.size() > 0){
    for(Iterator<fact> factit = facts.iterator();
            factit.hasNext();){
        fact f = factit.next();
        toPromela = toPromela.concat(
            f.promelaToString(ID, identificationNumbers,
            name)+ "\n");
    }
}

for(Iterator<locations> locit = locs.iterator();
        locit.hasNext();){
    locations l = locit.next();
    toPromela = toPromela.concat("
        bool bool"+l.getName()+";\n");
}
toPromela = toPromela.concat("    bool loop;\n\n");
toPromela = toPromela.concat("    do\n");
toPromela = toPromela.concat("    ::(turn == ag_"+name+" &&
    "+name+"Active == true && EnvironmentActive == true)
    ->\n");
if(thoughtframes.size() > 0) {
    toPromela = toPromela.concat("
        /*********Thoughtframes*********/\n");
    toPromela = toPromela.concat("
        thoughtframes:\n");
    toPromela = toPromela.concat("    d_step{\n");
    toPromela = toPromela.concat("
        printf(\"Checking for active thoughtframes
        \\n\");\n");
    toPromela = toPromela.concat("
        pri = -1;\n");
    for(Iterator<thoughtframe> thoughtit =
            thoughtframes.iterator(); thoughtit.hasNext
            ();){
        thoughtframe t = thoughtit.next();
        toPromela = toPromela.concat(t.toPromelaString
            (identificationNumbers, name)+"\n");
    }
    toPromela = toPromela.concat("
        \n/*Stack is empty, select a thoughtframe*/\n");
    toPromela = toPromela.concat("    }\n");
    toPromela = toPromela.concat("            if\n");
    toPromela = toPromela.concat("            ::(tf_"+name+
        "Top == -1 && "+name+"Active == true) ->\n");
    toPromela = toPromela.concat("
        d_step{\n");
    toPromela = toPromela.concat("
        printf(\"Thoughtframe stack is empty, checking
        for thoughtframes\\n\");\n");
    toPromela = toPromela.concat("
        i = 0;\n");
    toPromela = toPromela.concat("                }\n");
    toPromela = toPromela.concat("                do\n");
    toPromela = toPromela.concat("
        ::(tf"+name+"[i].elements[1] == 1 && i <= tf"+name+
        "Index && tf"+name+"[i].elements[2]  == pri)->\n");
    toPromela = toPromela.concat("
        break;\n");
    toPromela = toPromela.concat("
        ::((tf"+name+"[i].elements[1] != 1 || tf"+name+"[i].
        elements[2]!= pri) && i <= tf"+name+"Index)->\n");
    toPromela = toPromela.concat("
        i = i + 1;\n");
    toPromela = toPromela.concat("
        ::(i > tf"+name+"Index)->\n");
    toPromela = toPromela.concat("
        goto workframes;\n");
    toPromela = toPromela.concat("
        od;\n");
    toPromela = toPromela.concat("                if\n");
    toPromela = toPromela.concat("
        ::(i <= tf"+name+"Index)->\n");
    toPromela = toPromela.concat("    d_step{\n");
    toPromela = toPromela.concat("
        currentPri = pri;\n");
    toPromela = toPromela.concat("
        if\n");
    toPromela = toPromela.concat("
        ::(tf"+name+"[i].elements[3] < 3)->\n");
    toPromela = toPromela.concat("
        tf"+name+"[i].elements[3] = tf"+name+"[i].
        elements[3] - 1;\n");
```

```
toPromela = toPromela.concat("
    printf(\"Thoughtframes repeat variable is reduced
    \\n\");\n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    skip;\n");
toPromela = toPromela.concat("
    printf(\"Thoughtframes repeat variable is to always
    repeat\\n\");\n");
toPromela = toPromela.concat("
    fi;\n");
toPromela = toPromela.concat("
    /*Upload current tf data into current
    thoughtframe*/\n");
toPromela = toPromela.concat("
    printf(\"About to upload tf data into current, TF
    depth = %d\\n\", tfTop"+name+"[i]);\n");
toPromela = toPromela.concat("
    j = 0;\n");
toPromela = toPromela.concat("
    do\n");
toPromela = toPromela.concat("
    ::(j <= tfTop"+name+"[i])->\n");
toPromela = toPromela.concat("
    tf_stack"+name+"[j] = tf"+name+"[i].
    elements[j];\n");
toPromela = toPromela.concat("
    tf_"+name+"Top = tf_"+name+"Top + 1;\n");
toPromela = toPromela.concat("
    printf(\"    adding %d at index %d to current tf
    stack\\n\", tf_stack"+name+"[j], tf_"+name+"Top);
    \n");
toPromela = toPromela.concat("
    j = j+1; \n");
toPromela = toPromela.concat("
    :: else ->\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    od;\n");
toPromela = toPromela.concat("
    printf(\"Thoughtframe %e has been loaded into
    current tf\\n\", ThoughtframeIDs[tf_stack"+name+
    "[0]]);\n");
toPromela = toPromela.concat("
    i = 0;\n");
toPromela = toPromela.concat("
    j = 0;\n");
```

```
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    goto thoughtframes;\n");
toPromela = toPromela.concat("
    :: else ->\n");
toPromela = toPromela.concat("
    goto workframes;\n");
toPromela = toPromela.concat("
    fi;\n");
toPromela = toPromela.concat("
    :: else ->\n");
toPromela = toPromela.concat("
    do\n");
toPromela = toPromela.concat("
    ::(tf_"+name+"Top <= 5);\n");
toPromela = toPromela.concat("
    tf_"+name+"Top = -1;\n");
for(Iterator<thoughtframe> thoughtit = thoughtframes.
    iterator(); thoughtit.hasNext();){
    thoughtframe t = thoughtit.next();
    //Check if the workframe finishing has a varaible,
    //if so decrement the counter
    if(t.getVariables().size() > 0){
        toPromela = toPromela.concat("
            d_step{\n");
        //if it has a collectall then counter needs to
        // be decremented right to the end
        t.hasCollectAll();
        if(t.getContainsCollectAll()){
            toPromela = toPromela.concat("
                do\n");
            toPromela = toPromela.concat("
                ::("+name+"_tf_"+t.getName() +
                "_index > -1)->\n");

            toPromela = toPromela.concat("
                "+name+"_tf_"+t.getName() +
                "_index--;\n");
            toPromela = toPromela.concat("
                ::else->\n");
            toPromela = toPromela.concat("
                break;\n");
            toPromela = toPromela.concat("
                od;\n");
        }
        // if not then just -1
        else{
            toPromela = toPromela.concat("
                if\n");
                toPromela = toPromela.concat("
                    ::(tf_stack"+name+"[0] == "+t.getID()
                    +");\n");
                toPromela = toPromela.concat("
                    "+name+"_tf_"+t.getName() + "_index--;
                    \n");
                toPromela = toPromela.concat("
                    printf(\"Moving to next variable, "+
                    name+"_tf_"+t.getName() + "_index = %d
                    \\n\", "+name+"_tf_"+t.getName() + "
                    _index);\n");
                toPromela = toPromela.concat("
                    ::else->\n");
                toPromela = toPromela.concat("
                    skip;\n");
                toPromela = toPromela.concat("
                    fi;\n");
            }
            toPromela = toPromela.concat("    }\n");
        }
    }
    toPromela = toPromela.concat("
        goto thoughtframes;\n");
    toPromela = toPromela.concat("
        break;\n");
    List<event> agentsTFEvents = new ArrayList();
    for(Iterator<thoughtframe> thoughtit =
            thoughtframes.iterator(); thoughtit.hasNext();){
        thoughtframe t = thoughtit.next();
        t.hasCollectAll();
        agentsTFEvents.addAll(t.getEvents());
    }
    for(Iterator<event> eventit = agentsTFEvents.iterator();
            eventit.hasNext();){
        event e = eventit.next();
        toPromela = toPromela.concat("
            ::(tf_stack"+name+"[tf_"+name+"Top] == "+
            e.getID()+" && tf_"+name+"Top > 5) ->\n");
        String theEvent = e.toPromelaString(name,
            "thoughtframe", agents, objects);
        if(e.getCollectAll() && e.getHasVar() == true){
            toPromela = toPromela.concat("
                tempIndex = "+name+"_tf_"+e.getFName()+
                "_index;\n");
            toPromela = toPromela.concat("
                do\n");
            toPromela = toPromela.concat("
                ::("+name+"_tf_"+e.getFName()+"_index >
                -1)->\n");
            }
            toPromela = toPromela.concat(theEvent);
            if(e.getCollectAll() && e.getHasVar() == true){
                toPromela = toPromela.concat("
                    "+name+"_tf_"+e.getFName()+"_index--;\n");
                toPromela = toPromela.concat("
                    ::else->\n");
                toPromela = toPromela.concat("
                    break;\n");
                toPromela = toPromela.concat("
                    od;\n");
                toPromela = toPromela.concat("
                    skip;\n");
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+name+"_tf_"+e.getFName()+"_index =
                    tempIndex;\n");
                toPromela = toPromela.concat("
                    tempIndex = -1;\n");
                toPromela = toPromela.concat("
                    };\n");
            }
            toPromela = toPromela.concat("
                tf_"+name+"Top--;\n");
            Set updates = e.getBeliefUpdate();
            for(Iterator<String> iter =
                updates.iterator(); iter.hasNext();){
                String up = iter.next();
                }
        }
        toPromela = toPromela.concat("
            :: else ->\n");
        toPromela = toPromela.concat("
            printf(\" Error in processing thoughtframe concludes
            in agent "+name+" \\n \");\n");
        toPromela = toPromela.concat("
            od;\n");
        toPromela = toPromela.concat("
            fi;\n");
    }
    else if(workframes.size() > 0){
        toPromela = toPromela.concat("
            thoughtframes:\n");
        toPromela = toPromela.concat("
            goto workframes;\n");
    }

    if(workframes.size() > 0){
```

```
toPromela = toPromela.concat("
    /*********Workframes*********/\n");
toPromela = toPromela.concat("
    workframes:\n");
toPromela = toPromela.concat("
    /********Find active workframes*******/\n");
toPromela = toPromela.concat("            skip;\n");
toPromela = toPromela.concat("            d_step{\n");
toPromela = toPromela.concat("            printf(
    \"Processing workframes\\n\");\n");
toPromela = toPromela.concat("
    pri = -1; /* reset priority */\n");
toPromela = toPromela.concat("            }\n");
for(Iterator<workframe> workit = workframes.iterator();
        workit.hasNext();){
    toPromela = toPromela.concat("
        d_step{\n");
    workframe w = workit.next();
    toPromela = toPromela.concat(w.toPromelaString(
        identificationNumbers, name, name)+"\n");
    toPromela = toPromela.concat("            }\n");
}
toPromela = toPromela.concat("            d_step{\n");
toPromela = toPromela.concat("
    \n/*Check for suspension*/\n");
toPromela = toPromela.concat("
    printf(\"Checking whether to suspend current
    workframe \\n\");\n");
toPromela = toPromela.concat("            if\n");
toPromela = toPromela.concat("
    ::(wf_"+name+"Top != -1) ->\n");
toPromela = toPromela.concat("
    printf(\"Current pri = %d and pri of highest =
    %d\\n\", wf_stack"+name+"[2], pri);\n");
toPromela = toPromela.concat("                if\n");
toPromela = toPromela.concat("
    ::(pri > (wf_stack"+name+"[2]+3)) ->\n");
toPromela = toPromela.concat("
    printf(\"Suspending current workframe\\n\");\n");
toPromela = toPromela.concat("
    wf"+name+"Index = wf"+name+"Index +1; \n");
toPromela = toPromela.concat("
    i = 0;\n");
toPromela = toPromela.concat("
    printf(\"Adding Elements to set of workframes at
    index %d\\n\", wf"+name+"Index);\n");
toPromela = toPromela.concat("
    do\n");
toPromela = toPromela.concat("
    ::(i <= wf_"+name+"Top) ->\n");
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(i == 2) ->\n"); // If I'm looking at priority
// Set j to value of the priority
toPromela = toPromela.concat("
j = wf_stack"+name+"[i];\n");
// This checks to see if this wf has already been
// suspended
toPromela = toPromela.concat("                    do\n");
toPromela = toPromela.concat("
    ::(j > 0) ->\n");
// Subtract by 10 until j = 0 is < 0.  If < 0
//then it has already been suspended.
toPromela = toPromela.concat("
    j = j - 10;\n");
toPromela = toPromela.concat("
    ::(j == 0) ->\n");
toPromela = toPromela.concat("
    wf"+name+"[wf"+name+"Index].elements[i] =
    wf_stack"+name+"[i] + 2;\n");  // Add 0.2 to priority
toPromela = toPromela.concat("
    printf(\"---> wf"+name+"[%d].elements[%d] = %d\\n\",
    wf"+name+"Index, i,wf"+name+"[wf"+name+"Index].
    elements[i]);\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    printf(\"Workframe has already been suspended!
    \\n\");\n");
toPromela = toPromela.concat("
    printf(\"---> wf"+name+"[%d].elements[%d] = %d\\n\",
    wf"+name+"Index, i,wf"+name+"[wf"+name+"Index].
    elements[i]);\n");
// If < 0 then don't add anything to priority
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    od;\n");
toPromela = toPromela.concat("
    j = 0;\n");
toPromela = toPromela.concat("
    i = i+1;\n");
toPromela = toPromela.concat("
    ::(i == 3) ->\n");
toPromela = toPromela.concat("
    wf"+name+"[wf"+name+"Index].elements[i] = 1;\n");
toPromela = toPromela.concat("
    printf(\"---> wf"+name+"[%d].elements[%d] = %d\\n\",
    wf"+name+"Index, i,wf"+name+"[wf"+name+"Index]
    .elements[i]);\n");
toPromela = toPromela.concat("
    i = i+1;\n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    wf"+name+"[wf"+name+"Index].elements[i] =
    wf_stack"+name+"[i];\n");
toPromela = toPromela.concat("
    printf(\"---> wf"+name+"[%d].elements[%d] = %d\\n\",
    wf"+name+"Index, i,wf"+name+"[wf"+name+
    "Index].elements[i]);\n");
toPromela = toPromela.concat("
    i = i+1;\n");
toPromela = toPromela.concat("
    fi;\n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    od;\n");
toPromela = toPromela.concat("
    wfTop"+name+"[wf"+name+"Index] =
    wf_"+name+"Top;\n");
toPromela = toPromela.concat("
    wf_"+name+"Top = -1;\n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    printf(\"No need to suspend current\\n\");\n");
toPromela = toPromela.concat("
    skip;\n");
toPromela = toPromela.concat("
    fi;\n");
toPromela = toPromela.concat("
    i = 0;\n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    printf(\"No current workframe to suspend\\n\");\n");
toPromela = toPromela.concat("
    skip;\n");
toPromela = toPromela.concat("
    fi;\n");
```

```
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("processWorkframes:\n");
toPromela = toPromela.concat("
    \n/*Stack is empty, select a workframe*/\n");
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(wf_"+name+"Top == -1 &&
    "+name+"Active == true) ->\n");
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    printf(\"Stack is empty,
    selecting workframes\\n\");\n");
toPromela = toPromela.concat("
    i = 0;\n");
toPromela = toPromela.concat("
    j = 0;\n");
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("
    do\n");
toPromela = toPromela.concat("
    ::(wf"+name+"[i].elements[1] == 1 && i <=
    wf"+name+"Index && wf"+name+"[i].
    elements[2] == pri)->\n");
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    i = i + 1;\n");
toPromela = toPromela.concat("
    j = j + 1;\n");
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("
    ::((wf"+name+"[i].elements[1] != 1 ||
    wf"+name+"[i].elements[2] != pri) &&
    i <= wf"+name+"Index)->\n");
toPromela = toPromela.concat("
    i = i + 1;\n");
toPromela = toPromela.concat("
    ::(i > wf"+name+"Index)->\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    od;\n");
toPromela = toPromela.concat("
    i = 0;\n");
```

```
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    k = 1;\n");
toPromela = toPromela.concat("
    do\n");
toPromela = toPromela.concat("
    ::(wf"+name+"[i].elements[1] == 1 && i <=
    wf"+name+"Index && wf"+name+"[i].
    elements[2] == pri)->\n");
toPromela = toPromela.concat("
    printf(\"Uploading workframe %e data from workframe
    at index %d\\n\", WorkframeIDs[wf"+name+"[i].
    elements[0]], i);\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    ::(wf"+name+"[i].elements[1] == 1 && i <= wf"+name
    +"Index && wf"+name+"[i].elements[2] == pri &&
    k < j)->\n");
toPromela = toPromela.concat("
    i = i + 1;\n");
toPromela = toPromela.concat("
    k = k + 1;\n");
toPromela = toPromela.concat("
    ::((wf"+name+"[i].elements[1] != 1 || wf"+name+"[i].
    elements[2] != pri) && i <= wf"+name+"Index)->\n");
toPromela = toPromela.concat("
    i = i + 1;\n");
toPromela = toPromela.concat("
    ::(i > wf"+name+"Index)->\n");
toPromela = toPromela.concat("
    cnt"+name+" = cntEnvironment;\n");
toPromela = toPromela.concat("
    printf(\"No active workframes found, setting time to
    match environment's: cnt"+name+" = %d\\n\", cnt"+
    name+");\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    od;\n");
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(i <= wf"+name+"Index)->\n");
toPromela = toPromela.concat("
    d_step{\n");
```

```
toPromela = toPromela.concat("
    currentPri = pri;\n");
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(wf"+name+"[i].elements[3] < 3)->\n");
toPromela = toPromela.concat("
    wf"+name+"[i].elements[3] = wf"+name+"[i].
        elements[3] - 1;\n");
toPromela = toPromela.concat("
    printf(\"Repeat variable reduced, is now
    repeat = %d\\n\", wf"+name+"[i].elements[3]);
    \n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    skip;\n");
toPromela = toPromela.concat("
    fi;\n");
toPromela = toPromela.concat("
    /*Upload current wf data into current workframe*/
    \n");
toPromela = toPromela.concat("
    j=0;\n");
toPromela = toPromela.concat("
    do\n");
toPromela = toPromela.concat("
    ::(j <= wfTop"+name+"[i])->\n");
toPromela = toPromela.concat("
    wf_stack"+name+"[j] = wf"+name+"[i].elements[j];\n");
toPromela = toPromela.concat("
    wf_"+name+"Top = wf_"+name+"Top + 1;\n");
toPromela = toPromela.concat("
    printf(\"    adding %d at index %d in current wf
    stack\\n\", wf_stack"+name+"[j], wf_"+name+"Top);\n");
toPromela = toPromela.concat("
    j = j+1; \n");
toPromela = toPromela.concat("
    :: else ->\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    od;\n");
toPromela = toPromela.concat("
    i = 0;\n");
toPromela = toPromela.concat("
    j = 0;\n");
toPromela = toPromela.concat("
    printf(\"Workframe %e uploaded, back to start of
    processing\\n\", WorkframeIDs[wf_stack"+name+"
    [0]]);\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    goto processWorkframes;\n");
toPromela = toPromela.concat("
    :: else ->\n");
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    "+name+"Active = false;\n");
toPromela = toPromela.concat("
    printf(\"turn = Environment\\n\");\n");
toPromela = toPromela.concat("
    turn = Environment;\n");
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("
    fi;\n");
toPromela = toPromela.concat("
    ::(wf_"+name+"Top != -1 && "+name+"Active == true)
    ->\n");
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    printf(\"A current workframe found and concludes
    = %d\\n\", concludes);\n");
toPromela = toPromela.concat("
    i = 0;\n");
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(turn == ag_"+name+")->\n");
toPromela = toPromela.concat("
    i = 0;\n");
toPromela = toPromela.concat("
    popstack:\n");
toPromela = toPromela.concat("
    do\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[wf_"+name+"Top] == 0 &&
    wf_"+name+"Top > 5 && comm" + name + " ==
    false);\n");
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    wf_"+name+"Top--;\n");
```

```
toPromela = toPromela.concat("
    printf(\"Activity finished, there are now %d
    elements left on the stack next element is %d\\n\"
    ,(wf_"+name+"Top-5), wf_stack"+name+"[wf_"+name+
    "Top]);\n");
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[wf_"+name+"Top] == 0 && wf_"+
    name+"Top > 5 && comm" + name + " == true);\n");
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    wf_"+name+"Top--;\n");
toPromela = toPromela.concat("
    comm"+name+" = false;\n");
toPromela = toPromela.concat("
    concludes = false;\n");
toPromela = toPromela.concat("
    "+name+"_timeRemaining = 0;\n");
toPromela = toPromela.concat("
    printf(\""+name+"_timeRemaining set to 0\\n\");\n");
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[wf_"+name+"Top] >= 1 &&
        det_complete == false && concludes == false
        && wf_"+name+"Top > 5);\n");
for(Iterator<detectable> adit = agentsDetectables.
        iterator(); adit.hasNext();){
    detectable d = adit.next();
    toPromela = toPromela.concat("    d_step{\n");
    d.toPromelaString(identificationNumbers,workframes,
        name);
    toPromela = toPromela.concat("
    // Different to Object, objects act on facts only.
    /*has Variable = "+d.getHasVariable()+"*/\n");
    if(d.getHasVariable()){
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(wf_stack"+name+"[0] == "+ d.getwfNumber()
            +");\n");
        toPromela = toPromela.concat("
            searchID = 0;\n");
        toPromela = toPromela.concat("
            findID("+d.getTheVariable()+");\n");
            toPromela = toPromela.concat("
                printf(\"Search = %e\\n\",
                agentsObjectsIDs[searchID]);\n");
        }
        // Different to Object, objects act on facts only.
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::("+d.getFactGuardToString()+" && wf_stack"+
            name+"[0] == "+ d.getwfNumber()+")->\n");
        toPromela = toPromela.concat("
            printf(\""+d.getType()+" detectable has
            fired\\n\");\n");
        toPromela = toPromela.concat("
            " + d.getBeliefUpdateToString() +" = "+
            d.getFactUpdateToString()+";\n");
        toPromela = toPromela.concat("
            printf(\"DET BELIEF UPDATE " +
            d.getBeliefUpdateToString() +" = %d (if int) or
            %e (if String)\\n\", "+
            d.getBeliefUpdateToString()+","+
            d.getBeliefUpdateToString()+");\n");
        String tempName;
        if(d.getLeftOwner().equals("current"))
            tempName = name;
        else
            tempName = d.getLeftOwner();
        toPromela = toPromela.concat("
            activeDetectableType = "+d.getType()+";\n");
        toPromela = toPromela.concat("
            activeDetectableID = "+d.getID()+";\n");
        toPromela = toPromela.concat("
            ::else->\n");
        toPromela = toPromela.concat("
            skip;\n");
        toPromela = toPromela.concat("
            fi;\n");
        if(d.getHasVariable()){
            toPromela = toPromela.concat("
                ::else\n");
            toPromela = toPromela.concat("
                skip\n");
            toPromela = toPromela.concat("
                fi;\n");
        }
        toPromela = toPromela.concat("    }\n");
    }
    toPromela = toPromela.concat("
        if\n");
        toPromela = toPromela.concat("
            ::(activeDetectableType == impasse)->\n");
        toPromela = toPromela.concat("    d_step{\n");
        toPromela = toPromela.concat("
            wf_stack"+name+"[5] = activeDetectableID;\n");
        toPromela = toPromela.concat("
            printf(\"Suspending through impasse! Remaining time
            on activity is %d\\n\", wf_stack"+name+"
            [wf_"+name+"Top]);\n");
//Loop through and mark all workframes with this ID as impassed
        toPromela = toPromela.concat("
            i = 0;\n");
        toPromela = toPromela.concat("
            do\n");
        toPromela = toPromela.concat("
            ::(i <= wf"+name+"Index) ->\n");
        toPromela = toPromela.concat("
            if /*workframe has correct ID mark as impassed*/\n");
        toPromela = toPromela.concat("
            ::(wf"+name+"[i].elements[0] == wf_stack"+name
            +"[0])->\n");
        toPromela = toPromela.concat("
            printf(\"Workframe at index %d marked as impassed
            \\n\", i);\n");
        toPromela = toPromela.concat("
            wf"+name+"[i].elements[5] = activeDetectableID;\n");
        toPromela = toPromela.concat("
            ::else->\n");
        toPromela = toPromela.concat("
            skip;\n");
        toPromela = toPromela.concat("
            fi;\n");
        toPromela = toPromela.concat("
            i = i+ 1;\n");
        toPromela = toPromela.concat("
            ::else->\n");
        toPromela = toPromela.concat("
            break;\n");
        toPromela = toPromela.concat("
            od;\n");
        toPromela = toPromela.concat("
            wf"+name+"Index = wf"+name+"Index +1; \n");
        toPromela = toPromela.concat("
            i = 0;\n");
        toPromela = toPromela.concat("
            do\n");
        toPromela = toPromela.concat("
            ::(i <= wf_"+name+"Top) ->\n");
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(i == 2) ->\n"); // If I'm looking at priority
        // Set j to value of the priority
        toPromela = toPromela.concat("
            j = wf_stack"+name+"[i];\n");
// This checks to see if this wf has already been suspended
        toPromela = toPromela.concat("
            do\n");
        toPromela = toPromela.concat("
            ::(j > 0) ->\n");
        // Subtract by 10 until j = 0 is < 0.
        // If < 0 then it has already been suspended.
        toPromela = toPromela.concat("
            j = j - 10;\n");
        toPromela = toPromela.concat("
            ::(j == 0) ->\n");
        toPromela = toPromela.concat("
            wf"+name+"[wf"+name+"Index].elements[i] = wf_stack"+
            name+"[i] + 1;\n");  // Add 0.1 to priority
        toPromela = toPromela.concat("
            printf(\"----> wf"+name+"[%d].elements[%d] = %d
            \\n\",wf"+name+"Index, i,wf"+name+"[wf"+name+"Index]
            .elements[i] );\n");
        toPromela = toPromela.concat("
            break;\n");
        toPromela = toPromela.concat("
            ::else ->\n");
        toPromela = toPromela.concat("
            wf"+name+"[wf"+name+"Index].elements[i] =
            wf_stack"+name+"[i];\n");
        toPromela = toPromela.concat("
            printf(\"----> wf"+name+"[%d].elements[%d] = %d\\n\",
            wf"+name+"Index, i,wf"+name+"[wf"+name+"Index].
            elements[i] );\n");
        // If < 0 then don't add anything to priority
        toPromela = toPromela.concat("
            break;\n");
        toPromela = toPromela.concat("
            od;\n");
        toPromela = toPromela.concat("
            j = 0;\n");
        toPromela = toPromela.concat("
            i = i+1;\n");
        toPromela = toPromela.concat("
            ::(i == 3) ->\n");
        toPromela = toPromela.concat("
            wf"+name+"[wf"+name+"Index].elements[i] = 1;\n");
        toPromela = toPromela.concat("
    printf(\"----> wf"+name+"[%d].elements[%d] = %d\\n
    \",wf"+name+"Index, i,wf"+name+"[wf"+name+"Index]
    .elements[i] );\n");
toPromela = toPromela.concat("
    i = i+1;\n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    wf"+name+"[wf"+name+"Index].elements[i] =
    wf_stack"+name+"[i];\n");
toPromela = toPromela.concat("
    printf(\"----> wf"+name+"[%d].elements[%d] = %d
    \\n\", wf"+name+"Index, i,wf"+name+"[wf"+name+"Index]
    .elements[i] );\n");
toPromela = toPromela.concat("
    i = i+1;\n");
toPromela = toPromela.concat("
    fi;\n");
toPromela = toPromela.concat("
    ::else ->\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    od;\n");
toPromela = toPromela.concat("
    wfTop"+name+"[wf"+name+"Index] = wf_"+name+"Top;\n");
toPromela = toPromela.concat("
    printf(\"Stack is now empty, workframe has been
    suspended through impasse\\n\");\n");
toPromela = toPromela.concat("
    "+name+"_timeRemaining = -1;\n");
toPromela = toPromela.concat("
    det_complete = false;\n");
toPromela = toPromela.concat("
    currentPri = 0;\n");
toPromela = toPromela.concat("
    wf_"+name+"Top = -1;\n");
toPromela = toPromela.concat("
    activeDetectableType = null;\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    goto workframes;\n");
toPromela = toPromela.concat("
    ::(activeDetectableType == continue)->\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    printf(\"A continue detectable fired!\\n\");\n");
toPromela = toPromela.concat("
    "+name+"_timeRemaining = wf_stack"+name+
    "[wf_"+name+"Top];\n");
toPromela = toPromela.concat("
    printf(\""+name+" has an activity of duration %d
    \\n\", wf_stack"+name+"[wf_"+name+"Top]);\n");
toPromela = toPromela.concat("
    cnt"+name+" = cntEnvironment;\n");
toPromela = toPromela.concat("
    printf(\"cnt"+name+" = %d\\n\",cnt"+name+");\n");
toPromela = toPromela.concat("
    printf(\"turn = Environment\\n\");\n");
toPromela = toPromela.concat("
    activeDetectableType = null;\n");
toPromela = toPromela.concat("
    activeDetectableID = -1;\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    ::(activeDetectableType == abort)->\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    wf_"+name+"Top = 5;\n");
toPromela = toPromela.concat("
    printf(\"Aborting workframe through detectable
    \\n\");\n");
toPromela = toPromela.concat("
    activeDetectableType = null;\n");
toPromela = toPromela.concat("
    activeDetectableID = -1;\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    ::(activeDetectableType == complete)->\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    det_complete = true;\n");
toPromela = toPromela.concat("
    activeDetectableType = null;\n");
toPromela = toPromela.concat("
    printf(\"A complete detectable fired!\\n\");\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    ::(activeDetectableType == null)->\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    "+name+"_timeRemaining = wf_stack"+name+
    "[wf_"+name+"Top];\n");
toPromela = toPromela.concat("
    printf(\""+name+" has an activity of duration %d
    \\n\", wf_stack"+name+"[wf_"+name+"Top]);\n");
```

```
toPromela = toPromela.concat("
    cnt"+name+" = cntEnvironment;\n");
toPromela = toPromela.concat("
    printf(\"turn = Environment\\n\");\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    break;\n");
toPromela = toPromela.concat("
    fi;\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[wf_"+name+"Top] >= 1 &&
    det_complete == true && concludes == false &&
    wf_"+name+"Top > 5);\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    printf(\""+name+" is discarding an activity of
    duration %d\\n\", wf_stack"+name+"[wf_"+name+"Top])
    ;\n");
toPromela = toPromela.concat("
    wf_"+name+"Top--;\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[wf_"+name+"Top] >= 1 &&
    det_complete == false && concludes == true &&
    wf_"+name+"Top > 5);\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    printf(\"An activity found when concludes = true,
    breaking.\\n\");\n");
toPromela = toPromela.concat("
    concludes = false;\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    goto processWorkframes;\n");
List<event> agentsWFEvents = new ArrayList();
for(Iterator<workframe> workit = workframes.iterator();
        workit.hasNext();){
    workframe w = workit.next();
    w.hasCollectAll();
    agentsWFEvents.addAll(w.getEvents());
}
for(Iterator<event> eventit = agentsWFEvents.iterator();
        eventit.hasNext();){
    event e = eventit.next();
    if(e.getType() == eventType.CommAct){
        toPromela = toPromela.concat("
            ::(wf_stack"+name+"[wf_"+name+"Top] == "+
            e.getID()+" && wf_"+name+"Top > 5) ->\n");
        String theEvent = e.toPromelaString(name,
            "workframe", agents, objects);
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(wf_stack"+name+"[4] == 0 && concludes
            == true)->\n");
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            printf(\"An activity found when concludes =
            true, breaking.\\n\");\n");
        toPromela = toPromela.concat("
            concludes = false;\n");
        toPromela = toPromela.concat("
            }\n");
        toPromela = toPromela.concat("
            goto processWorkframes;\n");
        toPromela = toPromela.concat("
            ::else->\n");
        toPromela = toPromela.concat("
            skip;\n");
        toPromela = toPromela.concat("
            fi;\n");
        toPromela = toPromela.concat("    d_step{\n");
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(wf_stack"+name+"[4] == 0 && concludes
            == false && "+e.getDuration()+" > 0)->\n");
        toPromela = toPromela.concat("
            wf_"+name+"Top = wf_"+name+"Top+1;\n");
        toPromela = toPromela.concat("
            wf_stack"+name+"[wf_"+name+"Top] = "+
            e.getDuration()+";\n");
        toPromela = toPromela.concat("
            printf(\"Inserting a PA of time %d on to
            the stack\\n\","+e.getDuration()+");\n");
        toPromela = toPromela.concat("
            "+name+"_timeRemaining = wf_stack"+name+"
            [wf_"+name+"Top];\n");
        toPromela = toPromela.concat("
            wf_stack"+name+"[4] = 1;\n");
        toPromela = toPromela.concat("
            ::(wf_stack"+name+"[4] == 1)->\n");
        if(e.getCollectAll()){
            toPromela = toPromela.concat("
                tempIndex = "+name+"_wf_"+e.getFName()+
                "_index;\n");
            toPromela = toPromela.concat("
                do\n");
            toPromela = toPromela.concat("
                ::("+name+"_wf_"+e.getFName()+
                "_index > -1)->\n");
        }
        toPromela = toPromela.concat(theEvent);
        if(e.getCollectAll()){
            toPromela = toPromela.concat("
                "+name+"_wf_"+e.getFName()+
                "_index--;\n");
            toPromela = toPromela.concat("
                ::else->\n");
            toPromela = toPromela.concat("
                break;\n");
            toPromela = toPromela.concat("
                od;\n");
            toPromela = toPromela.concat("
                "+name+"_wf_"+e.getFName()+"_index =
                tempIndex;\n");
            toPromela = toPromela.concat("
                tempIndex = -1;\n");
        }
        toPromela = toPromela.concat("
            wf_stack"+name+"[wf_"+name+"Top] = 0;\n");
        toPromela = toPromela.concat("
            wf_stack"+name+"[4] = 0;\n");
        toPromela = toPromela.concat("
            comm"+name+" = true;\n");
        toPromela = toPromela.concat("
            ::(wf_stack"+name+"[4] == 0 && concludes
            == false && "+e.getDuration()+" == 0)->\n");
        toPromela = toPromela.concat(theEvent);
        toPromela = toPromela.concat("
            wf_"+name+"Top--;\n");
        Set updates = e.getBeliefUpdate();
        for(Iterator<String> iter =
                updates.iterator(); iter.hasNext();){
            String up = iter.next();
            toPromela = toPromela.concat("
                printf(\"There are now %d elements left
                on the stack\\n\",(wf_"+name+"Top-5))
                ;\n");
        }
        toPromela = toPromela.concat("
            fi;\n");
        toPromela = toPromela.concat("    }\n");
    }
    if(e.getType() == eventType.Move){
        toPromela = toPromela.concat("
    ::(wf_stack"+name+"[wf_"+name+"Top] ==
    "+e.getID()+" && wf_"+name+"Top > 5) ->\n");
String theEvent = e.toPromelaString(name,
    "workframe", agents, objects);
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[4] == 0 &&
    concludes == false)->\n");
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    wf_"+name+"Top = wf_"+name+"Top+1;\n");
toPromela = toPromela.concat("
    printf(\"Find ID of %e\\n\","+name+
    "_location["+name+"ID]);\n");
toPromela = toPromela.concat("
    findID("+name+"_location["+name+"ID]);\n");
toPromela = toPromela.concat("
    currentLoc = searchID - "+
    numberOfAgentsObjects+";\n");
toPromela = toPromela.concat("
    printf(\"Find ID of %e\\n\","+
    e.getWhomWhere()+");\n");
toPromela = toPromela.concat("
    findID("+e.getWhomWhere()+");\n");
toPromela = toPromela.concat("
    targetLoc = searchID - "+
    numberOfAgentsObjects+";\n");
toPromela = toPromela.concat("
    printf(\"currentLoc = %d and targetLoc =
    %d\\n\", currentLoc, targetLoc);\n");

toPromela = toPromela.concat("
    wf_stack"+name+"[wf_"+name+"Top] =
    adjacency[currentLoc].edges[targetLoc];\n");
toPromela = toPromela.concat("
    printf(\"Inserting a PA of time %d on to
    the stack\\n\",minDist);\n");
toPromela = toPromela.concat("
    "+name+"_timeRemaining = wf_stack"+name+
    "[wf_"+name+"Top];\n");
toPromela = toPromela.concat("
    wf_stack"+name+"[4] = 1;\n");
toPromela = toPromela.concat("
    }\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[4] == 0 && concludes
    == true)->\n");
```

```
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            printf(\"An activity found when concludes
            = true, breaking.\\n\");\n");
        toPromela = toPromela.concat("
            concludes = false;\n");
        toPromela = toPromela.concat("
            }\n");
        toPromela = toPromela.concat("
            goto processWorkframes;\n");
        toPromela = toPromela.concat("
            ::(wf_stack"+name+"[4] == 1)->\n");
        toPromela = toPromela.concat(theEvent);
        toPromela = toPromela.concat("
            "+e.getToPromelaMove());
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            wf_"+name+"Top--;\n");
        toPromela = toPromela.concat("
            wf_stack"+name+"[4] = 0;\n");
        toPromela = toPromela.concat("
            printf(\"There are now %d elements
            left on the stack\\n\",(wf_"+name+
            "Top-5));\n");
        Set updates = e.getBeliefUpdate();
        for(Iterator<String> iter = updates.iterator();
                iter.hasNext();){
            String up = iter.next();
        }
        Set factUpdates = e.getPromelaFactUpdate();
        toPromela = toPromela.concat("
            }\n");
        for(Iterator<String> iter = factUpdates
                .iterator(); iter.hasNext();){
            String up = iter.next();
        }
        toPromela = toPromela.concat("
            fi;\n");
    }
    if(e.getType() == eventType.Conc){
        toPromela = toPromela.concat("
            ::(wf_stack"+name+"[wf_"+name+"Top] ==
            "+e.getID()+" && wf_"+name+"Top > 5) ->\n");
        if(e.getFc() == 100 && e.getBc() == 100){
            toPromela = toPromela.concat("
                d_step{\n");
        }
        if(e.getCollectAll() && e.getHasVar() == true){
            toPromela = toPromela.concat("
                tempIndex = "+name+"_wf_"+e.getFName()+
                "_index;\n");
            toPromela = toPromela.concat("
                do\n");
            toPromela = toPromela.concat("
                ::("+name+"_wf_"+e.getFName()+"_index
                > -1)->\n");
        }
        String theEvent = e.toPromelaString(name,
            "workframe", agents, objects);
        toPromela = toPromela.concat(theEvent);
        if(e.getCollectAll() && e.getHasVar()
                == true){
            toPromela = toPromela.concat("
                "+name+"_wf_"+e.getFName()+
                "_index--;\n");
            toPromela = toPromela.concat("
                ::else->\n");
            toPromela = toPromela.concat("
                break;\n");
            toPromela = toPromela.concat("
                od;\n");
            toPromela = toPromela.concat("
                skip;\n");
            toPromela = toPromela.concat("
                "+name+"_wf_"+e.getFName()+
                "_index = tempIndex;\n");
            toPromela = toPromela.concat("
                tempIndex = -1;\n");
        }
        toPromela = toPromela.concat("
            wf_"+name+"Top--;\n");
        toPromela = toPromela.concat("
            printf(\"There are now %d elements left
            on the stack\\n\",(wf_"+name+"Top-5));\n");
            if(e.getFc() == 100 & e.getBc() == 100){
            toPromela = toPromela.concat("
                }\n");
        }
    }
}
toPromela = toPromela.concat("
    ::(wf_"+name+"Top <= 5 && concludes == false)
    ->\n");
toPromela = toPromela.concat("
    d_step{\n");
toPromela = toPromela.concat("
    "+name+"_timeRemaining = -1;\n");
for(Iterator<workframe> workit = workframes.iterator();
        workit.hasNext();){
    workframe w = workit.next();
    //Check if the workframe finishing has a varaible,
    //if so decrement the counter
    if(w.getVariables().size() > 0){
        //if it has a collectall then counter needs to
        //be decremented right to the end
        w.hasCollectAll();
        if(w.getContainsCollectAll()){
            toPromela = toPromela.concat("
                do\n");
            toPromela = toPromela.concat("
                ::("+name+"_wf_"+w.getName() +
                "_index > -1)->\n");
            toPromela = toPromela.concat("
                "+name+"_wf_"+w.getName() +
                "_index--;\n");
            toPromela = toPromela.concat("
                ::else->\n");
            toPromela = toPromela.concat("
                break;\n");
            toPromela = toPromela.concat("
                od;\n");

        }
        // if not then just -1
        else{
            toPromela = toPromela.concat("
                if\n");
            toPromela = toPromela.concat("
                ::(wf_stack"+name+"[0] == "+
                w.getID()+");\n");
            toPromela = toPromela.concat("
                "+name+"_wf_"+w.getName()
                + "_index--;\n");
            toPromela = toPromela.concat("
                printf(\"Moving to next variable,
                "+name+"_wf_"+w.getName() +
                "_index = %d\\n\", "+name+"_wf_"+
                w.getName() + "_index);\n");
            toPromela = toPromela.concat("
                ::else->\n");
            toPromela = toPromela.concat("
                skip;\n");
            toPromela = toPromela.concat("
                fi;\n");
        }
    }
}
toPromela = toPromela.concat("
    printf(\"Stack is now empty, workframe finished and
    concludes = %d\\n\", concludes);\n");
toPromela = toPromela.concat("
    "+name+"_timeRemaining = -1;\n");
toPromela = toPromela.concat("
    det_complete = false;\n");
toPromela = toPromela.concat("
    concludes = false;\n");
toPromela = toPromela.concat("
    currentPri = 0;\n");
toPromela = toPromela.concat("
    wf_"+name+"Top = -1;\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    goto thoughtframes;\n");
toPromela = toPromela.concat("
    ::(wf_"+name+"Top <= 5 && concludes == true)
    ->\n");
toPromela = toPromela.concat("    d_step{\n");
for(Iterator<workframe> workit = workframes.iterator();
        workit.hasNext();){
    workframe w = workit.next();
    //Check if the workframe finishing has a varaible,
    //if so decrement the counter
    if(w.getVariables().size() > 0){
        //if it has a collectall then counter needs
        //to be decremented right to the end
        w.hasCollectAll();
        if(w.getContainsCollectAll()){
            toPromela = toPromela.concat("
                do\n");
            toPromela = toPromela.concat("
                ::("+name+"_wf_"+w.getName() +
                "_index > -1)->\n");

            toPromela = toPromela.concat("
                "+name+"_wf_"+w.getName() +
                "_index--;\n");
            toPromela = toPromela.concat("
                ::else->\n");
            toPromela = toPromela.concat("
                break;\n");
            toPromela = toPromela.concat("
                od;\n");

        }
        // if not then just -1
        else{
            toPromela = toPromela.concat("
                if\n");
            toPromela = toPromela.concat("
                ::(wf_stack"+name+"[0] == "+
                w.getID()+");\n");
            toPromela = toPromela.concat("
                "+name+"_wf_"+w.getName() +
                "_index--;\n");
            toPromela = toPromela.concat("
                ::else->\n");
            toPromela = toPromela.concat("
                skip;\n");
            toPromela = toPromela.concat("
                fi;\n");
        }
    }
}
toPromela = toPromela.concat("
    printf(\"Stack is now empty, workframe finished
    concludes = %d\\n\", concludes);\n");
toPromela = toPromela.concat("
    "+name+"_timeRemaining = -1;\n");
toPromela = toPromela.concat("
    det_complete = false;\n");
toPromela = toPromela.concat("
    concludes = false;\n");
toPromela = toPromela.concat("
    currentPri = 0;\n");
toPromela = toPromela.concat("
    wf_"+name+"Top = -1;\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    goto thoughtframes;\n");
toPromela = toPromela.concat("
    :: else ->\n");
toPromela = toPromela.concat("
    printf(\" wf_stack"+name+"[wf_"+name+"Top] = %d
    && det_complete = %d && concludes = %d && wf_"+
    name+"Top = %d\\n \", wf_stack"+name+"[wf_"+
    name+"Top], det_complete, concludes, wf_"+name+
    "Top);\n");
toPromela = toPromela.concat("
    printf(\" Error in processing workframe concludes
    at depth %d in agent "+name+" can't find element
    %d\\n \",wf_"+name+"Top, wf_stack"+name+"
    [wf_"+name+"Top]);\n");
toPromela = toPromela.concat("
    od;\n");
toPromela = toPromela.concat("
    printf(\"Setting turn = Environment with "+name+
    "_timeRemaining = %d\\n\", "+name+"_timeRemaining)
    ;\n");
toPromela = toPromela.concat("
    turn = Environment;\n");
toPromela = toPromela.concat("                 if\n");
toPromela = toPromela.concat("
    ::(cnt"+name+" != cntEnvironment && turn ==
    ag_"+name+") ->\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    timeDeduction = cntEnvironment - cnt"+name+";\n");
toPromela = toPromela.concat("
    printf(\"Time to deducted from current activity
    is %d\\n\", timeDeduction);\n");
toPromela = toPromela.concat("
    cnt"+name+" = cntEnvironment;\n");
toPromela = toPromela.concat("
    printf(\"cnt"+name+" = %d\\n\",cnt"+name+");\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(wf_"+name+"Top != -1 && comm" + name + "
    == false)->\n");
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[wf_"+name+"Top] > 0)->\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    new = wf_stack"+name+"[wf_"+name+"Top] - t
    imeDeduction;\n");
toPromela = toPromela.concat("
    printf(\"Activity duration changes from %d to %d
    \\n\", wf_stack"+name+"[wf_"+name+"Top], new);\n");
toPromela = toPromela.concat("
    wf_stack"+name+"[wf_"+name+"Top] = new;\n");
toPromela = toPromela.concat("    }\n");
toPromela = toPromela.concat("
    if\n");
toPromela = toPromela.concat("
    ::(wf_stack"+name+"[wf_"+name+"Top] == 0)->\n");
toPromela = toPromela.concat("    d_step{\n");
toPromela = toPromela.concat("
    printf(\"Activity is finished, returning
    to finish concludes\\n\");\n");
```

```
        toPromela = toPromela.concat("
            concludes = true;\n");
        toPromela = toPromela.concat("    }\n");
        toPromela = toPromela.concat("
            goto popstack;\n");
        toPromela = toPromela.concat("
            ::else->\n");
        toPromela = toPromela.concat("
            goto thoughtframes;\n");
        toPromela = toPromela.concat("
            fi;\n");
        toPromela = toPromela.concat("
            ::else->\n");
        toPromela = toPromela.concat("
            skip;\n");
        toPromela = toPromela.concat("
            fi;\n");
        toPromela = toPromela.concat("
            ::else->\n");
        toPromela = toPromela.concat("
            skip;\n");
        toPromela = toPromela.concat("
            fi;\n");
        toPromela = toPromela.concat("
            ::(cnt"+name+" == cntEnvironment && turn ==
            ag_"+name+") ->\n");
        toPromela = toPromela.concat("
            goto thoughtframes;\n");
        toPromela = toPromela.concat("
            fi;\n");
        toPromela = toPromela.concat("
            printf(\"turn = Environment\\n\");\n");
        toPromela = toPromela.concat("
            turn = Environment;\n");
        toPromela = toPromela.concat("
            fi;\n");
        toPromela = toPromela.concat("
            fi;\n");
    }
    else{
        toPromela = toPromela.concat("
            workframes:\n");
        toPromela = toPromela.concat("
            skip;\n");
        toPromela = toPromela.concat("    d_step{\n");
        toPromela = toPromela.concat("
            "+name+"Active = false;\n");
        toPromela = toPromela.concat("
            printf(\""+name+" is passing control to
                Environment\\n\");\n");
            toPromela = toPromela.concat("
                printf(\"turn = Environment\\n\");\n");
            toPromela = toPromela.concat("
                turn = Environment;\n");
            toPromela = toPromela.concat("    }\n");
        }
        toPromela = toPromela.concat("
            ::(turn == ag_"+name+" && EnvironmentActive ==
            false)->\n");
        boolean andNeeded = false;
        toPromela = toPromela.concat("    d_step{\n");
        toPromela = toPromela.concat("
            printf(\""+name+" has received Terminate
            Command!\\n\");\n");
        toPromela = toPromela.concat("
            printf(\"turn = Environment\\n\");\n");
        toPromela = toPromela.concat("    }\n");
        toPromela = toPromela.concat("        break;\n");
        toPromela = toPromela.concat("    od;\n");
        toPromela = toPromela.concat("    if\n");
        toPromela = toPromela.concat("
            ::(turn == ag_"+name+")->\n");
        toPromela = toPromela.concat("
            turn = Environment;\n");
        toPromela = toPromela.concat("    fi;\n");
        toPromela = toPromela.concat("}\n");
        return toPromela;
    }

    public String getWorkThoughtInitialisation()
    {
        return initialisePromela;
    }
    public String getName()
    {
        return name;
    }
    public int getID()
    {
        return ID;
    }
    public String getDisplay()
    {
        return display;
    }
    public String getCost()
    {
        return cost;
    }
    public String getTimeUnit()
    {
        return timeUnit;
    }
    public String getLocation()
    {
        return location;
    }
    public Set getRelations()
    {
        return relations;
    }
    public Set getActivities()
    {
        return activities;
    }
    public Set getAttributes()
    {
        return attributes;
    }
    public Set getRBeliefs()
    {
        return beliefs;
    }
    public Set getWorkframes()
    {
        return workframes;
    }
    public Set getThoughtframes()
    {
        return thoughtframes;
    }
    public Set getMemberOf()
    {
        return memberOf;
    }

}
```

## C.4 Groups

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/
```

```
/*
Stores all information on the Group of agents.
*/

import java.util.*;
class group
{
    String name; // Name of the group
    String display;
    String cost;
    String timeUnit;
    String location;

    //  All the details group has
    Set<relation> relations = new HashSet<relation>();
    Set<variable> variables = new HashSet<variable>();
    Set<detectable> detectables = new HashSet<detectable>();
    Set<activity> activities = new HashSet<activity>();
    Set<attribute> attributes = new HashSet<attribute>();
    Set<belief> beliefs = new HashSet<belief>();
    Set<fact> facts = new HashSet<fact>();
    Set<guard> guards = new HashSet<guard>();
    Set<conclude> concludes = new HashSet<conclude>();
    Set<workframe> workframes = new HashSet<workframe>();
    Set<thoughtframe> thoughtframes = new HashSet<thoughtframe>();
    Set<String> memberOf = new HashSet<String>();

    public group()
    { }

    public group(
    String new_name,
    String new_display,
    String new_cost,
    String new_timeUnit,
    String new_location,
    Set new_memberOf,
    Set new_relations,
    Set new_activities,
    Set new_attributes,
    Set new_beliefs,
    Set new_facts,
    Set new_workframes,
    Set new_thoughtframes){

        name = new_name;
        display = new_display;
        cost = new_cost;
        timeUnit = new_timeUnit;
        location = new_location;
        memberOf = new_memberOf;
        relations = new_relations;
        activities = new_activities;
        attributes = new_attributes;
        beliefs = new_beliefs;
        facts = new_facts;
        workframes = new_workframes;
        thoughtframes = new_thoughtframes;
    }

    public void inheritFromMemberOf(Set groups){
        for(Iterator<group> groupit = groups.iterator();
                groupit.hasNext();){
            group g = groupit.next();
            if(memberOf.contains(g.getName())){
                Set<String> tempMemberOf = new HashSet<String>
                    (g.getMembersOf());
                for(Iterator<String> memIt =
                        tempMemberOf.iterator();
                        memIt.hasNext();){
                    String m = memIt.next();
                    if(!memberOf.contains(m)){
                        memberOf.add(m);
                        inheritFromMemberOf(groups);
                    }
                }
                relations.addAll(g.getRelations());
                activities.addAll(g.getActivities());
                attributes.addAll(g.getAttributes());
                beliefs.addAll(g.getBeliefs());
                workframes.addAll(g.getWorkframes());
                thoughtframes.addAll(g.getThoughtframes());
            }
        }
    }

    public String getName()
    {
        return name;
    }
    public String getDisplay()
    {
        return display;
    }
    public String getCost()
    {
        return cost;
    }
    public String getTimeUnit()
    {
        return timeUnit;
    }
    public String getLocation()
    {
        return location;
    }
    public Set getRelations()
    {
        return relations;
    }
    public Set getActivities()
    {
        return activities;
    }
    public Set getAttributes()
    {
        return attributes;
    }
    public Set getBeliefs()
    {
        return beliefs;
    }
    public Set getFacts()
    {
        return facts;
    }
    public Set getWorkframes()
    {
        return workframes;
    }
    public Set getThoughtframes()
    {
        return thoughtframes;
    }
    public Set getMembersOf(){
        return memberOf;
    }
}
```

## C.5 Classes

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
```

```
**/

/*
Stores all information on the Class of objects.
*/

import java.util.*;
class b_class
{
    String name; // Name of the class
    String display;
    String cost;
    String timeUnit;
    String location;

    //  All the details class has
    Set<relation> relations = new HashSet<relation>();
    Set<variable> variables = new HashSet<variable>();
    Set<detectable> detectables = new HashSet<detectable>();
    Set<activity> activities = new HashSet<activity>();
    Set<attribute> attributes = new HashSet<attribute>();
    Set<belief> beliefs = new HashSet<belief>();
    Set<fact> facts = new HashSet<fact>();
    Set<guard> guards = new HashSet<guard>();
    Set<conclude> concludes = new HashSet<conclude>();
    Set<workframe> workframes = new HashSet<workframe>();
    Set<thoughtframe> thoughtframes = new HashSet<thoughtframe>();
    Set<String> memberOf = new HashSet<String>();

    public b_class()
    { }

    public b_class(
    String new_name,
    String new_display,
    String new_cost,
    String new_timeUnit,
    String new_location,
    Set new_memberOf,
    Set new_relations,
    Set new_activities,
    Set new_attributes,
    Set new_beliefs,
    Set new_facts,
    Set new_workframes,
    Set new_thoughtframes){

        name = new_name;
        display = new_display;
        cost = new_cost;
        timeUnit = new_timeUnit;
        location = new_location;
        memberOf = new_memberOf;
        relations = new_relations;
        activities = new_activities;
        attributes = new_attributes;
        beliefs = new_beliefs;
        facts = new_facts;
        workframes = new_workframes;
        thoughtframes = new_thoughtframes;
    }

    public void inheritFromMemberOf(Set classes){
        for(Iterator<b_class> classit = classes.iterator();
                classit.hasNext();){
            b_class c = classit.next();
            if(memberOf.contains(c.getName())){
                Set<String> tempMemberOf = new HashSet<String>
                    (c.getMembersOf());
                for(Iterator<String> memIt = tempMemberOf.iterator();
                        memIt.hasNext();){
                    String m = memIt.next();
                    if(!memberOf.contains(m)){
                        memberOf.add(m);
                        inheritFromMemberOf(classes);
                    }
                }
                relations.addAll(c.getRelations());
                activities.addAll(c.getActivities());
                attributes.addAll(c.getAttributes());
                beliefs.addAll(c.getBeliefs());
                workframes.addAll(c.getWorkframes());
                thoughtframes.addAll(c.getThoughtframes());
            }
        }
    }

    public String getName()
    {
        return name;
    }
    public String getDisplay()
    {
        return display;
    }
    public String getCost()
    {
        return cost;
    }
    public String getTimeUnit()
    {
        return timeUnit;
    }
    public String getLocation()
    {
        return location;
    }
    public Set getRelations()
    {
        return relations;
    }
    public Set getActivities()
    {
        return activities;
    }
    public Set getAttributes()
    {
        return attributes;
    }
    public Set getBeliefs()
    {
        return beliefs;
    }
    public Set getFacts()
    {
        return facts;
    }
    public Set getWorkframes()
    {
        return workframes;
    }
    public Set getThoughtframes()
    {
        return thoughtframes;
    }
    public Set getMembersOf(){
        return memberOf;
    }
}
```

## C.6 Attributes

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
```

```
**/

/*This forms the attributes for agents, objects and possibly
areas/area definitions.*/

import java.util.Stack;
class attribute
{

    String name; // Name of the attribute
    String privacy; // public or private etc.
    String type; // e.g. int, String, boolean etc.

    String toPromela = "";

    public attribute(){}

    public attribute(String new_privacy, String new_type,
            String new_name)
    {
        name = new_name;
        privacy = new_privacy;
        type = new_type;
    }

    public String toPromelaString(int numberOfEverything,
            String agentName)
    {

        toPromela = "";

        if(type.equals("boolean"))
            type = "bool";
        if(!type.equals("bool") && !type.equals("int") &&
                !type.equals("String"))
            type = "mtype";

        if(type.equals("String"))
            toPromela = toPromela.concat("    mtype " + agentName
                + "_" + name + "[" + numberOfEverything + "];");
        else
            toPromela = toPromela.concat("    "+type + " " +
                agentName+ "_" +name+ "[" + numberOfEverything +
                "];");

        return toPromela;

    }
    public String factToPromelaString(int numberOfEverything,
            String agentName)
    {

        toPromela = "";

        if(type.equals("boolean"))
            type = "bool";
        if(!type.equals("bool") && !type.equals("int") &&
                !type.equals("String"))
            type = "mtype";

        if(type.equals("String"))
            toPromela = toPromela.concat("    mtype " + "fact_" +
                name + "[" + numberOfEverything + "];");
        else
            toPromela = toPromela.concat("    "+type + " fact_"
                +name+ "[" + numberOfEverything + "];");

        return toPromela;

    }
    public String getPromela(){
        return toPromela;
    }

    public String getName()
    {
        return name;
    }
    public String getPrivacy()
    {
        return privacy;
    }
    public String getType()
    {
        return type;
    }

}
```

## C.7 Relationships

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/
```

```
/*
Details about types of relations which exist.
*/

import java.util.Stack;
class relation
{

    String name; // relation name
    String privacy; // public or private etc.
    String to; // Which object or agent it refers to

    /****************
    *For Promela use*
    *****************/
    String toPromela = "";
    int numberOf;

    public relation(String new_privacy, String new_name,
            String new_to)
    {
        name = new_name;
        privacy = new_privacy;
        to = new_to;
    }

    public String toPromelaString(int numberOfEverything,
            String agentName)
    {
        toPromela = "";

        toPromela = toPromela.concat("array " + agentName+ "_"
            + name + "["+numberOfEverything+"];");

        return toPromela;
    }

    public String factToPromelaString(int numberOfEverything,
            String agentName)
    {
        toPromela = "";

        toPromela = toPromela.concat("array " + "fact_" + name +
            "["+numberOfEverything+"];");

        return toPromela;
    }
    public String getName() {
        return name;
    }

}
```

## C.8 Beliefs

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Beliefs of the agents/objects and possibly even areas.
*/

import java.util.Stack;
class belief
{

String about; // Who the belief is about
String attribute; // the attribute name
 // Used mainly due to relations, if not a relation it is just an "="
String mathSymbol;
String value; // the value of the belief

String toPromela = "";

public belief(String new_about, String new_attribute,
            String new_mathSymbol, String new_value)
{
about = new_about;
attribute = new_attribute;
mathSymbol = new_mathSymbol;
value = new_value;
}

public String promelaToString(int ID, String
            identificationNumbers[], String agentName) {
String tempAbout = "";
String tempValue = "";
if(!about.equals("current")){
for(int i = 0; i < identificationNumbers.length; i++) {
try{
if(identificationNumbers[i].equals(about)){
```

```
ID = i;
i = identificationNumbers.length;
}
}
catch(Exception E){
System.out.println("Found a Null where there
                        shouldn't be!");
}
}
}
else{
tempAbout = about;
about = agentName;
}
if(value.equals("current")){
tempValue = value;
value = agentName;
}
// if involves a relation
if(!mathSymbol.equals("=")&&!mathSymbol.equals("!=")&&
                !mathSymbol.equals("<")&&!mathSymbol.equals(">")&&
                !mathSymbol.equals("<=")&&!mathSymbol.equals(">=")){
toPromela = toPromela.concat(" " + agentName + "_" +
                mathSymbol + "[" + about + "ID].elements["+value+"ID]
                = 1;" );
}
else{
if(attribute != null)
toPromela = toPromela.concat(" " + agentName+ "_" +
                    attribute + "["+ about + "ID] = " + value+ ";");
else
toPromela = toPromela.concat(" "+ agentName+ "_" +
                    attribute + "["+ about + "ID] = " + value+ ";");
}
//reset to current - incase in a group not a single agent
if(tempAbout.equals("current"))
about = tempAbout;
if(tempValue.equals("current"))
value = tempValue;
//return the value
toPromela = toPromela.concat("\n");

return toPromela;
}

}
```

## C.9 Facts

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Facts.
*/

import java.util.Stack;
class fact
{

    String about;  // Who the fact is about
    String attribute; // The attribute it belongs to
    String mathSymbol; // if it is an "=" or a relation
    String value; // the value it holds

    String toPromela = "";

    public fact(String new_about, String new_attribute, String
            new_mathSymbol, String new_value)
    {
        about = new_about;
        attribute = new_attribute;
        mathSymbol = new_mathSymbol;
        value = new_value;
    }

    public String promelaToString(int ID, String
        identificationNumbers[], String agentName) {

    String tempAbout = "";
        String tempValue = "";
        if(!about.equals("current")){
            for(int i = 0; i < identificationNumbers.length; i++) {
                if(identificationNumbers[i].equals(about)){
                    ID = i;
                    i = identificationNumbers.length;
                }
            }
        }
        else{
            tempAbout = about;
            about = agentName;
        }
        if(value.equals("current")){
            tempValue = value;
            value = agentName;
        }
        // if involves a relation
        if(!mathSymbol.equals("=")&&!mathSymbol.equals("!=")&&
                !mathSymbol.equals("<")&&!mathSymbol.equals(">")&&
                !mathSymbol.equals("<=")&&!mathSymbol.equals(">=")){
            toPromela = toPromela.concat(" " + "fact_" + mathSymbol
                + "[" + about + "ID].elements["+value+"ID] = 1;" );
        }
        else{
            if(attribute != null)
                toPromela = toPromela.concat(" " + "fact_" +
                    attribute + "["+ about + "ID] = " + value+ ";");
            else
                toPromela = toPromela.concat(" "+ "fact_" +
                    attribute + "["+ about + "ID] = " + value+ ";");
        }
        //reset to current - incase in a group not a single agent
        if(tempAbout.equals("current"))
            about = tempAbout;
        if(tempValue.equals("current"))
            value = tempValue;

        //return the value


        return toPromela;
    }

}
```

## C.10 Activities

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
This class stores all the activity definitions within the Brahms code.
Activities can be primtive, move or communicate activity.  Conclude
option is also available but not used in this class.
*/
```

```
import java.util.*;

class activity
{
    // Type of event e.g. PrimAct, Conc, CommAct or Move.
    // Defined in event.java
    eventType type;
    String name;  //  Name of activity

     // The parametername and type of parameter accepted.
    String paramType;
    String parameter;
    String paramType2;
    String parameter2;

    // Used for communication or move, this state who the message is
    //for or where the agent is moving to depending on the activity
    String whom_where;
    Set<messages> mess = new HashSet<messages>();
    int duration; // duration of the activity

    public activity(eventType new_type, String new_name,
            String new_paramType, String new_parameter, String
            new_paramType2, String new_parameter2, int new_duration,
            String new_whom_where, Set<messages> new_mess){
        type = new_type;
        name = new_name;
        paramType = new_paramType;
        paramType2 = new_paramType2;
        parameter = new_parameter;
        parameter2 = new_parameter2;
        duration = new_duration;
        whom_where = new_whom_where;
        mess = new_mess;
    }

    public Set getMess(){
        return mess;
    }
    public String getWhomWhere(){
        return whom_where;
    }
     public eventType getType()
     {
        return type;
     }

     public String getName()
     {
        return name;
     }//  Name of activity

     public String getParamType()
     {
        return paramType;
     }
     public String getParamType2()
     {
        return paramType2;
     }

     public String getParameter()
     {
        return parameter;
     }

     public int getDuration()
     {
        return duration;
     }
}
```

## C.11 Events

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

import java.util.Stack;
import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

/*
Event refers to a conclude or an activity call.
*/

enum eventType {PrimAct, Conc, CommAct, Move}

class event
{


    eventType type; // Conclude, primitive activity, communication etc
```

```
// if activity
String name; // name of the variable
int duration; //  Which class of objects or group of agents
// Who it is communicating with or where it is going to,
// if activity is communicate or move
String tempWhom_where;
String whom_where;
String whom_where2; // 2nd parameter
// If variable passed through, which class or group is it
String paramType;
 // If variable passed through, which class or group is it
String paramType2;
// Who the message is about, communication only
String messAbout;
// The attribute the message is concerned about
String messAtt;
// Who the message is about, communication only. NEW VALUE
String messAbout2;
// The attribute the message is concerned about.  NEW VALUE
String messAtt2;
Set<messages> mess;

//Temp fields to change back to previous values
String OldAttributeOwner;
String OldValueOwner;
String OldValueOwner2;
String OldValue;
String OldWhom_where;

// if conclude
int concID; // Promela use
String f_name; // name of the workframe it is from.
// if the workframe it is from contains a collectAll variable
boolean isCollectAll = false;
// Name of who owns the attribute to be updated
String attributeOwner;
String attributeName; // Name of the attribute to be updated
 // If updated value involves another attribute then this
 // stores the owner
String valueOwner;
// if there are two then this holds this second e.g.
// "current.total = Alex.a + Mary.b" then valueOwner2 = Mary
String valueOwner2;
String value; // Used if there is just a single value
String valueAttr; // attribute of valueOwner
String valueAttr2; // attribute of valueOwner2
// +, -, * or /.  Only one is required as Brahms doesn't
// allow a+b+c etc.
```

```
String valueOperator;

// Stores all activities so event can be tied
// to the activity it belongs
Set<activity> activities = new HashSet<activity>();
// Variables can be as a parameter being passed to an activity
Set<variable> variables = new HashSet<variable>();

/****************
*Used in Promela*
*****************/
// Whether or not event passes through a variable
boolean hasVar = false;

// Has an attribute and refers to current agent/object
boolean attributeOwnerCurrent =false;
// Value in conclude is an attribute with an owner
boolean valueOwnerCurrent =false;
// There is a second
boolean valueOwner2Current =false;
// There is a message or destination about "current"
boolean whomWhereCurrent = false;
// There is a message about "current"
boolean messAboutCurrent = false;
// There is a message about "current"
boolean messAbout2Current = false;

String toPromela = "";
Set<String> PromelaBeliefUpdate = new HashSet<String>();
Set<String> PromelaFactUpdate = new HashSet<String>();
String toPromelaMove = "";

int bc; // Belief condition
int fc; // fact condition

public event()
{}
// if activity
public event(int new_concID, String new_f_name, String new_name,
        String new_whom_where, String new_whom_where2,
        int new_duration, Set<activity> new_activities,
        Set<variable> new_variables)
{
    name = new_name;
    f_name = new_f_name;
    whom_where = new_whom_where;
    whom_where2 = new_whom_where2;
    duration = new_duration;
    activities = new_activities;
```

```
variables = new_variables;
concID = new_concID;
//if it has no duration find the acitivty it is associated
//to and find from there
if(duration == 0)
{
    for (Iterator<activity> actIt = activities.iterator();
            actIt.hasNext(); ) {
        activity act = actIt.next();
        if(act.getType() == eventType.PrimAct){
            String temp = act.getName();
            if(temp.equals(name)){
                duration = act.getDuration();
                type = act.getType();
                break;
            }
        }
        if(act.getType() == eventType.CommAct){
            String temp = act.getName();
            if(act.getParamType() != null)
                paramType = act.getParamType();
            if(act.getParamType2() != null)
                paramType2 = act.getParamType2();
            if(temp.equals(name)){
                duration = act.getDuration();
                type = act.getType();
                if(act.getParameter() != null &&
                        act.getWhomWhere() != null){
                    if(!(act.getParameter().
                            equals(act.getWhomWhere()))){
                        whom_where = act.getWhomWhere();
                    }
                }
                else{
                    whom_where = act.getWhomWhere();
                }
                mess = new HashSet<messages>(act.getMess());
                break;
            }
        }
        if(act.getType() == eventType.Move){
            String temp = act.getName();
            if(temp.equals(name)){
                duration = act.getDuration();
                if(act.getParameter() != null &&
                        act.getWhomWhere() != null){
                    if(!(act.getParameter().equals
                            (act.getWhomWhere()))){
                        whom_where = act.getWhomWhere();
                    }
                }
                else{
                    whom_where = act.getWhomWhere();
                }
                type = act.getType();
                break;
            }
        }
    }
}
    OldAttributeOwner = attributeOwner;
    OldValueOwner = valueOwner;
    OldValueOwner2 = valueOwner2;
    OldValue = value;
    OldWhom_where = whom_where;
}

// if conclude
public event(int new_concID, String new_f_name, eventType
        new_type, String new_attributeOwner, String
        new_attributeName, String new_valueOwner, String
        new_valueOwner2, String new_value, String new_valueAttr,
        String new_valueAttr2, String new_valueOperator, int
        new_bc, int new_fc, Set<variable> new_variables)
{
    concID = new_concID;
    f_name = new_f_name;
    type = new_type;
    attributeOwner = new_attributeOwner;
    attributeName = new_attributeName;
    valueOwner = new_valueOwner;
    valueOwner2 = new_valueOwner2;
    value = new_value;
    valueAttr = new_valueAttr;
    valueAttr2 = new_valueAttr2;
    valueOperator = new_valueOperator;
    duration = -1;
    bc = new_bc;
    fc = new_fc;
    variables = new_variables;

    OldAttributeOwner = attributeOwner;
    OldValueOwner = valueOwner;
    OldValueOwner2 = valueOwner2;
    OldValue = value;
    OldWhom_where = whom_where;
}
```

```
public String toPromelaString(String agentName, String frame,
        Set agents, Set objects){
    toPromela = "";
    toPromelaMove = "";
    PromelaBeliefUpdate.clear();
    PromelaFactUpdate.clear();
    boolean hasVariable = false;
    boolean attributeOwnerVariable = false;
    boolean valueOwnerVariable = false;
    boolean valueOwner2Variable = false;

    attributeOwner = OldAttributeOwner;
    valueOwner = OldValueOwner;
    valueOwner2 = OldValueOwner2;
    value = OldValue;
    whom_where = OldWhom_where;
    if(frame.equals("workframe"))
        frame = "_wf_";
    else
        frame = "_tf_";

    for(Iterator<variable> varit = variables.iterator();
            varit.hasNext();){
        variable v = varit.next();
        try{
            if(attributeOwner.equals(v.getName())){
                attributeOwner = "";
                attributeOwner = attributeOwner.concat(
                        agentName+frame+f_name+"_var["+agentName+
                        frame+f_name+"_index].var_elements["+
                        v.getVarNo()+"]");
                attributeOwnerVariable = true;
                hasVar = true;
            }
        }
        catch(Exception e){}
        try{
            if(valueOwner.equals(v.getName())){
                valueOwner = "";
                valueOwner = valueOwner.concat(""+agentName+
                frame+f_name+"_var["+agentName+frame+
                f_name+"_index].var_elements["+v.getVarNo()+"]");
                //System.out.println("/*" + valueOwner + "*/");
                hasVar = true;
                valueOwnerVariable = true;
            }
        }
        catch(Exception e){}
        try{
            if(valueOwner2.equals(v.getName())){
                valueOwner2 = "";
                valueOwner2 = valueOwner2.concat(""+agentName+
                    frame+f_name+"_var["+agentName+frame+
                    f_name+"_index].var_elements["+v.
                    getVarNo()+"]");
                hasVar = true;
                valueOwner2Variable = true;
            }
        }
        catch(Exception e){}

        try{
            if(value.equals(v.getName())){
                value = "";
                value = value.concat(""+agentName+frame+f_name+
                    "_var["+agentName+frame+f_name+"_index].
                    var_elements["+v.getVarNo()+"]");
                hasVar = true;
            }
        }
        catch(Exception e){}

        try{
            if(whom_where.equals(v.getName())){
                hasVariable = true;
                whom_where = "";
                whom_where = whom_where.concat(""+agentName+frame+
                    f_name+"_var["+agentName+frame+f_name+"_index]
                    .var_elements["+v.getVarNo()+"]");
                hasVar = true;
            }
        }
        catch(Exception e){}
    }
    boolean attributeOwnerCurrent =false;
    boolean valueOwnerCurrent =false;
    boolean valueOwner2Current =false;
    if(type == eventType.Conc){
        if(attributeOwner != null){
            if(attributeOwner.equals("current")){
                attributeOwner = agentName;
                attributeOwnerCurrent =true;
            }
        }
        if(valueOwner != null){;
            if(valueOwner.equals("current")){
                valueOwner = agentName;
                valueOwnerCurrent = true;
        }
    }
    if(valueOwner2 != null){
        if(valueOwner2.equals("current")){
            valueOwner2 = agentName;
            valueOwner2Current = true;
        }
    }
    if(valueOwner == null && valueOwner2 == null && value
            != null){
        //same for facts
        if(fc ==100 && frame.equals("_wf_")){
            if(attributeOwnerVariable == false){
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+"fact_"+attributeName + "[" +
                    attributeOwner + "ID] = " + value + ";\n");
                toPromela = toPromela.concat("
                    printf(\"FACT UPDATE1: fact_"+
                    attributeName + "[" + attributeOwner +
                    "ID] = %e (String) or %d (Integar)\\n\",
                    fact_"+attributeName + "[" + attributeOwner
                    + "ID], fact_"+ attributeName + "[" +
                    attributeOwner + "ID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            else{
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+"searchID = 0;\n");
                toPromela = toPromela.concat("
                    "+"findID("+attributeOwner+");\n");
                toPromela = toPromela.concat("
                    "+"fact_"+attributeName + "[searchID]
                        = " + value + ";\n");
                toPromela = toPromela.concat("
                printf(\"Search = %e\\n\",
                    agentsObjectsIDs[searchID]);\n");
                toPromela = toPromela.concat("
                    printf(\"FACT UPDATE2: fact_"+attributeName
                    + "[searchID] = %e (String) or %d (Integar)
                    \\n\", fact_"+attributeName + "[searchID],
                    fact_"+attributeName + "[searchID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
        if(attributeOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"fact_"+attributeName + "[" +
                attributeOwner + "ID] = " + value + ";\n");
            toPromela = toPromela.concat("
                printf(\"FACT UPDATE3: fact_"+attributeName
                + "[" + attributeOwner + "ID] = %e (String)
                or %d (Integar)\\n\", fact_"+attributeName
                + "[" + attributeOwner + "ID], fact_"+
                attributeName + "[" + attributeOwner +
                "ID]);\n");
            toPromela = toPromela.concat("
                }\n");
        }
        else{
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"searchID = 0;\n");
            toPromela = toPromela.concat("
                "+"findID("+attributeOwner+");\n");
            toPromela = toPromela.concat("
                "+"fact_"+attributeName + "[searchID]
                = " + value + ";\n");
            toPromela = toPromela.concat("
                printf(\"Search = %e\\n\",
                agentsObjectsIDs[searchID]);\n");
            toPromela = toPromela.concat("
                printf(\"FACT UPDATE4: fact_"+
                attributeName + "[searchID] = %e
                (String) or %d (Integar)\\n\",
                fact_"+attributeName + "[searchID],
                fact_"+attributeName + "[searchID]);
                \n");
            toPromela = toPromela.concat("
                }\n");
        }
    }
    else if(fc == 0  || frame.equals("_tf_"))
        toPromela = toPromela.concat("
            /*Fact not updated*/\n");
    else if(frame.equals("_wf_")){
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(true)->\n");
        if(attributeOwnerVariable == false){
```

```
                    toPromela = toPromela.concat("
                        d_step{\n");
                    toPromela = toPromela.concat("
                        "+"fact_"+attributeName + "[" +
                        attributeOwner + "ID] = " +
                        value + ";\n");
                    toPromela = toPromela.concat("
                        printf(\"FACT UPDATE5: fact_"+
                        attributeName + "[" + attributeOwner
                        + "ID] = %e (String) or %d (Integer)
                        \\n\", fact_"+attributeName + "[" +
                        attributeOwner + "ID], fact_"+
                        attributeName + "[" + attributeOwner
                        + "ID]);\n");
                    toPromela = toPromela.concat("
                        }\n");
                }
                else{
                    toPromela = toPromela.concat("
                        d_step{\n");
                    toPromela = toPromela.concat("
                        "+"searchID = 0;\n");
                    toPromela = toPromela.concat("
                        "+"findID("+attributeOwner+");\n");
                    toPromela = toPromela.concat("
                        "+"fact_"+attributeName + "[searchID]
                        = " + value + ";\n");
                    toPromela = toPromela.concat("
                        printf(\"FACT UPDATE6: fact_"+attributeName
                        + "[searchID] = %e (String) or %d (Integer)
                        \\n\", fact_"+attributeName + "[searchID],
                        fact_"+attributeName + "[searchID]);\n");
                    toPromela = toPromela.concat("
                        }\n");
                }
                toPromela = toPromela.concat("
                    ::(true)->\n");
                toPromela = toPromela.concat("
                    skip;\n");
                toPromela = toPromela.concat("
                    fi;");
            }
// If belief condition is 100% then just put a straight conclude
            if(bc ==100){
                if(attributeOwnerVariable == false){
                    toPromela = toPromela.concat("
                        d_step{\n");
                    toPromela = toPromela.concat("
                        "+agentName+"_"+attributeName + "[" +
                attributeOwner + "ID] = " + value + ";\n");
            toPromela = toPromela.concat("
                printf(\"BELIEF UPDATE: "+agentName+"_"+
                attributeName + "[" + attributeOwner + "
                ID] = %e (String) or %d (Integar)\\n\", "
                +agentName+"_"+attributeName + "[" +
                attributeOwner + "ID], "+ agentName+"_"+
                attributeName + "[" + attributeOwner +
                "ID]);\n");
            toPromela = toPromela.concat("
            }\n");
        }
        else{
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"searchID = 0;\n");
            toPromela = toPromela.concat("
                "+"findID("+attributeOwner+");\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName +
                "[searchID] = " + value + ";\n");
            toPromela = toPromela.concat("
                printf(\"Search = %e\\n\",
                agentsObjectsIDs[searchID]);\n");
            toPromela = toPromela.concat("
                printf(\"BELIEF UPDATE: "+agentName+
                "_"+attributeName + "[searchID] = %e
                (String) or %d (Integar)\\n\", "+
                agentName+"_"+attributeName + "[searchID],
                " + agentName+"_"+attributeName +
                "[searchID]);\n");
            toPromela = toPromela.concat("
                }\n");
        }
    }
    else if(bc == 0){
        toPromela = toPromela.concat("
            /*Belief not updated*/\n");
    }
    //If not create a split where conclude is not done
    else{
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(true)->\n");
        if(attributeOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
                toPromela = toPromela.concat("
                    " + agentName+"_"+attributeName +
                    "[" + attributeOwner + "ID] = "
                    + value + ";\n");
                toPromela = toPromela.concat("
                    printf(\"BELIEF UPDATE: "+agentName
                    +"_"+attributeName + "[" + attributeOwner
                    + "ID] = %e (String) or %d (Integar)
                    \\n\", "+agentName+"_"+attributeName +
                    "[" + attributeOwner + "ID], "+
                    agentName+"_"+attributeName +
                    "[" + attributeOwner + "ID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            else{
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+"searchID = 0;\n");
                toPromela = toPromela.concat("
                    "+"findID("+attributeOwner+");\n");
                toPromela = toPromela.concat("
                    "+agentName+"_"+attributeName +
                    "[searchID] = " + value + ";\n");
                toPromela = toPromela.concat("
                    printf(\"Search = %e\\n\",
                    agentsObjectsIDs[searchID]);\n");
                toPromela = toPromela.concat("
                    printf(\"BELIEF UPDATE: "+agentName+"_"+
                    attributeName + "[searchID] = %e (String)
                    or %d (Integar)\\n\", "+agentName+"_"+
                    attributeName + "[searchID]," +
                    agentName+"_"+attributeName +
                    "[searchID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            toPromela = toPromela.concat("
                ::(true)->\n");
            toPromela = toPromela.concat("
                skip;\n");
            toPromela = toPromela.concat("
                fi;");
        }
    }

    if(valueOwner != null && valueOwner2 == null
            && value == null){
```

```
//same for facts
if(fc ==100  && frame.equals("_wf_"))
    if(attributeOwnerVariable == false){
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            "+"fact_"+attributeName + "[" +
            attributeOwner + "ID] = " + agentName
            +"_"+valueAttr + "[" + valueOwner
            + "ID];\n");
        toPromela = toPromela.concat("
            printf(\"FACT UPDATE7: fact_"+
            attributeName + "[" + attributeOwner
            + "ID] = %e (String) or %d (Integer)\\n\",
            fact_"+attributeName + "[" + attributeOwner
            + "ID], fact_"+attributeName + "[" +
            attributeOwner + "ID]);\n");
        toPromela = toPromela.concat("
            }\n");
    }
    else{
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            "+"searchID = 0;\n");
        toPromela = toPromela.concat("
            "+"findID("+attributeOwner+");\n");
        toPromela = toPromela.concat("
            "+"fact_"+attributeName + "[searchID]
            = " + agentName+"_"+valueAttr + "["
            + valueOwner + "ID];\n");
        toPromela = toPromela.concat("
            printf(\"FACT UPDATE8: fact_"+attributeName
            + "[searchID] = %e (String) or %d (Integer)
            \\n\", fact_"+attributeName + "[searchID],
            fact_"+attributeName + "[searchID]);\n");
        toPromela = toPromela.concat("
        }\n");
    }
else if(fc == 0  || frame.equals("_tf_"))
    toPromela = toPromela.concat("
        /*Fact not updated*/\n");
else{
    toPromela = toPromela.concat("
        if\n");
    toPromela = toPromela.concat("
        ::(true)->\n");
    if(attributeOwnerVariable == false){
        toPromela = toPromela.concat("
                            d_step{\n");
                        toPromela = toPromela.concat("
                            "+"fact_"+attributeName + "[" +
                            attributeOwner + "ID] = " + agentName+"_"+
                            valueAttr + "[" + valueOwner + "ID];\n");
                        toPromela = toPromela.concat("
                            printf(\"FACT UPDATE9: fact_"+attributeName
                            + "[" + attributeOwner + "ID] = %e (String)
                            or %d (Integar)\\n\", fact_"+attributeName
                            + "[" + attributeOwner + "ID], fact_"+
                            attributeName + "[" + attributeOwner + "
                            ID]);\n");
                        toPromela = toPromela.concat("
                            }\n");
                    }
                    else{
                        toPromela = toPromela.concat("
                            d_step{\n");
                        toPromela = toPromela.concat("
                            "+"searchID = 0;\n");
                        toPromela = toPromela.concat("
                            "+"findID("+attributeOwner+");\n");
                        toPromela = toPromela.concat("
                            "+"fact_"+attributeName + "[searchID] = " +
                            agentName+"_"+valueAttr + "[" + valueOwner
                            + "ID];\n");
                        toPromela = toPromela.concat("
                            printf(\"FACT UPDATE10: fact_"+
                            attributeName + "[searchID] = %e (String)
                            or %d (Integar)\\n\", fact_"+attributeName
                            + "[searchID], fact_"+attributeName +
                            "[searchID]);\n");
                        toPromela = toPromela.concat("
                            }\n");
                    }
                    toPromela = toPromela.concat("
                        ::(true)->\n");
                    toPromela = toPromela.concat("
                        skip;\n");
                    toPromela = toPromela.concat("
                        fi;");
                }
// If belief condition is 100% then just put a straight conclude
                if(bc ==100)
                    if(attributeOwnerVariable == false){
                        toPromela = toPromela.concat("
                            d_step{\n");
                        toPromela = toPromela.concat("
                            "+agentName+"_"+attributeName + "[" +
                 attributeOwner + "ID] = " + agentName
                 +"_"+valueAttr + "[" + valueOwner +
                 "ID];\n");
            toPromela = toPromela.concat("
                printf(\"BELIEF UPDATE: "+agentName+"_"+
                attributeName + "[" + attributeOwner +
                "ID] = %e (String) or %d (Integegar)\\n\",
                "+agentName+"_"+attributeName + "[" +
                attributeOwner + "ID], "+ agentName+"_"+
                attributeName + "[" + attributeOwner + "
                ID]);\n");
            toPromela = toPromela.concat("
            }\n");
        }
        else{
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"searchID = 0;\n");
            toPromela = toPromela.concat("
                "+"findID("+attributeOwner+");\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName + "[searchID]
                = " + agentName+"_"+valueAttr + "[" +
                valueOwner + "ID];\n");
            toPromela = toPromela.concat("
                printf(\"BELIEF UPDATE: "+agentName+"_"+
                attributeName + "[searchID] = %e (String)
                or %d (Integar)\\n\", "+agentName+"_"+
                attributeName + "[searchID]," + agentName
                +"_"+attributeName + "[searchID]);\n");
            toPromela = toPromela.concat("
                }\n");
        }
    else if(bc == 0)
        toPromela = toPromela.concat("
            /*Belief not updated*/\n");
    else{ //If not create a split
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(true)->\n");
        if(attributeOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName + "[" +
                 attributeOwner + "ID] = " + agentName+
                 "_"+valueAttr + "[" + valueOwner + "ID];
```

```
                    \n");
                toPromela = toPromela.concat(" 0
                    printf(\"BELIEF UPDATE: "+agentName+"_"+
                    attributeName + "[" + attributeOwner + "
                    ID] = %e (String) or %d (Integar)\\n\", "
                    +agentName+"_"+attributeName + "[" +
                    attributeOwner + "ID], "+ agentName+"_"+
                    attributeName + "[" + attributeOwner +
                    "ID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            else{
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+"searchID = 0;\n");
                toPromela = toPromela.concat("
                    "+"findID("+attributeOwner+");\n");
                toPromela = toPromela.concat("
                    "+agentName+"_"+attributeName + "
                    [searchID] = " + agentName+"_"+
                    valueAttr + "[" + valueOwner + "ID];\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            toPromela = toPromela.concat("
                ::(true)->\n");
            toPromela = toPromela.concat("
                skip;\n");
            toPromela = toPromela.concat("
                fi;");
        }
    }

    if(valueOwner != null && valueOwner2 == null && value
            != null){
        //same for facts
        if(fc ==100 && frame.equals("_wf_")){
            if(attributeOwnerVariable == false &&
                    valueOwnerVariable == false){
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+"fact_"+attributeName + "[" +
                    attributeOwner + "ID] = " + agentName+
                    "_"+valueAttr + "[" + valueOwner + "ID] "+
                    valueOperator + " " + value + ";\n");
                toPromela = toPromela.concat("
            printf(\"FACT UPDATE11: fact_"+
            attributeName + "[" + attributeOwner +
            "ID] = %e (String) or %d (Integar)\\n\"
            , fact_"+attributeName + "[" +
            attributeOwner + "ID], fact_"+
            attributeName + "[" + attributeOwner + "
            ID]);\n");
        toPromela = toPromela.concat("
            }\n");
    }
    else if(attributeOwnerVariable == true &&
            valueOwnerVariable == false){
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            "+"searchID = 0;\n");
        toPromela = toPromela.concat("
            "+"findID("+attributeOwner+");\n");
        toPromela = toPromela.concat("
            "+"fact_"+attributeName + "[searchID] = " +
                agentName+"_"+valueAttr + "[" +
                valueOwner + "ID] " + valueOperator
                + " " + value + ";\n");
        toPromela = toPromela.concat("
            "+"printf(\"FACT UPDATE: fact_"+
            attributeName + "[searchID] = %e (String)
            or %d (Integar)\\n\",fact_"+attributeName
            + "[searchID], fact_"+ attributeName +
            "[searchID]);\n");
        toPromela = toPromela.concat("
        }\n");
    }
    else if(attributeOwnerVariable == true &&
            valueOwnerVariable == true){
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            "+"searchID = 0;\n");
        toPromela = toPromela.concat("
            "+"findID("+attributeOwner+");\n");
        toPromela = toPromela.concat("
            "+"multiVarOne = searchID;\n");
        toPromela = toPromela.concat("
            "+"findID("+valueOwner+");\n");
        toPromela = toPromela.concat("
            "+"multiVarTwo = searchID;\n");
        toPromela = toPromela.concat("
            "+"fact_"+attributeName + "[multiVarOne]="
            + agentName+"_"+valueAttr+"[multiVarTwo] "
```

```
                + valueOperator + " " + value + ";\n");
            toPromela = toPromela.concat("
                "+"printf(\"FACT UPDATE: fact_"+
                attributeName + "[multiVarOne] = %e
                (String) or %d (Integar) \\n\",
                fact_"+attributeName + "[multiVarOne],
                fact_"+attributeName + "[multiVarOne]);
                \n");
            toPromela = toPromela.concat("
                "+"multiVarOne = 0;\n");
            toPromela = toPromela.concat("
                "+"multiVarTwo = 0;\n");
            toPromela = toPromela.concat("
                }\n");
        }
    }
    else if(fc == 0 || frame.equals("_tf_")){
        toPromela = toPromela.concat("
            /*Fact not updated*/\n");
    }
    else if(frame.equals("_wf_")){
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(true)->\n");
        if(attributeOwnerVariable == false &&
                valueOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                    "+"fact_"+attributeName + "[" +
                    attributeOwner + "ID] = " + agentName
                    +"_"+valueAttr + "[" + valueOwner +
                    "ID] " + valueOperator + " " +
                    value + ";\n");
            toPromela = toPromela.concat("
                printf(\"FACT UPDATE12: fact_"+
                attributeName + "[" + attributeOwner
                + "ID] = %e (String) or %d (Integar)\\n\",
                fact_"+attributeName + "[" + attributeOwner
                + "ID], fact_"+attributeName + "[" +
                attributeOwner + "ID]);\n");
            toPromela = toPromela.concat("
                }\n");
        }
        else if(attributeOwnerVariable == true &&
                valueOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
    toPromela = toPromela.concat("
        "+"searchID = 0;\n");
    toPromela = toPromela.concat("
        "+"findID("+attributeOwner+");\n");
    toPromela = toPromela.concat("
        "+"fact_"+attributeName + "[searchID] = "
        + agentName+"_"+valueAttr + "[" +
        valueOwner + "ID] " + valueOperator +
        " " + value + ";\n");
    toPromela = toPromela.concat("
        "+"printf(\"FACT UPDATE13: fact_"+
        attributeName + "[searchID] = %e
        (String) or %d (Integer)\\n\",
        fact_"+attributeName + "[searchID],
        fact_"+attributeName + "[searchID]);\n");
    toPromela = toPromela.concat("
        }\n");
}
else if(attributeOwnerVariable == true &&
        valueOwnerVariable == true){
    toPromela = toPromela.concat("
        d_step{\n");
    toPromela = toPromela.concat("
        "+"searchID = 0;\n");
    toPromela = toPromela.concat("
        "+"findID("+attributeOwner+");\n");
    toPromela = toPromela.concat("
        "+"multiVarOne = searchID;\n");
    toPromela = toPromela.concat("
        "+"findID("+valueOwner+");\n");
    toPromela = toPromela.concat("
        "+"multiVarTwo = searchID;\n");
    toPromela = toPromela.concat("
        "+"fact_"+attributeName + "[multiVarOne]
        = " + agentName+"_"+valueAttr + "
        [multiVarTwo] " + valueOperator + " " +
        value + ";\n");
    toPromela = toPromela.concat("
        "+"printf(\"FACT UPDATE14: fact_"+
        attributeName + "[multiVarOne] = %e
        (String) or %d (Integer)\\n\",
        fact_"+attributeName + "[multiVarOne],
        fact_"+attributeName +
        "[multiVarOne]);\n");
    toPromela = toPromela.concat("
        "+"multiVarOne = 0;\n");
    toPromela = toPromela.concat("
        "+"multiVarTwo = 0;\n");
    toPromela = toPromela.concat("
                }\n");
            toPromela = toPromela.concat("
                ::(true)->\n");
            toPromela = toPromela.concat("
                skip;\n");
            toPromela = toPromela.concat("
                fi;");
            }
        }
        // If belief condition is 100%
        if(bc ==100){
            if(attributeOwnerVariable == false){
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+agentName+"_"+attributeName + "[" +
                    attributeOwner + "ID] = " + agentName+
                    "_"+valueAttr + "[" + valueOwner + "ID] "
                    + valueOperator + " " + value + ";\n");
                toPromela = toPromela.concat("
                    printf(\"BELIEF UPDATE: "+agentName+"_"+
                    attributeName + "[" + attributeOwner +
                    "ID] = %e (String) or %d (Integar)\\n\",
                    "+agentName+"_"+attributeName + "[" +
                    attributeOwner + "ID], "+ agentName+"_"+
                    attributeName + "[" + attributeOwner +
                    "ID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            else if(attributeOwnerVariable == true &&
                    valueOwnerVariable == false){
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+"searchID = 0;\n");
                toPromela = toPromela.concat("
                    "+"findID("+attributeOwner+");\n");
                toPromela = toPromela.concat("
                    "+agentName+"_"+attributeName +
                    "[searchID] = " + agentName+"_"+
                    valueAttr + "[" + valueOwner + "ID] " +
                    valueOperator + " " + value + ";\n");
                toPromela = toPromela.concat("
                    printf(\"BELIEF UPDATE: "+agentName+"_"+
                    attributeName + "[searchID] = %e (String)
                    or %d (Integar)\\n\", "+agentName+"_"+
                    attributeName + "[searchID]," + agentName+
                    "_"+attributeName + "[searchID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            else if(attributeOwnerVariable == true &&
                    valueOwnerVariable == true){
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+"searchID = 0;\n");
                toPromela = toPromela.concat("
                    "+"findID("+attributeOwner+");\n");
                toPromela = toPromela.concat("
                    "+"multiVarOne = searchID;\n");
                toPromela = toPromela.concat("
                    "+"findID("+valueOwner+");\n");
                toPromela = toPromela.concat("
                    "+"multiVarTwo = searchID;\n");
                toPromela = toPromela.concat("
                    "+agentName+"_"+attributeName +
                    "[multiVarOne] = " + agentName+"_"+
                    valueAttr + "[multiVarTwo] " +
                    valueOperator + " " + value + ";\n");
                    toPromela = toPromela.concat("
                    "+"printf(\"BELIEF UPDATE: "+agentName
                    +"_"+ attributeName + "[multiVarOne] =
                    %e (String) or %d (Integar)\\n\", "+
                    agentName+"_"+ attributeName + "
                    [multiVarOne],"+agentName+ "_" +
                    attributeName
                    + "[multiVarOne]);\n");
                toPromela = toPromela.concat("
                    "+"multiVarOne = 0;\n");
                toPromela = toPromela.concat("
                    "+"multiVarTwo = 0;\n");
                toPromela = toPromela.concat("
                    }\n");
            }
        }
        else if(bc == 0){
            toPromela = toPromela.concat("
                /*Belief not updated*/\n");
        }
        else{ //If not create a split
            toPromela = toPromela.concat("
                if\n");
            toPromela = toPromela.concat("
                ::(true)->\n");
            if(attributeOwnerVariable == false){
                toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName + "[" +
                attributeOwner + "ID] = " + agentName+"_"+
                valueAttr + "[" + valueOwner + "ID] " +
                valueOperator + " " + value + ";\n");
            toPromela = toPromela.concat("
                printf(\"BELIEF UPDATE: "+agentName+"_"+
                attributeName + "[" + attributeOwner + "ID]
                = %e (String) or %d (Integer)\\n\", "+
                agentName+"_"+attributeName + "[" +
                attributeOwner + "ID], "+ agentName+"_"+
                attributeName + "[" + attributeOwner +
                "ID]);\n");
            toPromela = toPromela.concat("
                }\n");
        }
        else if(attributeOwnerVariable == true &&
                valueOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"searchID = 0;\n");
            toPromela = toPromela.concat("
                "+"findID("+attributeOwner+");\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName + "[searchID]
                = " + agentName+"_"+valueAttr + "[" +
                valueOwner + "ID] " + valueOperator + " " +
                value + ";\n");
            toPromela = toPromela.concat("
            printf(\"BELIEF UPDATE: "+agentName+"_"+
            attributeName + "[searchID] = %e (String) or
            %d (Integer)\\n\", "+agentName+"_"+
            attributeName + "[searchID]," + agentName
            +"_"+attributeName + "[searchID]);\n");
            toPromela = toPromela.concat("
                }\n");
        }
        else if(attributeOwnerVariable == true &&
                valueOwnerVariable == true){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"searchID = 0;\n");
            toPromela = toPromela.concat("
                "+"findID("+attributeOwner+");\n");
            toPromela = toPromela.concat("
                "+"multiVarOne = searchID;\n");
            toPromela = toPromela.concat("
                "+"findID("+valueOwner+");\n");
            toPromela = toPromela.concat("
                "+"multiVarTwo = searchID;\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName +
                "[multiVarOne] = " + agentName+"_"+
                valueAttr + "[multiVarTwo] " +
                valueOperator + " " + value + ";\n");
            toPromela = toPromela.concat("
                "+"printf(\"BELIEF UPDATE: "+
                agentName+"_"+
                attributeName + "[multiVarOne] = %e
                (String) or %d (Integar)\\n\", "+
                agentName+"_"+ attributeName +
                "[multiVarOne], "+agentName+ "_" +
                attributeName + "[multiVarOne]);\n");
            toPromela = toPromela.concat("
                "+"multiVarOne = 0;\n");
            toPromela = toPromela.concat("
                "+"multiVarTwo = 0;\n");
            toPromela = toPromela.concat("
                }\n");
        }
        toPromela = toPromela.concat("
            ::(true)->\n");
        toPromela = toPromela.concat("
            skip;\n");
        toPromela = toPromela.concat("
            fi;");
    }
}


if(valueOwner != null && valueOwner2 != null){
    if(fc ==100 && frame.equals("_wf_")) //same for facts
        if(attributeOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"fact_"+attributeName + "[" +
                 attributeOwner + "ID] = " + agentName+"_"+
                 valueAttr + "[" + valueOwner + "ID] " +
                 valueOperator + " " + agentName + "_" +
                 valueAttr2 + "[" + valueOwner2 + "ID] "
                 + ";\n");
            toPromela = toPromela.concat("
                printf(\"FACT UPDATE15: fact_"+
                attributeName + "[" +
                attributeOwner + "ID] = %e (String)
                or %d (Integar)\\n\", fact_"+
                attributeName + "[" + attributeOwner +
                "ID], fact_"+ attributeName + "[" +
                attributeOwner + "ID]);\n");
            toPromela = toPromela.concat("
                }\n");
        }
        else{
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"searchID = 0;\n");
            toPromela = toPromela.concat("
                "+"findID("+attributeOwner+");\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName + "
                [searchID] = " + agentName+"_"+
                valueAttr + "[" + valueOwner + "ID] " +
                valueOperator + " " + agentName + "_" +
                valueAttr2 + "[" + valueOwner2 + "ID] "
                + ";\n");
            toPromela = toPromela.concat("
                }\n");
        }
    else if(fc == 0 || frame.equals("_tf_"))
        toPromela = toPromela.concat("
            /*Fact not updated*/\n");
    else if(frame.equals("_wf_")){
        toPromela = toPromela.concat("
            if\n");
        toPromela = toPromela.concat("
            ::(true)->\n");
        if(attributeOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"fact_"+attributeName + "[" +
                 attributeOwner + "ID] = " + agentName+
                 "_"+valueAttr + "[" + valueOwner + "ID]
                 " + valueOperator + " " + agentName + "_"
                 + valueAttr2 + "[" + valueOwner2 + "ID] "
                 + ";\n");
            toPromela = toPromela.concat("
            printf(\"FACT UPDATE16: fact_"+attributeName
            + "[" + attributeOwner + "ID] = %e (String)
            or %d (Integar)\\n\", fact_"+attributeName
            + "[" + attributeOwner + "ID], fact_"+
            attributeName + "[" + attributeOwner +
            "ID]);\n");
        toPromela = toPromela.concat("
                }\n");
        }
        else{
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"searchID = 0;\n");
            toPromela = toPromela.concat("
                "+"findID("+attributeOwner+");\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName +
                "[searchID] = " + agentName+"_"+
                valueAttr + "[" + valueOwner + "ID] "
                + valueOperator + " " + agentName + "_"
                + valueAttr2 + "[" + valueOwner2 + "ID]
                " + ";\n");
            toPromela = toPromela.concat("
                }\n");
        }
        toPromela = toPromela.concat("
            ::(true)->\n");
        toPromela = toPromela.concat("
            skip;\n");
        toPromela = toPromela.concat("
            fi;");
    }
    if(bc ==100)
        if(attributeOwnerVariable == false){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+agentName+"_"+attributeName + "[" +
                attributeOwner + "ID] = " + agentName
                +"_"+valueAttr + "[" + valueOwner + "ID] "
                + valueOperator + " " + agentName + "_" +
                valueAttr2 + "[" + valueOwner2 + "ID] " +
                ";\n");
            toPromela = toPromela.concat("
                printf(\"BELIEF UPDATE: "+agentName+"_"+
                attributeName + "[" + attributeOwner + "ID]
                = %e (String) or %d (Integar)\\n\",
                "+agentName+"_"+attributeName + "[" +
                attributeOwner + "ID], "+ agentName+"_"+
                attributeName + "[" + attributeOwner +
                "ID]);\n");
            toPromela = toPromela.concat("
                }\n");
    }
    else{
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            "+"searchID = 0;\n");
        toPromela = toPromela.concat("
            "+"findID("+attributeOwner+");\n");
        toPromela = toPromela.concat("
            "+agentName+"_"+attributeName + "[searchID]
            = " + value + ";\n");
        toPromela = toPromela.concat("
            printf(\"Search = %e\\n\",
            agentsObjectsIDs[searchID]);\n");
        toPromela = toPromela.concat("
            printf(\"BELIEF UPDATE: "+agentName+"_"+
            attributeName + "[searchID] = %e (String)
            or %d (Integar)\\n\", "+agentName+"_"+
            attributeName + "[searchID]," + agentName
            +"_"+attributeName + "[searchID]);\n");
        toPromela = toPromela.concat("
        }\n");
    }
else if(bc == 0)
    toPromela = toPromela.concat("
        /*Belief not updated*/\n");
//If not create a split where conclude is not done
else{
    toPromela = toPromela.concat("
        if\n");
    toPromela = toPromela.concat("
        ::(true)->\n");
    if(attributeOwnerVariable == false){
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            "+agentName+"_"+attributeName + "[" +
            attributeOwner + "ID] = " + agentName+"_"+
            valueAttr + "[" + valueOwner + "ID] " +
            valueOperator + " " + agentName + "_" +
            valueAttr2 + "[" + valueOwner2 + "ID] " +
            ";\n");
        toPromela = toPromela.concat("
            printf(\"BELIEF UPDATE: "+agentName+"_"+
            attributeName + "[" + attributeOwner +
            "ID] = %e (String) or %d (Integar)\\n\"
            , "+agentName+"_"+attributeName + "["
            + attributeOwner + "ID], "+ agentName
            +"_"+attributeName + "[" + attributeOwner
```

```
                    + "ID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            else{
                toPromela = toPromela.concat("
                    d_step{\n");
                toPromela = toPromela.concat("
                    "+"searchID = 0;\n");
                toPromela = toPromela.concat("
                    "+"findID("+attributeOwner+");\n");
                toPromela = toPromela.concat("
                    "+agentName+"_"+attributeName +
                    "[searchID] = " + agentName+"_"+valueAttr +
                    "[" + valueOwner + "ID] " + valueOperator +
                    " " + agentName + "_" + valueAttr2 +
                    "[" + valueOwner2 + "ID] " + ";\n");
                toPromela = toPromela.concat("
                    printf(\"BELIEF UPDATE: "+agentName+"_"+
                    attributeName + "[searchID] = %e (String)
                    or %d (Integar)\\n\", "+agentName+"_"+
                    attributeName + "[searchID]," +
                    agentName+"_"+attributeName +
                    "[searchID]);\n");
                toPromela = toPromela.concat("
                    }\n");
            }
            toPromela = toPromela.concat("
                ::(true)->\n");
            toPromela = toPromela.concat("
                skip;\n");
            toPromela = toPromela.concat("
                fi;");
        }
    }

    PromelaBeliefUpdate.add(attributeOwner+"_"+attributeName
        + "[" + attributeOwner + "ID]");
}
if(type == eventType.CommAct){
    if(whom_where.equals("current")){
        whom_where = agentName;
        whomWhereCurrent = true;
    }

    for (Iterator<messages> messIt = mess.iterator();
            messIt.hasNext(); ) {
        messages m = messIt.next();
        messAbout = m.getMessAbout();
```

```
messAbout2 = m.getMessAbout2();
messAtt = m.getMessAtt();
messAtt2 = m.getMessAtt2();
boolean messHasVar = false;

if(messAbout.equals("current")){
    messAbout = agentName;
    messAboutCurrent = true;
}
if(messAbout2.equals("current")){
    messAbout2 = agentName;
    messAbout2Current = true;
}
for(Iterator<variable> varit = variables.iterator();
        varit.hasNext();){
    variable v = varit.next();
    if(messAbout.equals(v.getName())){
        messAbout = "";
        messAbout = messAbout.concat(""+agentName+
            frame+f_name+"_var["+agentName+frame+
            f_name+"_index].var_elements["+
            v.getVarNo()+"]");
        messHasVar = true;
    }
    if(messAbout2.equals(v.getName())){
        messAbout2 = "";
        messAbout2 = messAbout2.concat(""+agentName+
            frame+f_name+"_var["+agentName+frame+
            f_name+"_index].var_elements["+
            v.getVarNo()+"]");
        messHasVar = true;
    }
}
if(hasVariable == false && messHasVar == false){
    toPromela = toPromela.concat("
        "+"/*No variable*/\n");
    toPromela = toPromela.concat("
        "+whom_where+"_"+messAtt + "[" + messAbout +
        "ID] = " + agentName+"_"+ messAtt2 + "[" +
        messAbout + "ID];\n");
    toPromela = toPromela.concat("
        printf(\"COMM BELIEF UPDATE: "+whom_where+
        "_"+messAtt + "[" + messAbout + "ID] = %e
        (String) or %d (Integar)\\n\", "+whom_where
        +"_"+messAtt + "[" + messAbout + "ID], "+
        whom_where+"_"+messAtt + "[" + messAbout +
        "ID]);\n");
    PromelaBeliefUpdate.add("
        "+whom_where+"_"+messAtt + "[" + agentName +
        "ID]");
}
else if(hasVariable == false && messHasVar == true){
    toPromela = toPromela.concat("
        "+"/*Agent/Object in message is a variable
        */\n");
    toPromela = toPromela.concat("
        d_step{\n");
    toPromela = toPromela.concat("
        "+"          searchID = 0;\n");
    toPromela = toPromela.concat("
        "+"          findID("+messAbout+");\n");
    toPromela = toPromela.concat("
        printf(\"Search = %e\\n\", agentsObjectsIDs
        [searchID]);\n");
    toPromela = toPromela.concat("
        "+whom_where+"_"+messAtt + "[searchID] = " +
        agentName+"_"+ messAtt2 + "[searchID];\n");
    toPromela = toPromela.concat("
        printf(\"COMM BELIEF UPDATE: "+whom_where
        +"_"+messAtt + "[searchID] = %e (String)
        or %d (Integar)\\n\", "+whom_where+"_"+
        messAtt + "[searchID], "+ whom_where+"_"+
        messAtt + "[searchID]);\n");
    PromelaBeliefUpdate.add("
        "+whom_where+"_"+messAtt + "[" +
        agentName + "ID]");
    toPromela = toPromela.concat("
        }\n");
}
else if(hasVariable == true && messHasVar == false){
    toPromela = toPromela.concat("
        "+"/*Receiver is a variable*/\n");
    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        if(ag.getMemberOf().contains(paramType) ||
                ag.getMemberOf().contains
                (paramType2)){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"if\n");
            toPromela = toPromela.concat("
                "+"::("+whom_where+" == "+
                ag.getName()+")->\n");
            toPromela = toPromela.concat("
                "+"    "+ag.getName()+"_"+messAtt
                + "["+messAbout+"ID] = " + agentName
                +"_"+ messAtt2 + "[" + messAbout +
                "ID];\n");
            toPromela = toPromela.concat("
                printf(\"COMM BELIEF UPDATE: "+
                ag.getName()+"_"+messAtt + "[" +
                messAbout + "ID] = %e (String) or %d
                (Integar)\\n\", "+ag.getName()+"_"+
                messAtt + "[" + messAbout + "ID], "+
                ag.getName() +"_"+messAtt + "[" +
                messAbout + "ID]);\n");
            toPromela = toPromela.concat("
                "+"::else ->\n");
            toPromela = toPromela.concat("
                "+"    skip;\n");
            toPromela = toPromela.concat("
                "+"fi;\n");
            toPromela = toPromela.concat("
                }\n");
        }
    }
    for (Iterator<object> objectit =
            objects.iterator();
            objectit.hasNext(); ){
        object ob = objectit.next();
        if(ob.getMemberOf().contains(paramType)
                || ob.getMemberOf().contains
                (paramType2)){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"if\n");
            toPromela = toPromela.concat("
                "+"::("+whom_where+" == "+
                ob.getName()+")->\n");
            toPromela = toPromela.concat("
                "+"    "+ob.getName()+"_"+messAtt +
                "["+messAbout+"ID] = " +
                agentName+"_"+ messAtt2 + "[" +
                messAbout + "ID];\n");
            toPromela = toPromela.concat("
                printf(\"COMM BELIEF UPDATE:
                "+ob.getName()+"_"+messAtt + "["
                + messAbout + "ID] = %e (String)
                or %d (Integar)\\n\", "+ob.getName
                ()+"_"+messAtt + "[" + messAbout +
                "ID], "+ ob.getName()+"_"+messAtt +
                "[" + messAbout + "ID]);\n");
            toPromela = toPromela.concat("
                "+"::else ->\n");
            toPromela = toPromela.concat("
                "+"    skip;\n");
            toPromela = toPromela.concat("
                "+"fi;\n");
            toPromela = toPromela.concat("
                }\n");
        }
    }
}
else if(hasVariable == true && messHasVar == true){
    toPromela = toPromela.concat("
        d_step{\n");
    toPromela = toPromela.concat("
        "+"/*Both are variables*/\n");
    toPromela = toPromela.concat("
        "+"          searchID = 0;\n");
    toPromela = toPromela.concat("
        "+"          findID("+messAbout+");\n");
    toPromela = toPromela.concat("
        printf(\"Search = %e\\n\", agentsObjectsIDs
        [searchID]);\n");
    toPromela = toPromela.concat("
        }\n");

    for (Iterator<agent> agentit = agents.iterator();
            agentit.hasNext(); ){
        agent ag = agentit.next();
        if(ag.getMemberOf().contains(paramType)
                || ag.getMemberOf().contains
                (paramType2)){
            toPromela = toPromela.concat("
                d_step{\n");
            toPromela = toPromela.concat("
                "+"if\n");
            toPromela = toPromela.concat("
                "+"::("+whom_where+" == "+
                ag.getName()+")->\n");
            toPromela = toPromela.concat("
                "+"    "+ag.getName()+"_"+messAtt +
                "[searchID] = " + agentName+"_"+
                messAtt2 + "[searchID];\n");
            toPromela = toPromela.concat("
                printf(\"COMM BELIEF UPDATE: "+ag.
                getName()+"_"+messAtt + "[searchID]
                = %e (String) or %d
                (Integar)\\n\", "+ag.getName()+"_"+
                messAtt + "[searchID], "+ ag.getName()+
                "_"+messAtt + "[searchID]);\n");
            toPromela = toPromela.concat("
                        "+"::else ->\n");
                    toPromela = toPromela.concat("
                        "+"    skip;\n");
                    toPromela = toPromela.concat("
                        "+"fi;\n");
                    toPromela = toPromela.concat("
                        }\n");
                }
            }
            for (Iterator<object> objectit = objects.
                   iterator(); objectit.hasNext(); ){
                object ob = objectit.next();
                if(ob.getMemberOf().contains(paramType) ||
                        ob.getMemberOf().contains(paramType2)){
                    toPromela = toPromela.concat("
                        d_step{\n");
                    toPromela = toPromela.concat("
                        "+"if\n");
                    toPromela = toPromela.concat("
                        "+"::("+whom_where+" == "+
                        ob.getName()+")->\n");
                    toPromela = toPromela.concat("
                        "+"    "+ob.getName()+"_"+messAtt +
                        "[searchID] = " + agentName+"_"+
                        messAtt2 + "[searchID];\n");
                    toPromela = toPromela.concat("
                        printf(\"COMM BELIEF UPDATE: "+ob.
                        getName()+"_"+messAtt + "[searchID]
                        = %e (String) or %d (Integar)\\n\", "
                        +ob.getName()+"_"+messAtt + "[searchID]
                        , "+ ob.getName()+"_"+messAtt + "
                        [searchID]);\n");
                    toPromela = toPromela.concat("
                        "+"::else ->\n");
                    toPromela = toPromela.concat("
                        "+"    skip;\n");
                    toPromela = toPromela.concat("
                        "+"fi;\n");
                    toPromela = toPromela.concat("
                        }\n");
                }
            }
        }
        messAbout = null;
        messAbout2 = null;
        messAtt = null;
        messAtt2 = null;
    }
}
```

```
    if(type == eventType.Move){
        toPromela = toPromela.concat("
            d_step{\n");
        toPromela = toPromela.concat("
            "+agentName+"_location[" + agentName + "ID] = " +
            whom_where + ";\n");
        toPromelaMove = toPromelaMove.concat("fact_location["+
            agentName+"ID] = " + whom_where + ";\n");
        toPromela = toPromela.concat("
            printf(\"MOVE BELIEF UPDATE: "+agentName+"_location[" +
            agentName + "ID] = %e (String) or %d (Integar)\\n\", "+
            agentName+"_location[" + agentName + "ID], "+
            agentName+"_location[" + agentName + "ID]);\n");
        PromelaBeliefUpdate.add(agentName+"_location[" +
            agentName + "ID]");
        PromelaBeliefUpdate.add("fact_location[" +
            agentName + "ID]");
        toPromela = toPromela.concat("
            }\n");

    }
    if(attributeOwnerCurrent)
        attributeOwner = "current";
    if(valueOwnerCurrent)
        valueOwner = "current";
    if(valueOwner2Current)
        valueOwner2 = "current";
    if(whomWhereCurrent)
        whom_where = "current";
    if(messAboutCurrent)
        messAbout = "current";
    if(messAbout2Current)
        messAbout2 = "current";

    return toPromela;

}

public void inCollectAll(){
    isCollectAll = true;
}

public String getWhomWhere(){
    return whom_where;
}

public Set getPromelaFactUpdate(){
    return PromelaFactUpdate;
```

```
}

public Set getBeliefUpdate(){
    return PromelaBeliefUpdate;
}

public String getToPromelaMove(){
    return toPromelaMove;
}

public eventType getType()
 {
    return type;
 }

public String getName()
 {
    return name;
 }

public int getID()
 {
    return concID;
 }
public int getDuration()
 {
    return duration;
 }
 public String getAttOwner()
 {
    return attributeOwner;
 }
 public String getAttName()
 {

    return attributeName;
 }
 public String getValueOwner()
 {
    return valueOwner;
 }
 public String getValue()
 {
    return value;
 }
public int getBc()
 {
    return bc;
 }
```

```
    public int getFc()
     {
        return fc;
     }
    public String getFName(){
        return f_name;
    }
    public boolean getCollectAll(){
        return isCollectAll;
    }
    public boolean getHasVar(){
        return hasVar;
    }

}
```

## C.12 Concludes

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

import java.util.Stack;
class conclude
{
    String association;  //  Who the conclude belief belongs to
    String belief; // name of the attribute
    String value; // new value
}
```

## C.13 Communication Messages

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Stores all messgae for communication.
*/

import java.util.*;
```

```
class messages {

    String messAbout;
    String messAbout2;

    String messAtt;
    String messAtt2;

    public messages(String new_messAbout, String new_messAbout2,
            String new_messAtt, String new_messAtt2) {

        messAbout = new_messAbout;
        messAbout2= new_messAbout2;
        messAtt = new_messAtt;
        messAtt2 = new_messAtt2;
    }

    public String getMessAbout(){
        return messAbout;
    }
    public String getMessAbout2(){
        return messAbout2;
    }
    public String getMessAtt(){
        return messAtt;
    }
    public String getMessAtt2(){
        return messAtt2;
    }
}
```

## C.14 Workframes

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Workframe details.
*/

import java.util.*;
class workframe
{   // name of agent it belongs to
    public String agent;
    // name of the workframe
    public String wf_name;
```

```
// repeat: true, once, false.
public String wf_repeat = "false";
// priority of the workframe
public int wf_priority = 0;
// The variables it contains
public Set<variable> vars = new HashSet();
// The detectables it contains
public Set<detectable> detectables = new HashSet();
// Guard condition of the thoughtframe
public Set<guard> guards = new HashSet();
// Events contained within the thoughtframe
public List<event> events = new ArrayList();

/****************
*For Promela use*
*****************/
public int ID; // workframes ID number
public boolean containsCollectAll = false;
// Details of all other agents/objects
public Set<agent> mas = new HashSet();
// Guards which don't contain variables
Set<String> staticGuards = new HashSet<String>();
// Guards which contain variables
Set<String> varGuards = new HashSet<String>();
// Names of the variables
public Set<String> VarNames = new HashSet();
String promelaString = "";

public workframe(){}

public workframe(Set<agent> new_mas, int new_ID,
String new_agent, String new_name, String new_repeat,
int new_priority, Set<variable> new_variables,
Set<detectable> new_detectables, Set<guard> new_guards,
List<event> new_events) {

    mas = new_mas;
    ID = new_ID;
    agent = new_agent;
    wf_name = new_name;
    wf_repeat = new_repeat;
    wf_priority = new_priority;
    vars = new_variables;
    detectables = new_detectables;
    guards = new_guards;
    events = new_events;
}

public String toPromelaString(String[] identificationNumbers,
String agentName, String agentObject){
    promelaString = "";
    varGuards.clear();
    staticGuards.clear();
    boolean hasVariable;
    //  Cycle through guards to find which contain variables
    for (Iterator<guard> Garit = guards.iterator();
    Garit.hasNext(); ) {
        guard gar = Garit.next();
        String guardToString = gar.toPromela(
            identificationNumbers, vars, agentName,
            agentObject);
        if(gar.getHasVariable())
            varGuards.add(guardToString);
        else
            staticGuards.add(guardToString);
    }
    if(vars.size() == 0){
        promelaString = promelaString.concat("
        /* Workframe "+ wf_name +", has no variables
        and agent name is "+agentName+"*/\n");

        promelaString = promelaString.concat("
            if\n");
        promelaString = promelaString.concat("
            ::(");
        for (Iterator<String> statGarit = staticGuards.iterator();
        statGarit.hasNext(); ) {
            String g = statGarit.next();
            if(statGarit.hasNext())
                promelaString = promelaString.concat(g + " && ");
            else
                promelaString = promelaString.concat(g +
                    ")-> \n");
        }
        // If statement for relations
        promelaString = promelaString.concat("
            i = 0;\n");
        promelaString = promelaString.concat("
            printf(\"Workframe "+wf_name+" is active\\n\");\n");
        promelaString = promelaString.concat("
            " + agentName + "wfActive(i, "+ID+")\n");
        promelaString = promelaString.concat("
            ::else->\n");
        promelaString = promelaString.concat("
            i = 0;\n");
        promelaString = promelaString.concat("                 "
            + agentName + "wfNotActive(i, " + ID + ")\n");
        promelaString = promelaString.concat("
            fi;\n");
    }
    else{ // Has variables
        promelaString = promelaString.concat("
            /* Workframe "+ wf_name +", has variables*/\n");
        if(!staticGuards.isEmpty()){
            promelaString = promelaString.concat("
                if\n");
            promelaString = promelaString.concat("
                ::(");
            for (Iterator<String> statGarit =
                staticGuards.iterator();
            statGarit.hasNext(); ) {
                String g = statGarit.next();
                if(statGarit.hasNext())
                    promelaString = promelaString.concat(g + "
                        && ");
                else
                    promelaString = promelaString.concat(g +
                        ")-> \n");
            }
        }
        else{
            promelaString = promelaString.concat("
                if\n");
            promelaString = promelaString.concat("
                ::(true)->\n");
        }
        //Check to see if any more instances remaining,
        //if not find some more (if available)
        promelaString = promelaString.concat("
            if\n");
        promelaString = promelaString.concat("
            ::("+agentName+"_wf_" + wf_name + "_index == -1)
                ->\n");
        String indent = "                    ";
        Iterator<variable> VariableIterator = vars.iterator();
        //Adds the variable bit, needs to be looped
        //via recusion to get right effect
        promelaString = promelaString.concat(indent +
            "printf(\"Assigning variable values for workframe
            "+wf_name+"\\n\");\n");
        promelaGuardWithVariables(VariableIterator, indent,
            agentName);
        promelaString = promelaString.concat(indent + "if\n");
        promelaString = promelaString.concat(indent +
            "::("+agentName+"_wf_"+wf_name+"_index > -1) ->\n");
        indent = indent.concat("    ");
        promelaString = promelaString.concat(indent + agentName +
                    "wfActive(i,"+ID+");\n");
                indent = indent.substring(0,(indent.length()-4));
                promelaString = promelaString.concat(indent + "::else
                    ->\n");
                indent = indent.concat("    ");
                promelaString = promelaString.concat(indent + agentName +
                    "wfNotActive(i,"+ID+");\n");
                indent = indent.substring(0,(indent.length()-4));
                promelaString = promelaString.concat(indent + "fi;\n");
                indent = indent.substring(0,(indent.length()-4));
                promelaString = promelaString.concat(indent + "::else
                    ->\n");
                indent = indent.concat("    ");
                promelaString = promelaString.concat(indent +
                    agentName + "wfActive(i,"+ID+");\n");
                indent = indent.substring(0,(indent.length()-4));
                promelaString = promelaString.concat(indent + "fi;\n");
                indent = indent.substring(0,(indent.length()-4));
                promelaString = promelaString.concat(indent + "::else
                    ->\n");
                indent = indent.concat("    ");
                promelaString = promelaString.concat(indent + "i = 0;\n");
                promelaString = promelaString.concat(indent + agentName +
                    "wfNotActive(i,"+ID+");\n");
                indent = indent.substring(0,(indent.length()-4));
                promelaString = promelaString.concat(indent + "fi;\n");
                promelaString = promelaString.concat(indent + "i = 0;\n");

        }

return promelaString;
}

    public void promelaGuardWithVariables(Iterator<variable> varit,
            String indent, String agentName){
        variable v = varit.next();
        int varNo = v.getVarNo();
        //promelaString = promelaString.concat(indent +
            "int "+ v.getAssociation() + "Counter;\n");
        promelaString = promelaString.concat(indent +
            v.getAssociation() + "Counter = 0;\n");
        promelaString = promelaString.concat(indent + "do\n");
        promelaString = promelaString.concat(indent + "::("+
            v.getAssociation() + "Counter < numberOfEverything &&
            "+v.getAssociation() + "members["+v.getAssociation() +
            "Counter] != 1)->\n");
        promelaString = promelaString.concat(indent + "    " +
            v.getAssociation() + "Counter++;\n");
        promelaString = promelaString.concat(indent + "::("+
        v.getAssociation() + "Counter < numberOfEverything &&
        "+v.getAssociation() + "members["+v.getAssociation() +
        "Counter] == 1)->\n");
    boolean varOfVar = false;
    for (Iterator<guard> Garit = guards.iterator();
            Garit.hasNext(); ) {
        guard gar = Garit.next();
        if(v.getName().equals(gar.getleftAssoc()) &&
                gar.getleftAttr()!= null){
            promelaString = promelaString.concat(indent + "
                printf(\"Checking for ID of %e\\n\",
                agentsObjectsIDs["+v.getAssociation()+
                "Counter]);\n");
            promelaString = promelaString.concat(indent + "
                findID(agentsObjectsIDs["+v.getAssociation()+
                "Counter]);\n");
            promelaString = promelaString.concat(indent +
                "    int var" + varNo + ";\n");
            promelaString = promelaString.concat(indent + "
                var" + varNo + " = searchID;\n");
            varOfVar = true;
        }
    }
    indent = indent.concat("    ");
    if(varit.hasNext()){
        promelaGuardWithVariables(varit, indent, agentName);
    }
    else{
        promelaString = promelaString.concat(indent+"if\n");
        promelaString = promelaString.concat(indent+"::(");
        boolean isFirst = true;
        for (Iterator<String> varGarit = varGuards.iterator();
            varGarit.hasNext(); ) {
            String g = varGarit.next();
            if(varGarit.hasNext())
                promelaString = promelaString.concat(g + " && ");
            else
                promelaString = promelaString.concat(" " + g + ")
                    ->\n");

        }
        //promelaString = promelaString.concat(")->\n");
        indent = indent.concat("    ");
        promelaString = promelaString.concat(indent + ""+
            agentName+"_wf_" + wf_name + "_index++;\n");
        for (Iterator<variable> Varit2 = vars.iterator();
                Varit2.hasNext(); ) {
            variable v2 = Varit2.next();
            varNo = v2.getVarNo();
            promelaString = promelaString.concat(indent +
                ""+agentName+"_wf_"+wf_name + "_var["+agentName+
                "_wf_"+
                 wf_name +"_index].var_elements["+varNo+"] =
                 agentsObjectsIDs["+v2.getAssociation()+
                 "Counter];\n");
            promelaString = promelaString.concat(indent +"
                printf(\"Inserting %e into variable "+v2.getName()
                +" of type "+v2.getType()+
                " \\n\", agentsObjectsIDs["+v2.getAssociation()+
                "Counter]);\n");
            if(v2.getType().equals("forone"))
                promelaString = promelaString.concat(indent +
                v2.getAssociation() +
                "Counter = numberOfEverything+1;\n");
            //varNo++;
        }
        indent = indent.substring(0,(indent.length()-4));
        promelaString = promelaString.concat(indent + "::else
            ->\n");
        promelaString = promelaString.concat(indent + "    skip;
            \n");
        promelaString = promelaString.concat(indent + "fi;\n");
    }
    promelaString = promelaString.concat(indent +
        v.getAssociation() + "Counter++;\n");
    indent = indent.substring(0,(indent.length()-4));
    promelaString = promelaString.concat(indent + "::else ->\n");
    indent = indent.concat("    ");
    promelaString = promelaString.concat(indent + "break; \n");
    indent = indent.substring(0,(indent.length()-4));
    promelaString = promelaString.concat(indent + "od; \n");
    indent = indent.concat("    ");


}
//Finds out if workframe has a collectAll in it and will
//therefore mark all events
public void hasCollectAll(){
    for(Iterator<variable> varit = vars.iterator();
            varit.hasNext();){
        variable v = varit.next();
        //If workframe has a collectAll variable then flag
        //all its events with collectAll
        if(v.getType().equals("collectall")){
            //System.out.println("Found a CollectAll");
            containsCollectAll = true;
            for(Iterator<event> eventit = events.iterator();
                    eventit.hasNext();){
```

```
                event e = eventit.next();
                e.inCollectAll();
            }
        }

    }
}

public boolean getContainsCollectAll(){
    return containsCollectAll;
}

public String getName()
 {
    return wf_name;
 }
public int getID()
 {
    return ID;
 }
public String getRepeat(){
    return wf_repeat;
}
public int getRepeatValue()
 {
    if(wf_repeat.equals("false"))
        return 1;
    else if(wf_repeat.equals("once"))
        return 2;
    else if(wf_repeat.equals("true"))
        return 3;
    else
        return 1;
 }
public int getPriority()
 {
    return wf_priority;
 }
 public Set getDetectables()
 {
    return detectables;
 }
 public Set getVariables()
 {
    return vars;
 }
 public Set getGuards()
 {
    return guards;
```

```
     }
     public List getEvents()
     {
        return events;
     }
    public int numOfEvents()
    {
        return events.size();
    }
}
```

## C.15 Detectables

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*The detectable of a workframe.  These are stored in sets
inside workframes*/

import java.util.*;
class detectable{

    String name; // Name of the detectable
    /*  Store the guard condtion of the detectable. e.g.
    "Alex.hunger = current.hunger".  leftOwner = Alex,
    leftAttribute = hunger, rightOwner = current and rightAttribute =
    hunger.  However if it was "Alex.hunger = 10" then 10 = value.
    */
    String leftOwner;
    String leftAttribute;
    String rightOwner;
    String rightAttribute;
    String value;
    // used incase of relation e.g. "Alex hasCheck check"
    // then math would be hasCheck
    String math;
    int dc; // Decision condidtion
    String type; // Abort, impasse, continue or complete.
    String f_name;

    boolean hasVariable = false;
    int leftVar = -1;
    int rightVar = -1;
    int valVar = -1;

    // Following are used for Promela
```

```
int AgentID; // ID of agent who owns it
int wfNumber; // ID number of the workframe it is in
int ID; // Its own ID number

String beliefGuardToString = "";
String factGuardToString = "";
String beliefUpdateToString = "";
String factUpdateToString = "";
String printBelief = "";
String theVariable = ""; //might not be needed
//Left side of guard as in does left = right?
String leftSide = "";
String rightSide = "";

public detectable(int new_ID, String new_name, String
        new_leftOwner, String new_leftAttribute, String
        new_rightOwner, String new_rightAttribute, String
        new_value, String new_math, String new_type, int
        new_dc, int new_wfNumber){
    ID = new_ID;
    name = new_name;
    leftOwner = new_leftOwner;
    leftAttribute = new_leftAttribute;
    rightOwner = new_rightOwner;
    rightAttribute = new_rightAttribute;
    value = new_value;
    math = new_math;
    type = new_type;
    dc = new_dc;
    wfNumber = new_wfNumber;

    if(math.equals("="))
        math = "==";
}

public String toPromelaString(String[] identificationNumbers,
        Set workframes, String agent) {
    Set<variable> vars = new HashSet();
    for(Iterator<workframe> workit = workframes.iterator();
            workit.hasNext();){
        workframe w = workit.next();
        if(w.getID() == wfNumber){
            vars = w.getVariables();
            f_name = w.getName();

        }
    }
    boolean leftOwnerCurrent = false;
    try{
            if(leftOwner.equals("current")){
                leftOwner = agent;
                leftOwnerCurrent = true;
            }
        }
        catch(Exception e){}
        boolean rightOwnerCurrent = false;
        try{
            if(rightOwner.equals("current")){
                rightOwner = agent;
                rightOwnerCurrent = true;
            }
        }
        catch(Exception e){}

        factGuardToString =  guardToString("fact", agent, vars);
        beliefUpdateToString = "";
        factUpdateToString = "";
        beliefUpdateToString =  beliefUpdateToString.concat(
            BeliefUpdateToString(agent, agent, vars));
        factUpdateToString =  factUpdateToString.concat(
            BeliefUpdateToString("fact", agent, vars));

        if(leftOwnerCurrent == true){
            leftOwner = "current";
        }
        if(rightOwnerCurrent == true){
            rightOwner = "current";
        }

        return factGuardToString;
    }


    public String guardToString(String agent, String agentName,
            Set vars){
        boolean leftAssocCurrent = false;
        boolean rightAssocCurrent = false;
        String guard = "";
        if(leftOwner.equals("current"))
        {
            leftOwner = agentName;
            leftAssocCurrent = true;
        }

        if(rightOwner != null){
            if(rightOwner.equals("current"))
            {
                rightOwner = agentName;
                rightAssocCurrent = true;
            }
        }

        //  Loop through to see if the guard contains a variable

        for(Iterator<variable> Varit = vars.iterator();
                Varit.hasNext();)
        {
            variable v = Varit.next();
            String varName = v.getName();
            if(leftOwner != null && leftOwner.equals(varName))
            {
                leftVar = v.getVarNo();
                hasVariable = true;
            }
            if(rightOwner != null && rightOwner.equals(varName ))
            {
                rightVar = v.getVarNo();
                hasVariable = true;
            }
            if(value != null && value.equals(varName))
            {
                valVar = v.getVarNo();
                hasVariable = true;
            }
        }
        if(hasVariable == false){
            if(!math.equals("=")&&!math.equals("==")&&!math.equals
                    ("!=")&&!math.equals("<")&&!math.equals(">")&&
                    !math.equals("<=")&&!math.equals(">=")){
                guard = guard.concat(agent + "_" + math + "[" +
                    leftOwner + "ID].elements["+rightOwner+"ID]
                        == 1" );
            }
            else{
                guard = guard.concat(agent + "_"+leftAttribute+
                    "["+/*left*/leftOwner+"ID] ");

                if(math.equals("="))
                    guard = guard.concat("==");
                else
                    guard = guard.concat(math);

                if(rightOwner != null){
                guard = guard.concat(rightOwner + "_"+
                    rightAttribute+"["+/*right*/rightOwner+"ID]");
                }
                else
```

```
            guard = guard.concat(value);
        }
    }
    else {
        // is it a relation
        if(!math.equals("==")&&!math.equals("!=")&&!math.equals
                ("<")&&!math.equals(">")&&!math.equals("<=")&&
                !math.equals(">=")&&!math.equals("=")){
            if(leftVar > -1 && valVar == -1){
                theVariable = agentName+"_wf_"+f_name+"_var["+
                    agentName+"_wf_"+f_name+"_index].var_elements
                    ["+leftVar+"]";
                guard = agent + "_" + math + "[searchID].elements
                ["+rightOwner+"ID"+"] == 1";
            }
            else if(leftVar == -1 && valVar > -1){
                theVariable = agentName+"_wf_"+f_name+"_var["+
                    agentName+"_wf_"+f_name+"_index].var_elements
                    ["+rightVar+"]";
                guard = agent + "_" + math + "["+leftOwner+"ID].
                    elements[searchID]"+"] == 1";
            }
            else{
                theVariable = agentName+"_wf_"+f_name+"_var["+
                    agentName+"_wf_"+f_name+"_index].
                    var_elements["+leftVar+"]";
                guard = agent + "_" + math + "[searchID]"+"].
                    elements["+agentName+"_wf_"+f_name+"_var
                    ["+agentName+"_wf_"+f_name+"_index].
                    var_elements["+rightVar+"]"+"] == 1";
            }
        }
        // not a relation
        else{
            if(leftVar > -1){
                theVariable = agentName+"_wf_"+f_name+"_var
                    ["+agentName+"_wf_"+f_name+"_index].
                    var_elements["+leftVar+"]";
                leftSide = agent + "_"+leftAttribute+
                    "[searchID]";
            }
            else{
                leftSide = agent + "_"+leftAttribute+"["+
                    leftOwner+"ID]";
            }

            if(rightVar > -1){
                theVariable = agentName+"_wf_"+f_name+"_var["+
                    agentName+"_wf_"+f_name+"_index].
                    var_elements["+valVar+"]";
                rightSide = agent + "_"+leftAttribute+"
                    [searchID]";
            }
            else if(rightOwner != null)
                rightSide = agent + "_"+leftAttribute+
                    "["+rightOwner+"ID] ";
            else{
                rightSide = value;
            }

            /*if(leftVar > -1)
                leftSide = agent + "_"+leftAttribute+"["+
                    leftOwner+"ID].elements["+agentName+
                    "_wf_"+f_name+"_var["+agentName+"_wf_"+
                    f_name+"_index].var_elements["+rightVar
                    +"]"+"]";
            else
                leftSide = agent + "_"+leftAttribute+"["+
                    leftOwner+"ID]";

            if(rightVar > -1)
                rightSide = agent + "_"+leftAttribute+"["+
                    leftOwner+"ID].elements["+agentName+"
                    _wf_"+f_name+"_var["+agentName+"_wf_"
                    +f_name+"_index].var_elements["
                    +leftVar+"]"+"]";
            else if(rightOwner != null)
                rightSide = agent + "_"+leftAttribute+"["+
                    rightOwner+"ID] ";
            else{
                rightSide = value;
            }
            */

            guard = guard.concat(leftSide + math + rightSide);
        }
    }
    if(leftAssocCurrent)
        leftOwner = "current";
    if(rightAssocCurrent)
        rightOwner = "current";

    return guard;


}

public String BeliefUpdateToString(String agent, String agentName,
    Set vars){

boolean leftAssocCurrent = false;
String update = "";
if(leftOwner.equals("current"))
{
    leftOwner = agentName;
    leftAssocCurrent = true;
}

//  Loop through to see if the guard contains a variable
for(Iterator<variable> Varit = vars.iterator();
        Varit.hasNext();)
{
    variable v = Varit.next();

    String varName = v.getName();
    if(leftOwner != null && leftOwner.equals(varName))
    {
        leftVar = v.getVarNo();
        hasVariable = true;
    }

    if(value != null && value.equals(varName))
    {
        valVar = v.getVarNo();
        hasVariable = true;
    }
}
if(hasVariable == false){
    if(!math.equals("=")&&!math.equals("==")&&!math.equals
        ("!=")&&!math.equals("<")&&!math.equals(">")&&!math.
        equals("<=")&&!math.equals(">=")){update = update.
        concat(agent + "_" + math + "[" + leftOwner + "ID].
        elements["+rightOwner+"ID] == 1" );
    }
    else{
        update = update.concat(agent + "_"+leftAttribute+
        "["+/*left*/leftOwner+"ID] ");
    }
}
else {
    // is it a relation
    if(!math.equals("==")&&!math.equals("!=")&&!math.
            equals("<")&&!math.equals(">")&&!math.equals
            ("<=")&&!math.equals(">=")){
        if(leftVar > -1 && valVar == -1){
            update = agent + "_" + math + "["+agentName+
            "_wf_"+f_name+"_var["+agentName+"_wf_"+f_name+
                "_index].var_elements["+leftVar+"]"+"].elements
                ["+rightOwner+"ID"+"] == 1";
            }
            else if(leftVar == -1 && valVar > -1){
                update = agent + "_" + math + "["+leftOwner+"ID]
                .elements["+agentName+"_wf_"+f_name+"_var["+
                agentName+"_wf_"+f_name+"_index].var_elements["+
                rightVar+"]"+"] == 1";
            }
            else{
                update = agent + "_" + math + "["+agentName+
                "_wf_"+f_name+"_var["+agentName+"_wf_"+f_name+
                "_index].var_elements["+leftVar+"]"+"].elements
                ["+agentName+"_wf_"+f_name+"_var["+agentName+
                "_wf_"+f_name+"_index].var_elements["+rightVar
                +"]"+"] == 1";
            }
        }
        // not a relation
        else{
            if(leftVar > -1)
                leftSide = agent + "_"+leftAttribute+
                    "[searchID]";
            else
                leftSide = agent + "_"+leftAttribute+
                    "["+leftOwner+"ID]";

            update = update.concat(leftSide);
        }
    }
    if(leftAssocCurrent)
        leftOwner = "current";

    return update;
}

public String getName(){
    return name;
}
public String getLeftOwner(){
    return leftOwner;
}
public String getRightOwner(){
    return rightOwner;
}
public String getLeftAttribute(){
    return leftAttribute;
}
public String getRightAttribute(){
        return rightAttribute;
    }
    public String getValue(){
        return value;
    }
    public String getMath(){
        return math;
    }
    public String getType(){
        return type;
    }

    public int getID(){
        return ID;
    }
    public int getwfNumber(){
        return wfNumber;
    }
    public String getBeliefGuardToString(){
        return beliefGuardToString;
    }

    public String getFactGuardToString(){
        return factGuardToString;
    }
    public String getBeliefUpdateToString(){
        return beliefUpdateToString;
    }
    public String getFactUpdateToString(){
        return factUpdateToString;
    }
    public boolean getHasVariable(){
        return hasVariable;
    }
    public String getTheVariable(){
        return theVariable;
    }
}
```

## C.16 Thoughtframes

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
```

```
Thoughtframe details.  Relations not yet implemented for thoughtframes
yet.
*/

import java.util.*;
class thoughtframe
{
    public String agent; // name of agent it belongs to
    public String tf_name; // name of the thoughtframe
    public String tf_repeat = "false"; // repeat: true, once, false
    public int tf_priority = 0; // priority of the thoughtframe
    // The variables it contains
    public Set<variable> vars = new HashSet();
    // Guard condition of the thoughtframe
    public Set<guard> guards = new HashSet();
    // Events contained within the thoughtframe
    public List<event> events = new ArrayList();

    /****************
    *For Promela use*
    ****************/
    public int ID; // thoughtframes ID number
    public boolean containsCollectAll = false;
    // Details of all other agents/objects
    public Set<agent> mas = new HashSet();
     // Guards which don't contain variables
    Set<String> staticGuards = new HashSet<String>();
    // Guards which contain variables
    Set<String> varGuards = new HashSet<String>();
    // Names of the variables
    public Set<String> VarNames = new HashSet();
    String promelaString = "";

    public thoughtframe(){}

    public thoughtframe(Set<agent> new_mas, int new_ID, String
        new_agent, String new_name, String new_repeat, int
        new_priority, Set<variable> new_variables, Set<guard>
        new_guards, List<event> new_events) {

        mas = new_mas;
        ID = new_ID;
        agent = new_agent;
        tf_name = new_name;
        tf_repeat = new_repeat;
        tf_priority = new_priority;
        vars = new_variables;

        guards = new_guards;
        events = new_events;
     }

    public String toPromelaString(String[]
        identificationNumbers, String agentName){
        promelaString = "";
        varGuards.clear();
        staticGuards.clear();
        boolean hasVariable;
        //  Cycle through guards to find which contain variables
        for (Iterator<guard> Garit = guards.iterator();
                Garit.hasNext(); ) {
            guard gar = Garit.next();
            String guardToString = gar.toPromela
                (identificationNumbers, vars, agentName, agentName);
            if(gar.getHasVariable())
                varGuards.add(guardToString);
            else
                staticGuards.add(guardToString);
        }
        if(vars.size() == 0){
            promelaString = promelaString.concat("
                /* thoughtframe "+ tf_name +", has no variables*/\n");
            promelaString = promelaString.concat("        if\n");
            promelaString = promelaString.concat("        ::(");
            for (Iterator<String> statGarit = staticGuards.iterator();
                    statGarit.hasNext(); ) {
                String g = statGarit.next();
                if(statGarit.hasNext())
                    promelaString = promelaString.concat(g + " && ");
                else
                    promelaString = promelaString.concat(
                        g + ")-> \n");
            }

            promelaString = promelaString.concat("
                i = 0;\n");

            promelaString = promelaString.concat("
                " + agentName + "tfActive(i, "+ID+")\n");
            promelaString = promelaString.concat("
                printf(\"--Thoughtframe "+tf_name+" is
                active\\n\")\n");
            promelaString = promelaString.concat("
                ::else->\n");
            promelaString = promelaString.concat("
                i = 0;\n");
            promelaString = promelaString.concat("
                " + agentName + "tfNotActive(i, " + ID + ")\n");
        promelaString = promelaString.concat("        fi;\n");
    }
    else{
        promelaString = promelaString.concat("
            /* thoughtframe "+ tf_name +", has variables*/\n");
        if(!staticGuards.isEmpty()){
            promelaString = promelaString.concat("        if\n");
            promelaString = promelaString.concat("        ::(");
            for (Iterator<String> statGarit = staticGuards.
                    iterator(); statGarit.hasNext(); ) {
                String g = statGarit.next();
                if(statGarit.hasNext())
                    promelaString = promelaString.
                        concat(g + " && ");
                else
                    promelaString = promelaString.concat(
                        g + ")-> \n");
            }
        }
        else{
            promelaString = promelaString.concat("
                if\n");
            promelaString = promelaString.concat("
                ::(true)->\n");
        }
        promelaString = promelaString.concat("            if\n");
        promelaString = promelaString.concat("
            ::("+agentName+"_tf_"+tf_name+ "_index == -1) ->\n");
        String indent = "                ";
        Iterator<variable> VariableIterator = vars.iterator();
        //Adds the variable bit, needs to be looped via
        //recusion to get right effect
        promelaString = promelaString.concat(indent +"
            printf(\"Assigning variable values for thoughtframe
            "+tf_name+"\\n\");\n");
        promelaGuardWithVariables(VariableIterator, indent,
            agentName);
        promelaString = promelaString.concat(indent + "if\n");
        promelaString = promelaString.concat(indent + "::("+
            agentName+"_tf_"+tf_name+"_index > -1) ->\n");
        indent = indent.concat("    ");
        promelaString = promelaString.concat(indent + agentName +
            "tfActive(i,"+ID+");\n");
        promelaString = promelaString.concat(indent + "printf(
            \"--Thoughtframe "+tf_name+" is active1\\n\")\n");
        indent = indent.substring(0,(indent.length()-4));
        promelaString = promelaString.concat(indent + "::else
            ->\n");
        indent = indent.concat("    ");
```

```
            promelaString = promelaString.concat(indent + agentName + "
                tfNotActive(i,"+ID+");\n");
            indent = indent.substring(0,(indent.length()-4));
            promelaString = promelaString.concat(indent + "fi;\n");
            indent = indent.substring(0,(indent.length()-4));
            promelaString = promelaString.concat(indent + "::else
                ->\n");
            indent = indent.concat("    ");
            promelaString = promelaString.concat(indent + agentName+"
                tfActive(i,"+ID+");\n");
            promelaString = promelaString.concat(indent + "printf(\"--
                Thoughtframe "+tf_name+" is active2\\n\")\n");
            indent = indent.substring(0,(indent.length()-4));
            promelaString = promelaString.concat(indent + "fi;\n");
            indent = indent.substring(0,(indent.length()-4));
            promelaString = promelaString.concat(indent + "::else
                ->\n");
            indent = indent.concat("    ");
            promelaString = promelaString.concat(indent + "i = 0;\n");
            promelaString = promelaString.concat(indent + agentName +
                "tfNotActive(i,"+ID+");\n");
            indent = indent.substring(0,(indent.length()-4));
            promelaString = promelaString.concat(indent + "fi;\n");
            promelaString = promelaString.concat(indent + "i = 0;\n");

    }

    return promelaString;
}

public void promelaGuardWithVariables(Iterator<variable> varit,
        String indent, String agentName){
    variable v = varit.next();
    int varNo = v.getVarNo();
    promelaString = promelaString.concat(indent +
        v.getAssociation() + "Counter = 0;\n");
    promelaString = promelaString.concat(indent + "do\n");
    promelaString = promelaString.concat(indent + "
        ::("+v.getAssociation() + "Counter <
        numberOfEverything && "+v.getAssociation() +
        "members["+v.getAssociation() + "Counter] != 1)->\n");
    promelaString = promelaString.concat(indent + "    "
        + v.getAssociation() + "Counter++;\n");
    promelaString = promelaString.concat(indent +
        "::("+v.getAssociation() + "Counter <
        numberOfEverything && "+v.getAssociation() +
        "members["+v.getAssociation() + "Counter] == 1)->\n");
    boolean varOfVar = false;
    for (Iterator<guard> Garit = guards.iterator();
            Garit.hasNext(); ) {
        guard gar = Garit.next();
        if(v.getName().equals(gar.getleftAssoc()) &&
                gar.getleftAttr()!= null){
            promelaString = promelaString.concat(indent + "
                printf(\"Checking for ID of %e\\n\",
                agentsObjectsIDs
                ["+v.getAssociation()+"Counter]);\n");
            promelaString = promelaString.concat(indent + "
                findID(agentsObjectsIDs["+v.getAssociation()+
                "Counter]);\n");
            promelaString = promelaString.concat(indent + "
                int var" + varNo + ";\n");
            promelaString = promelaString.concat(indent + "
                var" + varNo + " = searchID;\n");
            varOfVar = true;
        }
    }
    indent = indent.concat("    ");
    if(varit.hasNext()){
        promelaGuardWithVariables(varit, indent, agentName);
    }
    else{
        promelaString = promelaString.concat(indent+"if\n");
        promelaString = promelaString.concat(indent+"::(");

        boolean isFirst = true;
        for (Iterator<String> varGarit = varGuards.iterator();
                varGarit.hasNext(); ) {
            String g = varGarit.next();
            if(varGarit.hasNext())
                promelaString = promelaString.concat(g + " && ");
            else
                promelaString = promelaString.concat(" " + g + ")
                ->\n");
        }

        indent = indent.concat("    ");
        promelaString = promelaString.concat(indent + ""+agentName
            +"_tf_" + tf_name + "_index++;\n");
        for (Iterator<variable> Varit2 = vars.iterator();
                Varit2.hasNext(); ) {
            variable v2 = Varit2.next();
            varNo = v2.getVarNo();
            promelaString = promelaString.concat(indent +""+
                agentName+"_tf_"+tf_name + "_var["+agentName+"
                _tf_"+ tf_name +"_index].var_elements["+varNo+"]
                = agentsObjectsIDs["+v2.getAssociation()+
                "Counter];\n");
```

```
}
            promelaString = promelaString.concat(indent +
                "printf(\"    Inserting %e into variable "+v2.
                getName()+" of type "+v2.getType()+" \\n\",
                agentsObjectsIDs["+v2.getAssociation()+"Counter])
                ;\n");
            if(v2.getType().equals("forone"))
                promelaString = promelaString.concat(indent +
                v2.getAssociation() + "Counter =
                numberOfEverything+1;\n");
        }
        indent = indent.substring(0,(indent.length()-4));
        promelaString = promelaString.concat(indent + "::else
            ->\n");
        promelaString = promelaString.concat(indent + "
            skip;\n");
        promelaString = promelaString.concat(indent + "
            fi;\n");
    }
    promelaString =promelaString.concat(indent+v.getAssociation()
        + "Counter++;\n");
    indent = indent.substring(0,(indent.length()-4));
    promelaString = promelaString.concat(indent + "::else ->\n");
    indent = indent.concat("    ");
    promelaString = promelaString.concat(indent + "break; \n");
    indent = indent.substring(0,(indent.length()-4));
    promelaString = promelaString.concat(indent + "od; \n");
    indent = indent.concat("    ");


}

//Finds out if thoughtframe has a collectAll in it and will
//therefore mark all events
public void hasCollectAll(){
    for(Iterator<variable> varit = vars.iterator();
            varit.hasNext();){
        variable v = varit.next();
        if(v.getType().equals("collectall")){
            //System.out.println("Found a CollectAll");
            containsCollectAll = true;
            for(Iterator<event> eventit = events.iterator();
                    eventit.hasNext();){
                event e = eventit.next();
                e.inCollectAll();
            }
        }

    }
}
```

```
    public boolean getContainsCollectAll(){
        return containsCollectAll;
    }
    public String getName()
     {
        return tf_name;
     }
    public int getID()
     {
        return ID;
     }
    public String getRepeat(){
        return tf_repeat;
    }
    public int getRepeatValue()
     {
        if(tf_repeat.equals("false"))
            return 1;
        else if(tf_repeat.equals("once"))
            return 2;
        else if(tf_repeat.equals("true"))
            return 3;
        else
            return 1;
     }
    public int getPriority()
     {
        return tf_priority;
     }




    public Set getVariables()
     {
        return vars;
     }
     public Set getGuards()
     {
        return guards;
     }
     public List getEvents()
     {
        return events;
     }
}
```

## C.17 Guard Conditions

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Stores the guard conditions of Workframes, Thoughtframes and
Detectables.
*/

import java.util.*;
class guard
{
    // Temp variable to hold ID number of the agent the guard
    // belongs to, -1 means empty
    int leftID = -1;
    int rightID = -1;
    boolean hasVariable = false;
    // Who the leftAttr of the left hand variable is about
    String leftAssoc;
    String leftAttr; // left hand attribute
    //  Right hand attribute attribute if it is string or int
    String value;
    String mathSymbol; // if it is > or < or =
    String type; // not, known, knownval etc.
    String rightAssoc; // right hand attribute leftAssoc
    String rightAttr; // Right hand attribute


    String guardToString = "";
    String leftSide = ""; // Used to interpret into Promela
    String rightSide = ""; // Used to interpret into Promela

    public guard(){}

    public guard(String new_type, String new_mathSymbol, String
            new_leftAssoc, String new_rightAssoc, String new_leftAttr,
            String new_rightAttr, String new_value)
    {
        type = new_type;
        mathSymbol = new_mathSymbol;
        leftAssoc = new_leftAssoc;
        leftAttr = new_leftAttr;
        value = new_value;
        rightAssoc = new_rightAssoc;
        rightAttr = new_rightAttr;
    if(mathSymbol.equals("="))
        mathSymbol = "==";

}

public String relationInGuard(Set<relation> relations){
    String relationName = "";
    if(!mathSymbol.equals("==")&&!mathSymbol.equals("!=")&&
            !mathSymbol.equals("<")&&!mathSymbol.equals(">")&&
            !mathSymbol.equals("<=")&&!mathSymbol.equals(">=")){
        for(Iterator<relation> relit = relations.iterator();
                relit.hasNext(); ) {
            relation rel = relit.next();
            if(rel.getName().equals(mathSymbol)){
                relationName = mathSymbol;
                break;
            }
        }
    }
    return relationName;
}

public String toPromela(String[] identificationNumbers, Set vars,
        String agent, String agentObject)
{
    rightAttr + " " + value);
    boolean leftAssocCurrent = false;
    boolean rightAssocCurrent = false;
    guardToString = "";
    if(leftAssoc.equals("current"))
    {
        leftAssoc = agent;
        leftAssocCurrent = true;
    }

    if(rightAssoc != null){
        if(rightAssoc.equals("current"))
        {
            rightAssoc = agent;
            rightAssocCurrent = true;
        }
    }

    //  Loop through to see if the guard contains a variable
    for(Iterator<variable> Varit = vars.iterator();
            Varit.hasNext();)
    {
        variable var = Varit.next();
    String varName = var.getName();
    if(leftAssoc != null && leftAssoc.equals(varName))
    {
        hasVariable = true;
    // if(leftAttr == null){
            leftAssoc = var.getAssociation();
        //}
    }
    if(rightAssoc != null && rightAssoc.equals(varName ))
    {
        hasVariable = true;
        if(rightAttr == null){
            rightAssoc = var.getAssociation();

        }
    }
    if(value != null && value.equals(varName))
    {
        hasVariable = true;
        if(rightAttr == null){
            rightAssoc = var.getAssociation();

        }
    }
}
if(hasVariable == false){
    if(type.equals("knownval"))
    {
        if(!mathSymbol.equals("==")&&!mathSymbol.equals("!=")
                &&!mathSymbol.equals("<")&&!mathSymbol.equals
                (">")&&!mathSymbol.equals("<=")&&!mathSymbol.
                equals(">=")){
            guardToString = guardToString.concat(agentObject +
                "_" + mathSymbol + "[" + leftAssoc + "ID].
                elements["+rightAssoc+"ID] == 1" );
        }
        else{
            guardToString = guardToString.concat(agentObject +
                "_"+leftAttr+"["+/*left*/leftAssoc+"ID] ");

            if(mathSymbol.equals("="))
                guardToString = guardToString.concat("==");
            else
                guardToString = guardToString.concat
                    (mathSymbol);

            if(rightAssoc != null){
            guardToString = guardToString.concat(rightAssoc +
                "_"+rightAttr+"["+/*right*/rightAssoc+"ID ] ");
            }
            else
                guardToString = guardToString.concat(value);
        }
    }
}
else {
    boolean leftVarNoAtt = false;
    boolean rightVarNoAtt = false;
    boolean leftVarAtt = false;
    boolean rightVarAtt = false;
    // is it a relation
    if(!mathSymbol.equals("==")&&!mathSymbol.equals("!=")
            &&!mathSymbol.equals("<")&&!mathSymbol.equals
            (">")&&!mathSymbol.equals("<=")&&!mathSymbol.
            equals(">=")){
        leftSide = leftAssoc + "ID"; // Set default
        rightSide = rightAssoc + "ID"; // Set default
        for(Iterator<variable> Varit = vars.iterator();
                Varit.hasNext();)
        {
            variable var = Varit.next();
            String varAssoc = var.getAssociation();
            // if left side is the variable then add counter
            if(leftAssoc.equals(varAssoc))
                leftSide = varAssoc + "Counter";
            // if right side is the variable then add counter
            if(rightAssoc.equals(varAssoc))
                rightSide = varAssoc + "Counter";
        }
        guardToString = agentObject + "_" + mathSymbol + "[" +
            leftSide +"].elements["+ rightSide+"] == 1";
    }
    // not a relation
    else{
        leftSide = leftAssoc + "ID"; // Set default
        rightSide = rightAssoc + "ID"; // Set default

        for(Iterator<variable> Varit = vars.iterator();
                Varit.hasNext();)
        {
            variable var = Varit.next();
            String varAssoc = var.getAssociation();
            try{
                if(leftAssoc.equals(varAssoc) && leftAttr
                        == null)
                {
                    leftSide = "agentsObjectsIDs" + "[" +
                        varAssoc + "Counter]";
                }
            }
            catch(Exception e){}
            try{
                if(leftAssoc.equals(varAssoc) &&
                    leftAttr != null)
                {
                    leftSide = agentObject + "_" +
                        leftAttr + "[" + varAssoc + "Counter]";
                }
            }
            catch(Exception e){}
            try{
                if(rightAssoc.equals(varAssoc) &&
                    rightAttr == null)
                {
                    rightSide = "agentsObjectsIDs" + "["
                        + varAssoc + "Counter]";
                }
            }
            catch(Exception e){}
            try{
                if(rightAssoc.equals(varAssoc) && rightAttr
                    != null)
                {
                    rightSide = agentObject + "_" + rightAttr
                        + "[" + varAssoc + "Counter]";
                }
            }
            catch(Exception e){}
        }

        if(rightAssoc == null && rightAttr == null &&
                value != null)
            rightSide = value;

        guardToString = guardToString.concat(leftSide +
            mathSymbol + rightSide);
    }
}
if(leftAssocCurrent)
    leftAssoc = "current";
if(rightAssocCurrent)
    rightAssoc = "current";

return guardToString;
}
```

```
    public boolean getHasVariable()
    {
        return hasVariable;
    }

    public String getGuardToString()
    {
        return guardToString;
    }

    public String getleftAssoc()
    {
        return leftAssoc;
    }
    public String getrightAssoc()
    {
        return rightAssoc;
    }
    public String getleftAttr()
    {
        return leftAttr;
    }
    public String getValue()
    {
        return value;
    }
    public String getMathSymbol()
    {
        return mathSymbol;
    }
    public String getType()
    {
        return type;
    }
    public String getLeftSide(){
        return leftSide;
    }
}
```

## C.18 Variables

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Stores the variables of a workframe or thoughtframe.
```

```
*/
import java.util.Stack;

class variable
{

    String name; // name of the variable
    String type; // forone, foreach, collectall etc
    String association; //  Which class of objects or group of agents

    /****************
    *For Promela use*
    *****************/
    int varNo; // ID number for the variable

    public variable()
    {}

    public variable(int new_varNo, String new_type,
            String new_association, String new_name)
    {
        varNo = new_varNo;
        name = new_name;
        type = new_type;
        association = new_association;
    }

    public String getName()
    {
        return name;
    }
    public String getType()
    {
        return type;
    }
    public String getAssociation()
    {
        return association;
    }
    public int getVarNo(){
        return varNo;
    }
}
```

## C.19 Geography: Area Definitions

```
/**
*Author: Richard Stocker
```

```
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Simple class to store the area definition.
*/

import java.util.*;
class areaDefs
{
String name; // Name of the area definition
String ex; // What it extends

    public areaDefs(String new_name, String new_extends){

        name = new_name;
        ex = new_extends;

    }

    public String getName(){
        return name;
    }
}
```

## C.20 Geography: The Locations

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Perhaps would have been better labelling this as "area".
Essentially holds all the details of the areas that an agent/object
can visit.
*/

import java.util.*;
class locations
{
    String name; // Name of the area
    String instanceOf; // areaDef it belongs to
    String partOf; // area it is part of

    int locID; // Promela ID number
    public locations(String new_name, String new_instanceOf, String
        new_partOf, int new_locID){

        name = new_name;
        instanceOf = new_instanceOf;
        partOf = new_partOf;
        locID = new_locID;

        //System.out.println(name+" = " + locID);

    }

    public String getName(){
        return name;
    }

    public int getID(){
        return locID;
    }

    public String getInstanceOf(){
        return instanceOf;
    }

    public String getPartOf(){
        return partOf;
    }

}
```

## C.21 Geography: Paths between Areas

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

/*
Details of which area connects to what.
*/

class path
{
    String area1;
    String area2;
    int distance;

    public path(String new_area1, String new_area2,
            int new_distance){
        area1 = new_area1;
        area2 = new_area2;
        distance = new_distance;
    }

    public String getArea1(){
        return area1;
    }

    public String getArea2(){
        return area2;
    }
    public int getDist(){
        return distance;
    }
}
```

## C.22 Geography: Calculating Undefined Routes

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2012
**/

import java.util.*;
class Dijkstra
{
    // Set of all locations
    Set<locations> locs = new HashSet<locations>();
    int[][] adjacency;  // The adjacency matrix

    public Dijkstra(Set new_locs, int[][] new_adjacency){
        locs = new_locs;
        adjacency = new_adjacency;
    }

    public int[][] findShortestPaths(){

        int distance = 0;
        int temp;
        int minDist = -1;
        int dist[];
        int visit[];
        int k = 0;
        dist = new int[locs.size()];
        visit = new int[locs.size()];
        // store the new matrix
```

```
int[][] newAdjacency = new int[locs.size()][locs.size()];
// Set all values as 99999 in new matrix
for(int i = 0;i<locs.size();i++){
    for(int j = 0;j<locs.size();j++){
        newAdjacency[i][j] = 99999;
    }
}

//loop through all locations, represents the "from"
for(int source = 0;source<locs.size();source++){
    //System.out.println("Node = " + source);
    for(int destination = 0; destination <locs.size()
            ;destination++){
        //Set all locations as not visited
        for(int i = 0;i<locs.size();i++){
            visit[i] = 0;
        }
        //set all initial distances as max
        for(int i = 0;i<locs.size();i++){
            dist[i] = 99999;
        }
        //Reset currect distance travelled
        distance = 0;
        // Current location is the start location
        int current = source;
 //loop through all locations to check for connection
        boolean found = false;
        while(found == false){
            //Set the current place as visited
            visit[current] = 1;
        //assign distances from all immediate neighbours
            for(int j = 0;j<locs.size();j++){
                if((distance + adjacency[current][j])
                        < dist[j]){
                    dist[j] = adjacency[current][j] +
                        distance;
                }
            }
            // shortest distance in array
            int shortest = -1;
            int next = -1; // index of the shortest distance
            //Loop through to find next node to visit
            for(int i = 0;i<locs.size();i++){
     // Next node has to be shortest tentative distance away
                if(visit[i] == 0 && (dist[i] < shortest
                        || shortest == -1) && dist[i]
                        != 99999){
                    next = i; //assign as next node
                    shortest = dist[i];
```

```
                        }
                    }
                    //if there is a next node and it isn't the
                    //destination, assing new current and loop again
                    if(next != -1 && next != destination){
                        current = next;
                        distance = shortest;
                    }
                    else{ //break loop
                        found = true;
                        if(source == destination){
                            newAdjacency[source][destination] = 0;
                        }
                        else{
                            newAdjacency[source][destination] =
                                dist[destination];
                        }
                    }
                }

            }

        }
        return newAdjacency;

    }


}
```

# Appendix D

# The Brahms Models used in the Case Studies

The following sections provide the reader with the Brahms code used to create the simulations in the case studies. These are provided in the appendix so the reader can get a better insight into how the simulations were programmed, and they also show in detail what can be translated from Brahms into *PROMELA*.

## D.1 Robot Helper Case Study

```
areadef House extends BaseAreaDef { }
areadef Room extends House { }
areadef areaOfInterest extends Room { }
area Geography instanceof World { }
area AlexHouse instanceof House { }
area LivingRoom instanceof Room partof AlexHouse { }
area BedRoom instanceof Room partof AlexHouse { }
area Kitchen instanceof Room partof AlexHouse { }
area BathRoom instanceof Room partof AlexHouse { }
area chair instanceof areaOfInterest partof LivingRoom { }
area frontDoor instanceof areaOfInterest partof LivingRoom { }
area medCabinet instanceof areaOfInterest partof LivingRoom { }
area sinkOne instanceof areaOfInterest partof Kitchen { }
area dishWasher instanceof areaOfInterest partof Kitchen { }
area microWave instanceof areaOfInterest partof Kitchen { }
area medicationBox instanceof areaOfInterest partof LivingRoom { }
area bed instanceof areaOfInterest partof BedRoom { }
area toilet instanceof areaOfInterest partof BathRoom { }
area sinkTwo instanceof areaOfInterest partof BathRoom { }
area careCentre instanceof House {}

path chair_to_from_medCabinet {
    area1: chair;
    area2: medCabinet;
    distance: 10;
}
```

```
path chair_to_from_sinkOne {
    area1: chair;
    area2: sinkOne;
    distance: 10;
}

path chair_to_from_bed {
    area1: chair;
    area2: bed;
    distance: 15;
}

path chair_to_from_toilet {
    area1: chair;
    area2: toilet;
    distance: 5;
}

path chair_to_from_frontDoor {
    area1: chair;
    area2: frontDoor;
    distance: 5;
}

path toilet_to_from_sinkTwo {
    area1: toilet;
    area2: sinkTwo;
    distance: 2;
}

path sinkOne_to_from_dishWasher {
    area1: sinkOne;
    area2: dishWasher;
    distance: 2;
}

path dishWasher_to_from_microWave {
    area1: dishWasher;
    area2: microWave;
    distance: 2;
}

path careCentre_to_from_frontDoor {
    area1: careCentre;
    area2: frontDoor;
    distance: 2000;
}
```

```
group person{

}

agent Alex memberof person {
    location: chair;
    attributes:
        public int perceivedtime;
        public int howHungry;
        public int needToilet;
        public int waitingForFood;
        public boolean askedFood;
        public boolean updateAskedFood;
        public boolean hasFood;
        public boolean handsWashed;
        public boolean hasEmptyPlate;
        public boolean hasMedicationA;
        public boolean takeMedicationA;
        public boolean hasTakenMedicationA;
        public boolean toiletFlushed;
        public boolean evacuate;
        public boolean fireDecision;
        public boolean danger;
        public int missedMedicationA;
        public int timeSinceAskedFood;
    initial_beliefs:
        (current.timeSinceAskedFood = 0);
        (current.howHungry = 15);
        (current.fireDecision = false);
        (current.toiletFlushed = true);
        (current.handsWashed = true);
        (current.needToilet = 1);
        (current.askedFood = false);
        (current.waitingForFood = 0);
        (current.updateAskedFood = false);
        (current.perceivedtime = 1);
        (Campanile_Clock.time = 1);
        (current.hasFood = false);
        (current.takeMedicationA = false);
        (current.hasMedicationA = false);
        (current.hasTakenMedicationA = false);
        (current.hasEmptyPlate = false);
        (theEnvironment.fire = false);
        (current.evacuate = false);
    initial_facts:

    activities:
        primitive_activity eat() {
            max_duration: 1000;
        }

        primitive_activity watchTV() {
            max_duration: 2000;
        }

        primitive_activity takeMedication() {
            max_duration: 100;
        }

        communicate askForFood(){
            max_duration: 1;
            with: robotHelper;
            about:
                send(current.askedFood = current.askedFood);
        }

        primitive_activity wait(){
            max_duration: 100;
        }

        primitive_activity goToilet(){
            max_duration: 400;
        }

        primitive_activity flushToilet(){
            max_duration: 3;
        }

        primitive_activity washHands(){
            max_duration: 100;
        }

        move moveToChair() {
             location: chair;
        }

        move moveToMeds() {
             location: medCabinet;
        }

        move moveToBed() {
             location: bed;
        }

        move moveToSinkOne() {
             location: sinkOne;
        }
```

```
    move moveToDishWasher() {
         location: dishWasher;
    }

    move moveToMicroWave() {
         location: microWave;
    }

    move moveToToilet() {
         location: toilet;
    }

    move moveToSinkTwo() {
         location: sinkTwo;
    }

    move moveToFrontDoor() {
         location: frontDoor;
    }


workframes:

    workframe wf_watchTV {
        repeat: true;
        priority: 1;

        detectables:
            detectable stop{
                when(whenever)
                    detect((current.perceivedtime = 10),
                        dc:100)
                    then abort;
            }

        when(knownval(current.perceivedtime < 10) and
            knownval(current.location = chair))
        do {
            watchTV();
        }
    }

    workframe wf_evacuate {
        repeat: true;
        priority: 1000;

        when(knownval(current.evacuate = true))
        do {
            moveToFrontDoor();
        conclude((current.evacuate = false));
        conclude((current.perceivedtime = 10));
    }
}


workframe wf_goToilet {
    repeat: true;
    priority: 4;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.perceivedtime < 10) and
        knownval(current.needToilet > 5))
    do {
        moveToToilet();
        goToilet();
        conclude((current.needToilet = 0));
        conclude((current.handsWashed = false));
        conclude((current.toiletFlushed = false));
    }
}

workframe wf_flushToilet {
    repeat: true;
    priority: 7;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.perceivedtime < 10) and
        knownval(current.location = toilet) and
        knownval(current.toiletFlushed = false))
    do {
        flushToilet();
        conclude((current.toiletFlushed = true));
    }
}

workframe wf_flushToiletTwo {
    repeat: true;
    priority: 3;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.perceivedtime < 10) and
        knownval(current.location != toilet) and
        knownval(current.toiletFlushed = false))
    do {
        moveToToilet();
        flushToilet();
        conclude((current.toiletFlushed = true));
    }
}

workframe wf_forgetFlushToilet {
    repeat: true;
    priority: 7;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.perceivedtime < 10) and
        knownval(current.location = toilet) and
        knownval(current.toiletFlushed = false))
    do {
        conclude((current.toiletFlushed = true), fc:0);
    }
}

workframe wf_washHands {
    repeat: true;
    priority: 6;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.perceivedtime < 10) and
        knownval(current.handsWashed = false))
    do {
        moveToSinkTwo();
        washHands();
        conclude((current.handsWashed = true));
        moveToChair();
    }
}


workframe wf_askForFood {
    repeat: true;
    priority: 9;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.howHungry > 10) and
        knownval(current.perceivedtime < 10) and
        knownval(current.askedFood = false) and
        knownval(current.hasFood = false))
    do {
        conclude((current.askedFood = true));
        askForFood();
        conclude((current.waitingForFood = 0));
    }
}

workframe wf_takeMedicationA {
    repeat: true;
    priority: 10;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.takeMedicationA = true) and
        knownval(current.perceivedtime < 10) and
        knownval(current.location = chair))
    do {
        takeMedication();
        conclude((current.takeMedicationA = false));
        conclude((current.hasMedicationA = false));
        conclude((current.hasTakenMedicationA = true));
    }
}
```

```
workframe wf_DontTakeMedicationA {
    repeat: true;
    priority: 10;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.takeMedicationA = true) and
        knownval(current.perceivedtime < 10) and
        knownval(current.location = chair))
    do {
        takeMedication();
        conclude((current.takeMedicationA = false));
    }
}

workframe wf_eat {
    repeat: true;
    priority: 11;
    detectables:
        detectable stop{
            when(whenever)
                detect((current.perceivedtime = 10),
                    dc:100)
                then abort;
        }
    when(knownval(current.howHungry > 10) and
        knownval(current.perceivedtime < 10) and
        knownval(current.askedFood = true) and
        knownval(current.hasFood = true) and
        knownval(current.location = chair))
    do {
        eat();
        conclude((current.askedFood = false));
        conclude((current.howHungry = 0));
        conclude((current.waitingForFood = 0));
        conclude((current.hasFood = false));
        conclude((current.hasEmptyPlate = true));
        conclude((current.timeSinceAskedFood = 0));
    }
}

workframe wf_atToilet{
    repeat: true;
            detectables:
                detectable stop{
                        when(whenever)
                            detect((current.perceivedtime = 10),
                                dc:100)
                            then abort;
                    }
            when(knownval(current.perceivedtime < 10) and
                knownval(current.location = toilet) and
                knownval(current.handsWashed = true) and
                knownval(current.toiletFlushed = true) and
                knownval(current.needToilet < 5))
            do{
                moveToChair();
            }
        }

    thoughtframes:

        thoughtframe tf_durationAskedForFood {
            repeat: true;
            priority: 2;

            when(knownval(Campanile_Clock.time >
                    current.perceivedtime) and
                knownval(current.askedFood = true) and
                knownval(current.updateAskedFood = true))

            do {
                conclude((current.waitingForFood =
                    current.waitingForFood + 1), bc:100);
                conclude((current.updateAskedFood = false));
            }
        }

        thoughtframe tf_asTimeGoesBy {
            repeat: true;
            priority: 1;

            when(knownval(Campanile_Clock.time >
                    current.perceivedtime) and
                knownval(current.howHungry < 21 ))

            do {
                conclude((current.perceivedtime =
                    Campanile_Clock.time), bc: 100);
                conclude((current.howHungry =
                    current.howHungry + 3));
                conclude((current.updateAskedFood = true));
                conclude((current.needToilet =
                    current.needToilet +1));
    }
}

thoughtframe tf_timeSinceAskedFood {
    repeat: true;

    when(knownval(current.howHungry > 10 ) and
        knownval(Campanile_Clock.time >
        current.perceivedtime))

    do {
        conclude((current.perceivedtime =
            Campanile_Clock.time), bc: 100);
        conclude((current.howHungry =
            current.howHungry + 3));
        conclude((current.updateAskedFood = true));
        conclude((current.needToilet =
            current.needToilet +1));
        conclude((current.timeSinceAskedFood =
            current.timeSinceAskedFood+1));
    }
}

thoughtframe tf_takeMedicationA {
    repeat: true;
    priority: 3;

    when(knownval(current.perceivedtime < 10) and
        knownval(current.hasMedicationA = true) and
        knownval(current.takeMedicationA = false) and
        knownval(current.location = chair))

    do {
        conclude((current.takeMedicationA = true));
        conclude((current.hasMedicationA = false));
    }
}

thoughtframe tf_fireAlarm {
    repeat: true;
    priority: 99;

    when(knownval(theEnvironment.fire = true) and
        knownval(current.fireDecision = false))

    do {
        conclude((current.evacuate = true),bc:50);
        conclude((current.fireDecision = true));
    }
```

```
        }


    }
    agent robotHelper{
        location: chair;
        attributes:
            public int perceivedtime;
            public int fireAlerted;
            public boolean checkMedicationA;
            public boolean medNotificationA;
            public int timeSinceMedANotification;

        initial_beliefs:
            (current.timeSinceMedANotification = 0);
            (Alex.location = chair);
            (robotHouse.doorBellRang = false);
            (careWorker.location = careCentre);
            (Alex.hasEmptyPlate = false);
            (Alex.askedFood = false);
            (Alex.hasMedicationA = false);
            (Alex.missedMedicationA = 0);
            (Alex.takeMedicationA = false);
            (current.perceivedtime = 1);
            (current.fireAlerted = 0);
            (theEnvironment.fire = false);
            (Alex.evacuate = false);
            (Alex.danger = false);
            (current.medNotificationA = false);
            (Alex.hasTakenMedicationA = false);
        initial_facts:

        activities:

            primitive_activity getFood(){
                max_duration: 2;
            }

            primitive_activity cookFood(){
                max_duration: 1200;
            }

            primitive_activity placeFoodOnTray(){
                max_duration: 20;
            }

            primitive_activity putMedsOnTray(){
                max_duration: 15;
            }
```

```
primitive_activity pickupMeds(){
    max_duration: 100;
}

primitive_activity checkMedication(){
    max_duration: 10;
}

communicate tellAlexFood(){
    max_duration: 1;
    with: Alex;
    about:
        send(Alex.hasFood = Alex.hasFood);
}

communicate remindMedictionA(){
    max_duration: 1;
    with: Alex;
    about:
        send(Alex.takeMedicationA = Alex.takeMedicationA);
}


communicate tellHouseMed(){
    max_duration: 1;
    with: robotHouse;
    about:
        send(Alex.missedMedicationA =
            Alex.missedMedicationA);
}

communicate giveMedicationA(){
    max_duration: 1;
    with: Alex;
    about:
        send(Alex.hasMedicationA = Alex.hasMedicationA);
}

communicate alertFire(){
    max_duration: 1;
    with: Alex;
    about:
        send(Alex.evacuate = Alex.evacuate);
}

communicate TellAlexLocation(){
    max_duration: 1;
    with: careWorker;
    about:
        send(Alex.location = Alex.location);
}

communicate alexInDanger(){
    max_duration: 1;
    with: robotHouse;
    about:
        send(Alex.danger = Alex.danger);
}
primitive_activity openDoor(){
    max_duration: 1;
}
primitive_activity closeFrontDoor(){
    max_duration: 1;
}
primitive_activity wait(){
    max_duration: 1000;
}

primitive_activity platesInDishWasher(){
    max_duration: 50;
}

primitive_activity pickupPlates(){
    max_duration: 5;
}

move moveToChair() {
     location: chair;
}

move moveToMeds() {
     location: medCabinet;
}

move moveToBed() {
     location: bed;
}

move moveToSinkOne() {
     location: sinkOne;
}

move moveToDishWasher() {
     location: dishWasher;
}

move moveToMicroWave() {
```

```
          location: microWave;
     }

     move moveToToilet() {
          location: toilet;
     }

     move moveToSinkTwo() {
          location: sinkTwo;
     }

     move moveToFrontDoor() {
          location: frontDoor;
     }

workframes:


     workframe wf_answerDoor {
         repeat: true;
         priority: 10;

         when(knownval(robotHouse.doorBellRang = true))
         do{
             moveToFrontDoor();
             conclude((robotHouse.doorBellRang = false));
             openDoor();
             conclude((robotHouse.doorOpen = true));
             TellAlexLocation();
         }
     }

     workframe wf_closeDoor {
         repeat: true;
         priority: 10;

         when(knownval(robotHouse.doorBellRang = false) and
             knownval(robotHouse.doorOpen = true))
         do{
             closeFrontDoor();
             conclude((robotHouse.doorOpen = false));
         }
     }

     workframe wf_fireAlarmChair {
         repeat: true;
         priority: 100;

         detectables:
```

```
        detectable AlexSafe{
            when(whenever)
                detect((Alex.location = frontDoor), dc:100)
                then abort;
        }

    when(knownval(theEnvironment.fire = true) and
        knownval(Alex.location = chair))
    do{
        moveToChair();
        conclude((Alex.evacuate = true));
        alertFire();
    }

}

workframe wf_fireAlarmToilet {
    repeat: true;
    priority: 100;

    detectables:
        detectable AlexSafe{
            when(whenever)
                detect((Alex.location = frontDoor), dc:100)
                then abort;
        }

    when(knownval(theEnvironment.fire = true) and
        knownval(Alex.location = toilet))
    do{
        moveToToilet();
        conclude((Alex.evacuate = true));
        alertFire();
    }

}

workframe wf_fireAlarmSink {
    repeat: true;
    priority: 100;

    detectables:
        detectable AlexSafe{
            when(whenever)
                detect((Alex.location = frontDoor), dc:100)
                then abort;
        }

    when(knownval(theEnvironment.fire = true) and
        knownval(Alex.location = sinkTwo))
    do{
        moveToSinkTwo();
        conclude((Alex.evacuate = true));
        alertFire();
    }

}

workframe wf_alexInDanger {
    repeat: true;
    priority: 101;

    when(knownval(theEnvironment.fire = true) and
        knownval(Alex.evacuate = false) and
        knownval(current.fireAlerted > 5) and
        knownval(Alex.danger = false))
    do{
        conclude((Alex.danger = true));
        alexInDanger();
    }

}

workframe wf_cleanPlates {
        repeat: true;
        priority: 2;
        when(knownval(Alex.hasEmptyPlate = true))
        do {
            moveToChair();
            pickupPlates();
            moveToDishWasher();
            conclude((Alex.hasEmptyPlate = false));
            platesInDishWasher();
            moveToChair();
        }
}

workframe wf_medicationA {
    repeat: true;
    priority: 5;
    when(knownval(current.perceivedtime = 2) and
        knownval(Alex.hasMedicationA = false))
    do {
        moveToMeds();
        pickupMeds();
        moveToChair();
        putMedsOnTray();
        conclude((Alex.hasMedicationA = true));
        conclude((Alex.missedMedicationA = 0));
        giveMedicationA();
    }
}




workframe wf_checkMedicationA {
    repeat: true;
    priority: 3;
    detectables:
        detectable takenMedicationA{
            when(whenever)
                detect((Alex.hasMedicationA = false), dc:100)
                then abort;
        }
    when(knownval(current.perceivedtime > 2)and
        knownval(current.perceivedtime < 10) and
        knownval(Alex.hasMedicationA = true) and
        knownval(Alex.missedMedicationA < 2) and
        knownval(current.checkMedicationA = true))
    do {
        checkMedication();
        conclude((current.checkMedicationA = false));
        conclude((Alex.takeMedicationA = true));
        remindMedictionA();
        conclude((Alex.missedMedicationA =
            Alex.missedMedicationA + 1));
    }
}

workframe wf_notifyMedicationANotTaken {
    repeat: true;
    priority: 15;

    when(knownval(current.perceivedtime < 10) and
        knownval(Alex.missedMedicationA > 1) and
        knownval(current.medNotificationA = false))
    do{
        conclude((current.medNotificationA = true));
        tellHouseMed();
    }

}

workframe wf_getFood {
    repeat: true;
    priority: 4;
    when(knownval(Alex.askedFood = true))
    do {
        moveToMicroWave();
        getFood();
        cookFood();
        moveToChair();
        placeFoodOnTray();
        conclude((Alex.askedFood = false));
        conclude((Alex.hasFood = true));
        tellAlexFood();
       }
}

workframe wf_waitForInstruction {
    repeat: true;
    priority: 1;

    detectables:
        detectable platesToClean{
            when(whenever)
                detect((Alex.hasEmptyPlate = true), dc:100)
                then abort;
        }
        detectable feedAlex{
            when(whenever)
                detect((Alex.askedFood = true), dc:100)
                then abort;
        }

    when(knownval(current.perceivedtime < 10) and
        knownval(theEnvironment.fire = false))
    do{
        wait();
    }

}

workframe wf_endOfSimulation {
    repeat: true;
    priority: 999;

    when(knownval(current.perceivedtime < 10) and
        knownval(theEnvironment.fire =  true) and
        knownval(Alex.location = frontDoor))
    do{
        conclude((current.perceivedtime = 10));
    }

}
```

```
    thoughtframes:
        thoughtframe tf_updateTime {
            repeat: true;
            priority: 1;

            when(knownval(Campanile_Clock.time >
                current.perceivedtime))

            do {
                conclude((current.perceivedtime =
                    Campanile_Clock.time), bc: 100);
                conclude((current.checkMedicationA = true));
            }
        }

        thoughtframe tf_MedsUpdateTime {
            repeat: true;
            priority: 2;

            when(knownval(Campanile_Clock.time >
                current.perceivedtime) and
                knownval(current.medNotificationA = true) and
                knownval(Alex.hasTakenMedicationA = false))

            do {
                conclude((current.perceivedtime =
                    Campanile_Clock.time), bc: 100);
                conclude((current.timeSinceMedANotification =
                    current.timeSinceMedANotification + 1));
            }
        }
}

agent robotHouse{
    attributes:
        public int perceivedtime;
        public boolean alarmSounding;
        public boolean calledCareWorker;
        public boolean doorBellRang;
        public boolean doorOpen;
    initial_beliefs:
        (current.doorOpen = false);
        (current.doorBellRang = false);
        (current.calledCareWorker = false);
        (current.perceivedtime = 1);
        (current.alarmSounding = false);
        (theEnvironment.fire = false);
        (Alex.location = chair);
    (Alex.handsWashed = true);
    (Alex.toiletFlushed = true);
    (Alex.danger = false);
    (Alex.missedMedicationA = 0);
initial_facts:
    (current.doorOpen = false);

activities:
    primitive_activity monitorAlex(){
        max_duration: 3000;
    }
    primitive_activity pause(){
        max_duration: 120;
    }
    primitive_activity checkAlex(){
        max_duration: 1;
    }
    primitive_activity fireAlarm(){
        max_duration: 100;
    }
    communicate AlexWashHands(){
        max_duration: 1;
        with: Alex;
        about:
            send(Alex.handsWashed = Alex.handsWashed);
    }

    communicate AlexFlushToilet(){
        max_duration: 1;
        with: Alex;
        about:
            send(Alex.toiletFlushed = Alex.toiletFlushed);
    }

    communicate announceFireAlex(){
        max_duration: 1;
        with: Alex;
        about:
            send(theEnvironment.fire = theEnvironment.fire);
    }

    communicate doorBell(){
        max_duration: 1;
        with: robotHelper;
        about:
            send(current.doorBellRang = current.doorBellRang);
    }

    communicate callCareWorker(){
            max_duration: 1;
            with: careWorker;
            about:
                send(Alex.missedMedicationA =
                    Alex.missedMedicationA);
        }

        communicate announceFireRobot(){
            max_duration: 1;
            with: robotHelper;
            about:
                send(theEnvironment.fire = theEnvironment.fire),
                send(Alex.location = Alex.location);
        }
    workframes:

        workframe wf_monitorAlex {
            repeat: true;
            priority: 1;

            detectables:
                detectable AlexOnToilet{
                    when(whenever)
                        detect((Alex.location = toilet), dc:100)
                        then abort;
                }
                detectable fire{
                    when(whenever)
                        detect((theEnvironment.fire = true),
                            dc:100)
                        then abort;
                }

            when(knownval(current.perceivedtime < 10) and
                knownval(theEnvironment.fire = false))
            do{
                monitorAlex();
            }
        }

        workframe wf_soundFireAlarm {
            repeat: true;
            priority: 1000;
            detectables:
                detectable AlexSafe{
                    when(whenever)
                        detect((Alex.location = frontDoor),
                            dc:100)
                        then abort;
                }
            when(knownval(theEnvironment.fire = true)and
                knownval(Alex.location != frontDoor))
            do{
                conclude((current.alarmSounding = true));
                conclude((current.perceivedtime = 10));
                announceFireAlex();
                announceFireRobot();
                fireAlarm();

            }
        }

        workframe wf_AlexOnToilet{
            repeat: true;
            priority: 3;
            detectables:
                detectable AlexNotOnToilet{
                    when(whenever)
                        detect((Alex.location != toilet), dc:100)
                        then complete;
                }
                detectable fire{
                    when(whenever)
                        detect((theEnvironment.fire = true), dc:100)
                        then abort;
                }
            when(knownval(current.perceivedtime < 10) and
                knownval(Alex.location = toilet))
            do{
                monitorAlex();
                conclude((Alex.handsWashed = false),fc:0);
                conclude((Alex.toiletFlushed = false),fc:0);
            }
        }

        workframe wf_AlexWashHands{
            repeat: true;
            priority: 2;
            detectables:
                detectable AlexNotWashedHands{
                    when(whenever)
                        detect((Alex.handsWashed = true), dc:100)
                        then abort;
                }
                detectable fire{
                    when(whenever)
                        detect((theEnvironment.fire = true), dc:100)
                        then abort;
                }
            when(knownval(current.perceivedtime < 10) and
                knownval(Alex.location != toilet) and
                knownval(Alex.location != sinkTwo) and
                knownval(Alex.handsWashed = false))
            do{
                checkAlex();
                AlexWashHands();
                pause();
            }
        }

        workframe wf_AlexFlushToilet{
            repeat: true;
            priority: 4;
            detectables:
                detectable AlexNotFlushed{
                    when(whenever)
                        detect((Alex.toiletFlushed = true), dc:100)
                        then abort;
                }
                detectable fire{
                    when(whenever)
                        detect((theEnvironment.fire = true), dc:100)
                        then abort;
                }
            when(knownval(current.perceivedtime < 10) and
                knownval(Alex.location != toilet) and
                knownval(Alex.location != sinkTwo) and
                knownval(Alex.toiletFlushed = false))
            do{
                checkAlex();
                AlexFlushToilet();
            }
        }

        workframe wf_callCareWorker{
                repeat: true;
                priority: 10;
                detectables:

                when(knownval(current.perceivedtime < 10) and
                    knownval(Alex.missedMedicationA > 1) and
                    knownval(current.calledCareWorker = false))
                do{
                    callCareWorker();
                    conclude((current.calledCareWorker = true));
                }
        }
```

```
        workframe wf_doorBellRang{
                repeat: true;
                priority: 10;
                detectables:

                when(knownval(current.perceivedtime < 10) and
                    knownval(current.doorBellRang = true))
                do{
                    doorBell();
                    conclude((current.doorBellRang = false));
                }
        }

    thoughtframes:
        thoughtframe tf_updateTime {
            repeat: true;
            priority: 1;

            when(knownval(Campanile_Clock.time >
                current.perceivedtime))

            do {
                conclude((current.perceivedtime =
                    Campanile_Clock.time), bc: 100);
            }
        }

        thoughtframe tf_alarmOff {
            repeat: true;

            when(knownval(current.alarmSounding = true)and
                knownval(Alex.location = frontDoor))
            do{
                conclude((current.alarmSounding = false));
            }
        }

}
agent careWorker memberof person {
    location: careCentre;
    attributes:
        public int perceivedtime;
    initial_beliefs:
        (current.perceivedtime = 1);
        (Campanile_Clock.time = 1);
        (Alex.missedMedicationA = 0);
        (robotHouse.doorBellRang = false);
        (Alex.hasTakenMedicationA = false);
```

```
initial_facts:
activities:
    primitive_activity waitForAnswer(){
        max_duration: 5000;
    }

    primitive_activity wait(){
        max_duration: 5000;
    }

    primitive_activity stuff(){
        max_duration: 7800;
    }

    primitive_activity medicateAlex(){
        max_duration: 500;
    }

    communicate pressDoorBell(){
        max_duration: 1;
        with: robotHouse;
        about:
            send(robotHouse.doorBellRang =
                robotHouse.doorBellRang);
    }
    communicate informRobotMedsTaken(){
        max_duration: 1;
        with: robotHelper;
        about:
            send(Alex.hasTakenMedicationA =
                Alex.hasTakenMedicationA);
    }
    move moveToChair() {
         location: chair;
    }

    move moveToFrontDoor() {
         location: frontDoor;
    }
    move moveToCareCentre() {
         location: careCentre;
    }

workframes:
    workframe wf_careWorkerStuff{
        repeat: true;
        priority: 10;

        when(knownval(current.perceivedtime < 10) and
        knownval(Alex.missedMedicationA < 2) and
        knownval(current.location = careCentre))
    do{
        stuff();
    }
}

workframe wf_respondToCall{
    repeat: true;
    priority: 9;

    when(knownval(current.perceivedtime < 10) and
        knownval(Alex.missedMedicationA > 1) and
        knownval(current.location = careCentre))
    do{
        moveToFrontDoor();
    }
}

workframe gotoChair{
    repeat: true;
    priority: 8;

    when(knownval(current.perceivedtime < 10) and
        knownval(current.location = frontDoor) and
        knownval(robotHouse.doorBellRang = true))
    do{
        moveToChair();
    }
}

workframe waitForAlex {
    repeat: true;
    priority: 7;
    detectables:

        detectable alexAtChair{
            when(whenever)
                detect((Alex.location = chair), dc:100)
                then abort;
        }
    when(knownval(current.perceivedtime < 10) and
        knownval(current.location = chair) and
        knownval(Alex.location != chair))
    do{
        wait();
    }
}
```

```
    workframe administerMeds {
        repeat: true;
        priority: 6;
        when(knownval(current.perceivedtime < 10) and
            knownval(current.location = chair) and
            knownval(Alex.location = chair) and
            knownval(Alex.hasTakenMedicationA = false))
        do{
            medicateAlex();
            conclude((Alex.missedMedicationA = 0));
            conclude((Alex.hasTakenMedicationA = true));
            informRobotMedsTaken();
            moveToCareCentre();
        }
    }

    workframe wf_knockDoor{
        repeat: true;
        priority: 5;
        detectables:

            detectable doorOpen{
                when(whenever)
                detect((robotHouse.doorOpen = true), dc:100)
                    then abort;
            }

        when(knownval(current.perceivedtime < 10) and
            knownval(current.location = frontDoor) and
            knownval(robotHouse.doorBellRang = false))
        do{
            conclude((robotHouse.doorBellRang = true));
            pressDoorBell();
            waitForAnswer();
            conclude((robotHouse.doorBellRang = false));
        }
    }


thoughtframes:

    thoughtframe tf_asTimeGoesBy {
    repeat: true;
    priority: 1;

    when(knownval(Campanile_Clock.time > current.perceivedtime))

    do {
        conclude((current.perceivedtime = Campanile_Clock.time),
                bc: 100);
        }
    }


}


agent theEnvironment{

    attributes:
        boolean fire;
        int perceivedtime;
        int timeSinceFire;
    initial_beliefs:
        (current.fire = false);
        (current.timeSinceFire = 0);
        (current.perceivedtime = 1);

    initial_facts:
        (current.fire = false);
        (current.timeSinceFire = 0);

    workframes:
        workframe wf_fireTwentyOne{
            repeat: false;

            when(knownval(current.perceivedtime = 21) and
                knownval(current.fire = false))
            do{
                conclude((current.fire = true), fc:50);
            }

        }

        workframe wf_fireTen{
            repeat: false;

            when(knownval(current.perceivedtime = 10) and
                knownval(current.fire = false))
            do{
                conclude((current.fire = true), fc:50);
            }

        }

    thoughtframes:
        thoughtframe tf_asTimeGoesBy {
            repeat: true;
            priority: 1;

            when(knownval(Campanile_Clock.time >
                current.perceivedtime))

            do {
                conclude((current.perceivedtime =
                    Campanile_Clock.time), bc: 100);
            }
        }

        thoughtframe tf_asTimeGoesByTwo {
            repeat: true;
            priority: 2;

            when(knownval(Campanile_Clock.time >
                current.perceivedtime) and
                knownval(current.fire = true))

            do {
                conclude((current.perceivedtime =
                    Campanile_Clock.time), bc: 100);
                conclude((current.timeSinceFire =
                    current.timeSinceFire + 1), bc: 100);
            }
        }


}
agent Campanile_Clock  {

    attributes:
        public int time;

    initial_beliefs:
        (current.time = 1);

    initial_facts:
        (current.time = 1);


    activities:

        primitive_activity asTimeGoesBy() {
            max_duration: 3599;
        }


        communicate announceTimeToAlex() {
            max_duration: 1;
            with: Alex;
            about:
                send(current.time = current.time);

            when: end;
        }

        communicate announceTimeToCareWorker() {
            max_duration: 1;
            with: careWorker;
            about:
                send(current.time = current.time);

            when: end;
        }

        communicate announceTimeToHouse() {
            max_duration: 1;
            with: robotHelper;
            about:
                send(current.time = current.time);

            when: end;
        }

        communicate announceTimeToRobot() {
            max_duration: 1;
            with: robotHouse;
            about:
                send(current.time = current.time);

            when: end;
        }

        communicate announceTimeToEnvironment() {
            max_duration: 1;
            with: theEnvironment;
            about:
                send(current.time = current.time);

            when: end;
        }

    workframes:
        workframe wf_asTimeGoesBy {
            repeat: true;

            when(knownval(current.time < 10))
            do {
                asTimeGoesBy();
                conclude((current.time = current.time + 1),
                    bc:100, fc:100);
                announceTimeToEnvironment();
                announceTimeToAlex();
                announceTimeToHouse();
                announceTimeToRobot();
                announceTimeToCareWorker();
            }
        }
}
```

## D.2 The Digital Nurse Case Study

```
areadef hospital extends BaseAreaDef { }
areadef room extends BaseAreaDef { }
areadef bed extends BaseAreaDef {}
area hospitalGeography instanceof World { }
area medRoom instanceof room partof hospitalGeography { }
area staffRoom instanceof room partof hospitalGeography { }
area wardOne instanceof room partof hospitalGeography { }
area wardTwo instanceof room partof hospitalGeography { }
area bedOne instanceof bed partof wardOne { }
area bedTwo instanceof bed partof wardOne { }
area bedThree instanceof bed partof wardOne { }
area bedFour instanceof bed partof wardOne { }
area bedFive instanceof bed partof wardOne { }

path bedOne_to_from_bedTwo {
    area1: bedOne;
    area2: bedTwo;
    distance: 2;
}

path bedTwo_to_from_bedThree {
    area1: bedTwo;
    area2: bedThree;
    distance: 2;
}

path bedThree_to_from_bedFour {
    area1: bedThree;
    area2: bedFour;
    distance: 2;
}
```

```
path bedFour_to_from_bedFive {
    area1: bedFour;
    area2: bedFive;
    distance: 2;
}

path bedOne_to_from_wardTwo {
    area1: bedOne;
    area2: wardTwo;
    distance: 2;
}

path bedOne_to_from_medRoom {
    area1: bedOne;
    area2: medRoom;
    distance: 2;
}

path bedOne_to_from_staffRoom {
    area1: bedOne;
    area2: staffRoom;
    distance: 2;
}
class MyClock  {

    attributes:
        public int time;

    activities:

        primitive_activity asTimeGoesBy() {
            max_duration: 3599;
        }

        communicate announceTimeTo(TimeKeepers t) {
            max_duration: 1;
            with: t;
            about:
                send(current.time = current.time);

            when: end;
        }

    workframes:
        workframe wf_asTimeGoesBy {
            repeat: true;

            variables:
                collectall(TimeKeepers) t;

            when(knownval(current.time < 20) and
                knownval(t.sendTime = true))

            do {
                asTimeGoesBy();
                conclude((current.time = current.time + 1),
                    bc:100, fc:100);
                announceTimeTo(t);
            }
        }
}
object Campanile_Clock instanceof MyClock  {
    initial_beliefs:
        (Patient_one.sendTime = true);
        (Patient_two.sendTime = true);
        (Patient_three.sendTime = true);
        (Patient_four.sendTime = true);
        (Patient_five.sendTime = true);
        (Nurse_one.sendTime = true);
        (Nurse_two.sendTime = true);
        (Digital_Nurse_one.sendTime = true);
        (Digital_Nurse_two.sendTime = true);
        (Digital_Nurse_three.sendTime = true);
        (Doctor_one.sendTime = true);
        (Robot.sendTime = true);
        (current.time = 1);

initial_facts:
        (Patient_one.sendTime = true);
        (Patient_two.sendTime = true);
        (Patient_three.sendTime = true);
        (Patient_four.sendTime = true);
        (Patient_five.sendTime = true);
        (Nurse_one.sendTime = true);
        (Nurse_two.sendTime = true);
        (Digital_Nurse_one.sendTime = true);
        (Digital_Nurse_two.sendTime = true);
        (Digital_Nurse_three.sendTime = true);
        (Robot.sendTime = true);
        (current.time = 1);
}

group Digital_Nurse memberof TimeKeepers{
attributes:

    relations:
    initial_beliefs:
```

```
    (Monitor.breakfastTime = false);
    (Patient_one.breakfastChoiceMade = false);
    (Patient_two.breakfastChoiceMade = false);
    (Patient_three.breakfastChoiceMade = false);
    (Patient_four.breakfastChoiceMade = false);
    (Patient_five.breakfastChoiceMade = false);
    (Nurse_one.break = false);
    (Nurse_two.turnDuty = false);
initial_facts:
activities:
    communicate announceBreakfast(Nurse n) {
        max_duration: 1;
        with: n;
        about:
            send(Monitor.breakfastTime = Monitor.breakfastTime);

        when: end;
    }
    communicate notifyMedication(Patient pat) {
        max_duration: 1;
        with: Robot;
        about:
            send(pat.medication = pat.medication);
        when: end;
    }
    communicate notifyHeartAttackNurse(Patient pat, Nurse n) {
        max_duration: 1;
        with: n;
        about:
            send(pat.heartAttack = pat.heartAttack);
        when: end;
    }
    communicate notifyHeartAttackDoctor(Patient pat) {
        max_duration: 1;
        with: Doctor_one;
        about:
            send(pat.heartAttack = pat.heartAttack);
        when: end;
    }
    communicate notifyHeartAttackRobot(Patient pat) {
        max_duration: 1;
        with: Robot;
        about:
            send(pat.heartAttack = pat.heartAttack);
        when: end;
    }
    communicate sendRobotForBreakfast(Patient pat) {
        max_duration: 1;
        with: Robot;
        about:
            send(pat.breakfastChoice = pat.breakfastChoice);

        when: end;
    }
    communicate sendBreakTime() {
        max_duration: 1;
        with: Nurse_one;
        about:
            send(Nurse_one.break = Nurse_one.break);
        when: end;
    }
    communicate sendTurnDuty() {
        max_duration: 1;
        with: Nurse_two;
        about:
            send(Nurse_two.turnDuty = Nurse_two.turnDuty);
        when: end;
    }

workframes:
    workframe wf_breakfast{
        repeat: false;
        priority: 1;
        variables:
            forone(Nurse) n;
        when(knownval(Campanile_Clock.time = 8)and
            knownval(n hasDN current))

        do {
            conclude((Monitor.breakfastTime = true));
            announceBreakfast(n);
        }
    }
    workframe wf_RobotGetBreakfast{
        repeat: true;
        priority: 1;
        variables:
            forone(Patient) pat;
        when(knownval(pat.breakfastChoice > 0)and
            knownval(pat.breakfastChoiceMade = false))

        do {
            conclude((pat.breakfastChoiceMade = true));
            sendRobotForBreakfast(pat);
        }
    }
    workframe wf_notifyMedication{
        repeat: true;
    priority: 1;
    variables:
        forone(Patient) pat;
    when(knownval(pat.medication > 0))

    do {
        notifyMedication(pat);
        conclude((pat.medication = 0));

    }
}
workframe wf_notifyHeartAttackNurse{
    repeat: false;
    priority: 1;
    variables:
        forone(Nurse) n;
        forone(Patient) pat;
    when(knownval(n hasDN current) and
        knownval(pat.heartAttack = true)
        )

    do {
        notifyHeartAttackNurse(pat, n);
        notifyHeartAttackRobot(pat);
    }
}
workframe wf_notifyHeartAttackDoctor{
    repeat: false;
    priority: 1;
    variables:
        forone(Patient) pat;

    when(knownval(Doctor_one hasDN current) and
        knownval(pat.heartAttack = true)
        )

    do {
        notifyHeartAttackDoctor(pat);
    }
}

workframe wf_breakTime{
    repeat: false;
    priority: 1;
    when(knownval(current.perceivedTime > 9))
    do {
        conclude((Nurse_one.break = true));
        sendBreakTime();
        conclude((Nurse_two.turnDuty = true));
```

```
                sendTurnDuty();
            }
        }


    thoughtframes:
        thoughtframe tf_updateTime{
            repeat: true;
            priority: 1;

            when(knownval(Campanile_Clock.time >
                current.perceivedTime))

            do {
                conclude((current.perceivedTime =
                    Campanile_Clock.time));
            }
        }
}

agent Digital_Nurse_one memberof Digital_Nurse{
    initial_beliefs:
        (Nurse_one hasDN current);

}

agent Digital_Nurse_two memberof Digital_Nurse{
    initial_beliefs:
        (Nurse_two hasDN current);

}

agent Digital_Nurse_three memberof Digital_Nurse{
    initial_beliefs:
        (Doctor_one hasDN current);

}
group Doctors memberof TimeKeepers{
    attributes:
            public boolean performedCPR;
    relations:
        public Digital_Nurse hasDN;

    initial_beliefs:
        (Patient_one.location = bedOne);
        (Patient_two.location = bedTwo);
        (Patient_three.location = bedThree);
        (Patient_four.location = bedFour);
        (Patient_five.location = bedFive);
    (Patient_one.doctorVisit = false);
    (Patient_two.doctorVisit = false);
    (Patient_three.doctorVisit = false);
    (Patient_four.doctorVisit = false);
    (Patient_five.doctorVisit = false);
    (Patient_one.requestMeds = false);
    (Patient_two.requestMeds = false);
    (Patient_three.requestMeds = false);
    (Patient_four.requestMeds = false);
    (Patient_five.requestMeds = false);
    (Patient_one.medication = 3);
    (Patient_two.medication = 0);
    (Patient_three.medication = 1);
    (Patient_four.medication = 4);
    (Patient_five.medication = 6);
    (Patient_one.alive = true);
    (Patient_two.alive = true);
    (Patient_three.alive = true);
    (Patient_four.alive = true);
    (Patient_five.alive = true);
initial_facts:
    (current.performedCPR = false);
activities:
    primitive_activity examinePatient() {
        max_duration: 1000;
    }
    primitive_activity resuscitate() {
        max_duration: 100;
    }
    communicate sendMedication(Patient pat, Digital_Nurse DN) {
        max_duration: 1;
        with: DN;
        about:
            send(pat.medication = pat.medication);

        when: end;
    }
    move moveToBed(bed b) {
             location: b;
        }
workframes:

    workframe wf_visitPatient{
        repeat: true;
        priority: 1;
        variables:
            forone(Digital_Nurse) DN;
            forone(Patient) pat;
            forone(bed) b;
            when(knownval(current hasDN DN) and
                knownval(pat.doctorVisit = false) and
                knownval(pat.location = b))

            do {
                moveToBed(b);
                conclude((pat.doctorVisit = true));
                examinePatient();
                sendMedication(pat, DN);
            }
        }

        workframe wf_resuscitate{
            repeat: true;
            priority: 100;
            variables:
                forone(Patient) pat;
                forone(bed) b;
            detectables:
                detectable resuscitated {
                    when(whenever)
                        detect((pat.heartAttack = false), dc:100)
                        then abort;
                }
            when(knownval(pat.heartAttack = true) and
                knownval(pat.location = b) and
                knownval(pat.alive = true))

            do {
                moveToBed(b);
                resuscitate();
                conclude((current.performedCPR = true));
            }
        }

    thoughtframes:
}

agent Doctor_one memberof Doctors {
    location: staffRoom;
    initial_beliefs:
        (current hasDN Digital_Nurse_three);
}
class Monitors {
    attributes:
        public boolean announceWater;
        public int lowestWaterLevel;
        public boolean breakfastTime;
        public boolean lunchTime;
    public boolean teaTime;
initial_beliefs:
    (current.announceWater = false);
    (current.lowestWaterLevel = 1000);
    (Patient_one.waterLevel = 1000);
    (Patient_two.waterLevel = 1000);
    (Patient_three.waterLevel = 1000);
initial_facts:
    (current.timeWater = false);
activities:

    primitive_activity pause() {
        max_duration: 60;
    }

    primitive_activity wait() {
        max_duration: 400;
    }

    communicate announceDeathNurseOne(Patient pat) {
        max_duration: 1;
        with: Nurse_one;
        about:
            send(pat.alive = pat.alive);
        when: end;
    }
    communicate announceDeathNurseTwo(Patient pat) {
        max_duration: 1;
        with: Nurse_two;
        about:
            send(pat.alive = pat.alive);
        when: end;
    }
    communicate announceDeathDoctor(Patient pat) {
        max_duration: 1;
        with: Doctor_one;
        about:
            send(pat.alive = pat.alive);
        when: end;
    }
    communicate announceDeathRobot(Patient pat) {
        max_duration: 1;
        with: Robot;
        about:
            send(pat.alive = pat.alive);
        when: end;
    }

    communicate announceWater(Patient pat) {
        max_duration: 1;
        with: Robot;
        about:
            send(pat.waterLevel = pat.waterLevel);

        when: end;
    }
    communicate heartAttackDN_one(Patient pat) {
        max_duration: 1;
        with: Digital_Nurse_one;
        about:
            send(pat.heartAttack = pat.heartAttack);

        when: end;
    }
    communicate heartAttackDN_two(Patient pat) {
        max_duration: 1;
        with: Digital_Nurse_two;
        about:
            send(pat.heartAttack = pat.heartAttack);

        when: end;
    }
    communicate heartAttackDN_three(Patient pat) {
        max_duration: 1;
        with: Digital_Nurse_three;
        about:
            send(pat.heartAttack = pat.heartAttack);

        when: end;
    }


workframes:
    workframe wf_lowWater {
        repeat: true;
        priority: 1;
        variables:
            forone(Patient) pat;
        detectables:
            detectable pat_Water{
                when(whenever)
                    detect((pat.waterLevel > 200), dc:100)
                    then abort;
            }
        detectable pat_Water2{
            when(whenever)
                detect((pat.waterLevel < 1001), dc:100)
                then continue;
        }

    when(knownval(pat.waterLevel < 201))
        do {
        announceWater(pat);
        wait();
    }
}




workframe wf_HeartAttackAlarm {
    repeat: true;
    priority: 100;
    variables:
        forone(Patient) pat;

    when(knownval(pat.heartAttack = true) and
        knownval(pat.alive = true) and
        knownval(pat.timeSinceHeartAttack < 5))
    do {
        conclude((pat.heartAttack = true), bc:100);
        heartAttackDN_one(pat);
        heartAttackDN_two(pat);
        heartAttackDN_three(pat);
        pause();
        conclude((pat.timeSinceHeartAttack =
            pat.timeSinceHeartAttack +1),fc:100);
    }
}

workframe wf_death {
    repeat: true;
    priority: 101;
    variables:
        forone(Patient) pat;

    when(knownval(pat.heartAttack = true) and
        knownval(pat.timeSinceHeartAttack > 4))
    do {
        conclude((pat.alive = false),bc: 100,fc:100);
        conclude((pat.timeSinceHeartAttack = 0),fc:100);
        announceDeathNurseOne(pat);
        announceDeathNurseTwo(pat);
        announceDeathDoctor(pat);
        announceDeathRobot(pat);
    }
}
```

```
        workframe wf_resuscitated {
            repeat: true;
            priority: 102;
            variables:
                forone(Patient) pat;
                forone(Nurse) n;

            when(knownval(pat.heartAttack = true) and
                knownval(pat.timeSinceHeartAttack < 3) and
                knownval(n.turnDuty = true) and
                knownval(n.performedCPR = true) and
                knownval(Robot.performedCPR = true) and
                knownval(Doctor_one.performedCPR = true))
             do {
                conclude((pat.heartAttack = false),fc:100);
            }
        }


}

object Monitor instanceof Monitors{

}


group Nurse memberof TimeKeepers{
    attributes:
        public boolean turnDuty;
        public boolean foodDuty;
        public boolean ReadyToTurnPatientOne;
        public boolean ReadyToTurnPatientTwo;
        public boolean ReadyToTurnPatient;
        public boolean turningPatient;
        public boolean performedCPR;
        public boolean break;
    relations:
        public Digital_Nurse hasDN;

    initial_beliefs:
        (current.break = false);
        (current.turningPatient = false);
        (Patient_one.readyToBeTurned = false);
        (Patient_two.readyToBeTurned = false);
        (Patient_three.readyToBeTurned = false);
        (Patient_four.readyToBeTurned = false);
        (Patient_five.readyToBeTurned = false);
        (Patient_one.location = bedOne);
        (Patient_two.location = bedTwo);
        (Patient_three.location = bedThree);
        (Patient_four.location = bedFour);
        (Patient_five.location = bedFive);
        (Patient_one.needTurning = true);
        (Patient_two.needTurning = false);
        (Patient_three.needTurning = true);
        (Patient_four.needTurning = false);
        (Patient_five.needTurning = true);
        (Patient_one.turned = true);
        (Patient_two.turned = true);
        (Patient_three.turned = true);
        (Patient_four.turned = true);
        (Patient_five.turned = true);
        (Patient_one.nilByMouth = true);
        (Patient_two.nilByMouth = false);
        (Patient_three.nilByMouth = false);
        (Patient_four.nilByMouth = false);
        (Patient_five.nilByMouth = false);
        (Patient_one.breakfastRequest = false);
        (Patient_two.breakfastRequest = false);
        (Patient_three.breakfastRequest = false);
        (Patient_four.breakfastRequest = false);
        (Patient_five.breakfastRequest = false);
        (Patient_one.breakfastChoiceMade = false);
        (Patient_two.breakfastChoiceMade = false);
        (Patient_three.breakfastChoiceMade = false);
        (Patient_four.breakfastChoiceMade = false);
        (Patient_five.breakfastChoiceMade = false);
        (Patient_one.alive = true);
        (Patient_two.alive = true);
        (Patient_three.alive = true);
        (Patient_four.alive = true);
        (Patient_five.alive = true);
    initial_facts:
        (current.performedCPR = false);
        (current.break = false);
    activities:
        move moveToBed(bed b) {
            location: b;
        }
        move moveToStaffRoom() {
            location: staffRoom;
        }
        primitive_activity waitToTurn() {
            max_duration: 500;
        }
        primitive_activity haveBreak() {
            max_duration: 1800;
```

```
}
primitive_activity resuscitate() {
    max_duration: 100;
}
primitive_activity turnPatient() {
    max_duration: 240;
}
primitive_activity wait() {
    max_duration: 5;
}
communicate requestBreakfastChoice(Patient pat) {
    max_duration: 1;
    with: pat;
    about:
        send(pat.breakfastRequest = pat.breakfastRequest);
    when: end;
}
communicate requestLunchChoice(Patient pat) {
    max_duration: 1;
    with: pat;
    about:
        send(pat.breakfastRequest = pat.breakfastRequest);
    when: end;
}
communicate requestTeaChoice(Patient pat) {
    max_duration: 1;
    with: pat;
    about:
        send(pat.breakfastRequest = pat.breakfastRequest);

    when: end;
}
communicate sendBreakfastChoice(Patient pat,
        Digital_Nurse DN) {
    max_duration: 1;
    with: DN;
    about:
        send(pat.breakfastChoice = pat.breakfastChoice);

    when: end;
}
communicate patientToTurn(Patient pat) {
    max_duration: 1;
    with: Robot;
    about:
        send(pat.readyToBeTurned = pat.readyToBeTurned);
    when: end;
}
communicate sendTurnDutytwo() {
```

```
            max_duration: 1;
            with: Nurse_two;
            about:
                send(Nurse_two.turnDuty = Nurse_two.turnDuty);
            when: end;
    }
workframes:

    workframe wf_turnOne{
        repeat: true;
        priority: 1;
        variables:
            forone(Patient) pat;
            forone(bed) b;

        detectables:
            detectable waitForRobot {
                when(whenever)
                    detect((pat.location = Robot.location),
                        dc:100)
                    then abort;
            }

        when(knownval(current.perceivedTime = 8) and
                    knownval(pat.turned = false) and
                    knownval(pat.readyToBeTurned = false) and
                    knownval(pat.needTurning = true) and
                    knownval(current.turnDuty = true) and
                    knownval(current.turningPatient = false) and
                    knownval(pat.location = b))

        do {
            moveToBed(b);
            conclude((pat.readyToBeTurned = true));
            conclude((current.turningPatient = true));
            patientToTurn(pat);
            waitToTurn();
            conclude((pat.readyToBeTurned = false));
            conclude((current.turningPatient = false));
        }
    }

    workframe wf_break{
        repeat: true;
        priority: 20;

        when(knownval(current.break = true) and
            knownval(current.perceivedTime < 20))
        do {
```

```
        moveToStaffRoom();
        haveBreak();
        conclude((current.break = false));
        conclude((Nurse_two.turnDuty = false));
        sendTurnDutytwo();
    }
}

workframe wf_turnTwo{
    repeat: true;
    priority: 1;
    variables:
        forone(Patient) pat;

    when(knownval(pat.readyToBeTurned = true) and
        knownval(current.turnDuty = true))

    do {
        turnPatient();
        conclude((pat.turned = true));
        conclude((pat.readyToBeTurned = false));
        conclude((current.turningPatient = false));
        conclude((pat.timeSinceTurned = 0));
    }
}

workframe wf_breakfast{
    repeat: true;
    priority: 2;
    variables:
        forone(Patient) pat;

    when(knownval(Monitor.breakfastTime = true) and
        knownval(pat.nilByMouth = false) and
        knownval(pat.breakfastRequest = false) and
        knownval(current.turnDuty = true))

    do {
        conclude((pat.breakfastRequest = true));
        requestBreakfastChoice(pat);
        wait();
    }
}
workframe wf_sendForBreakfast{
    repeat: true;
    priority: 3;
    variables:
        forone(Digital_Nurse) DN;
        forone(Patient) pat;
        when(knownval(current hasDN DN) and
            knownval(pat.breakfastChoice > 0) and
            knownval(pat.breakfastChoiceMade = false) and
            knownval(current.turnDuty = true))

        do {
                conclude((pat.breakfastChoiceMade = true));
                sendBreakfastChoice(pat, DN);
        }
    }
    workframe wf_resuscitate{
        repeat: true;
        priority: 100;
        variables:
            forone(Patient) pat;
            forone(bed) b;
        detectables:
            detectable resuscitated {
                when(whenever)
                    detect((pat.heartAttack = false), dc:100)
                    then abort;
            }
        when(knownval(pat.heartAttack = true) and
            knownval(pat.location = b) and
            knownval(pat.alive = true))

        do {
            moveToBed(b);
            resuscitate();
            conclude((current.performedCPR = true));
        }
    }

thoughtframes:
    thoughtframe tf_updateTime{
        repeat: true;
        priority: 1;

        when(knownval(Campanile_Clock.time >
            current.perceivedTime))

        do {
            conclude((current.perceivedTime =
                Campanile_Clock.time));
        }
    }

    thoughtframe tf_updateTimeTurn{
            repeat: true;
            priority: 2;

            when(knownval(Campanile_Clock.time >
                current.perceivedTime)
                and knownval(current.perceivedTime > 7))

            do {
                conclude((current.perceivedTime =
                    Campanile_Clock.time));
                conclude((Patient_one.timeSinceTurned =
                    Patient_one.timeSinceTurned + 1));
                conclude((Patient_two.timeSinceTurned =
                    Patient_two.timeSinceTurned + 1));
                conclude((Patient_three.timeSinceTurned =
                    Patient_three.timeSinceTurned + 1));
                conclude((Patient_four.timeSinceTurned =
                    Patient_four.timeSinceTurned + 1));
                conclude((Patient_five.timeSinceTurned =
                    Patient_five.timeSinceTurned + 1));
            }
        }

        thoughtframe tf_turnPatients{
            repeat: false;
            priority: 1;
            variables:
                collectall(Patient) pat;

            when(knownval(current.perceivedTime > 7) and
                        knownval(pat.turned = true))

            do {
                conclude((pat.turned = false));
            }
        }
}
agent Nurse_one memberof Nurse{
location: staffRoom;
initial_beliefs:
        (current.turnDuty = true);
        (current hasDN Digital_Nurse_one);
        (current.turnDuty = true);
initial_facts:
        (current.turnDuty = true);
        (current hasDN Digital_Nurse_one);
        (current.turnDuty = true);

}
```

```
agent Nurse_two memberof Nurse{
    location: staffRoom;
    initial_beliefs:
        (current.turnDuty = false);
        (current hasDN Digital_Nurse_two);
        (current.turnDuty = false);
    initial_facts:
        (current.turnDuty = false);
        (current hasDN Digital_Nurse_two);
        (current.turnDuty = false);
    }

    group Patient memberof TimeKeepers{
    attributes:
        public boolean heartAttackRisk;
        public boolean heartAttack;
        public boolean needTurning;
        public boolean doctorVisit;
        public boolean requestMeds;
        public int medication;
        public boolean turned;
        public boolean nilByMouth;
        public int waterLevel;
        public int drink;
        public int thirst;
        public int timeSinceHeartAttack;
        public int timeSinceTurned;
        public boolean breakfastRequest;
        public int breakfastChoice;
        public boolean breakfastChoiceMade;
        public boolean readyToBeTurned;
        public boolean alive;
        public boolean timeWater;

    initial_beliefs:
        (current.heartAttack = false);
        (current.alive = true);
        (current.thirst = 0);
        (current.waterLevel = 1000);
        (Nurse_one.foodDuty = true);
        (Nurse_two.foodDuty = false);

    initial_facts:
        (current.heartAttack = false);
        (current.turned = false);
        (current.waterLevel = 1000);
        (current.timeSinceHeartAttack = 0);
        (current.alive = true);
        (current.timeSinceTurned = 0);
    activities:
        primitive_activity countWater() {
            max_duration: 3600;
        }
        primitive_activity drinkWater(){
            max_duration: 50;
        }
        communicate stateBreakfastChoice(Nurse nur) {
            max_duration: 1;
            with: nur;
            about:
                send(current.breakfastChoice =
                    current.breakfastChoice);
            when: end;
        }


    workframes:
        workframe wf_drinkWater {
            repeat: true;
            priority: 1;

            when(knownval(current.waterLevel > 200) and
                knownval(current.thirst = 1) and
                knownval(current.heartAttack = false))

            do {
                conclude((current.thirst = 0));
                conclude((current.waterLevel =
                    current.waterLevel - current.drink));
                conclude((Monitor.announceWater = false),
                    bc:0, fc:100);
            }
        }
        workframe wf_requestBreakfast {
            repeat: false;
            priority: 1;
            variables:
                forone(Nurse) nur;
            when(knownval(current.breakfastRequest = true) and
            knownval(current.heartAttack = false) and
            knownval(nur.foodDuty = true))

            do {
                stateBreakfastChoice(nur);
            }
        }
        workframe wf_haveHeartAttack {
            repeat: false;
            priority: 100;
            when(knownval(current.heartAttack = true))
            do {
                conclude((current.heartAttack = true),
                    bc:100, fc:100);
            }
        }

        workframe wf_lowWaterTime {
            repeat: true;
            priority: 2;

            detectables:
                detectable pat_Water{
                    when(whenever)
                        detect((current.waterLevel > 200), dc:100)
                        then abort;
                }
            when(knownval(current.waterLevel < 201) and
                knownval(current.timeWater = false))
                do {
                    countWater();
                    conclude((current.timeWater = true));
            }
        }

    thoughtframes:
        thoughtframe tf_updateTimeHeartAttack {
            repeat: false;
            priority: 2;

            when(knownval(Campanile_Clock.time = 10) and
                knownval(current.heartAttackRisk = true) and
                knownval(current.heartAttack = false))

            do {
                conclude((current.perceivedTime =
                    Campanile_Clock.time));
                conclude((current.heartAttack = true),
                    bc:50, fc:50);
                conclude((current.thirst = 1));
            }
        }
        thoughtframe tf_updateTimeHeartAttackTwo {
            repeat: false;
            priority: 2;

            when(knownval(Campanile_Clock.time = 12) and
                knownval(current.heartAttackRisk = true) and
                knownval(current.heartAttack = false))

            do {
                conclude((current.perceivedTime =
                    Campanile_Clock.time));
                conclude((current.heartAttack = true));
                conclude((current.thirst = 1));
            }
        }
        thoughtframe tf_updateTimeHeartAttackThree {
            repeat: false;
            priority: 2;

            when(knownval(Campanile_Clock.time = 14) and
                knownval(current.heartAttackRisk = true) and
                knownval(current.heartAttack = false))

            do {
                conclude((current.perceivedTime =
                    Campanile_Clock.time));
                conclude((current.heartAttack = true));
                conclude((current.thirst = 1));
            }
        }
        thoughtframe tf_updateTime{
            repeat: true;
            priority: 1;

            when(knownval(Campanile_Clock.time >
                current.perceivedTime))

            do {
                conclude((current.perceivedTime =
                    Campanile_Clock.time));
                conclude((current.thirst = 1));
            }
        }

}

agent Patient_one memberof Patient{
    location: bedOne;
    initial_beliefs:
        (current.heartAttackRisk = true);
        (current.needTurning = true);
        (current.drink = 100);
        (current.nilByMouth = true);
    initial_facts:
        (current.heartAttackRisk = true);
        (current.needTurning = true);
        (current.nilByMouth = true);
}

agent Patient_two memberof Patient{
    location: bedTwo;
    initial_beliefs:
        (current.heartAttackRisk = false);
        (current.needTurning = false);
        (current.drink = 200);
        (current.nilByMouth = false);
        (current.breakfastChoice = 3);
    initial_facts:
        (current.heartAttackRisk = false);
        (current.needTurning = false);
        (current.nilByMouth = false);
}


agent Patient_three memberof Patient{
    location: bedThree;
    initial_beliefs:
        (current.heartAttackRisk = false);
        (current.needTurning = true);
        (current.drink = 200);
        (current.nilByMouth = false);
        (current.breakfastChoice = 1);
    initial_facts:
        (current.heartAttackRisk = false);
        (current.needTurning = true);
        (current.nilByMouth = false);
}

agent Patient_four memberof Patient{
    location: bedFour;
    initial_beliefs:
        (current.heartAttackRisk = false);
        (current.needTurning = false);
        (current.drink = 50);
        (current.nilByMouth = false);
        (current.breakfastChoice = 3);
    initial_facts:
        (current.heartAttackRisk = false);
        (current.needTurning = false);
        (current.nilByMouth = false);
}

agent Patient_five memberof Patient{
    location: bedFive;
    initial_beliefs:
        (current.heartAttackRisk = false);
        (current.needTurning = true);
        (current.drink = 150);
        (current.nilByMouth = false);
        (current.breakfastChoice = 1);
    initial_facts:
        (current.heartAttackRisk = false);
        (current.needTurning = true);
        (current.nilByMouth = false);
}
agent Robot memberof TimeKeepers {
    location: medRoom;
    attributes:
        public boolean performedCPR;

    initial_beliefs:
        (Patient_one.readyToBeTurned = false);
        (Patient_two.readyToBeTurned = false);
        (Patient_three.readyToBeTurned = false);
        (Patient_four.readyToBeTurned = false);
        (Patient_five.readyToBeTurned = false);
        (Patient_one.location = bedOne);
        (Patient_two.location = bedTwo);
        (Patient_three.location = bedThree);
        (Patient_four.location = bedFour);
        (Patient_five.location = bedFive);
        (Patient_one.waterLevel = 1000);
        (Patient_two.waterLevel = 1000);
        (Patient_three.waterLevel = 1000);
        (Patient_four.waterLevel = 1000);
        (Patient_five.waterLevel = 1000);
        (Patient_one.breakfastChoiceMade = true);
        (Patient_two.breakfastChoiceMade = true);
        (Patient_three.breakfastChoiceMade = true);
        (Patient_four.breakfastChoiceMade = true);
        (Patient_five.breakfastChoiceMade = true);
        (Patient_one.medication = 0);
        (Patient_two.medication = 0);
        (Patient_three.medication = 0);
        (Patient_four.medication = 0);
        (Patient_five.medication = 0);
        (Patient_one.alive = true);
        (Patient_two.alive = true);
        (Patient_three.alive = true);
        (Patient_four.alive = true);
        (Patient_five.alive = true);
```

```
initial_facts:
    (current.performedCPR = false);
activities:
    move moveToBed(bed b) {
        location: b;
    }
    move moveToStaffRoom() {
        location: staffRoom;
    }
    move moveToMedRoom() {
        location: medRoom;
    }
    primitive_activity getMedication(){
        max_duration:400;
    }
    primitive_activity refillWater(){
        max_duration:300;
    }
    primitive_activity fetchBreakfast(){
        max_duration:600;
    }
    primitive_activity turnPatient() {
        max_duration: 240;
    }
    primitive_activity resuscitate() {
        max_duration: 100;
    }
workframes:

    workframe wf_turnPatient {
        repeat: true;
        priority: 1;
        variables:
                forone(Patient) pat;
                forone(bed) b;
        when(knownval(pat.readyToBeTurned = true) and
        knownval(pat.location = b))
        do{
            moveToBed(b);
            turnPatient();
            conclude((pat.readyToBeTurned = false));
        }
    }

    workframe wf_refillWater {
        repeat: true;
        variables:
                foreach(Patient) pat;
                forone(bed) b;
        when(knownval(pat.waterLevel < 201) and
            knownval(pat.location = b))
        do{
            moveToStaffRoom();
            refillWater();
            moveToBed(b);
            conclude((pat.waterLevel = 1000));
        }

    }
    workframe wf_fetchBreakfast {
        repeat: true;
        variables:
                forone(Patient) pat;
                forone(bed) b;
        when(knownval(pat.breakfastChoice > 0) and
            knownval(pat.breakfastChoiceMade = true) and
            knownval(pat.location = b))
        do{
            moveToStaffRoom();
            conclude((pat.breakfastChoiceMade = false));
            fetchBreakfast();
            moveToBed(b);
        }

    }

    workframe wf_medication {
        repeat: true;
        variables:
                forone(Patient) pat;
                forone(bed) b;
        when(knownval(pat.medication > 0) and
            knownval(pat.location = b))
        do{
            moveToMedRoom();
            getMedication();
            moveToBed(b);
            conclude((pat.medication = 0));
        }

    }

    workframe wf_resuscitate{
        repeat: true;
        priority: 100;
        variables:
            forone(Patient) pat;
            forone(bed) b;
```

```
            detectables:
                detectable resuscitated {
                    when(whenever)
                        detect((pat.heartAttack = false), dc:100)
                        then abort;
                }
            when(knownval(pat.heartAttack = true) and
                knownval(pat.location = b) and
                knownval(pat.alive = true))

            do {
                moveToBed(b);
                resuscitate();
                conclude((current.performedCPR = true));
            }
        }


    thoughtframes:


}
group TimeKeepers {
    attributes:
        public int perceivedTime;
        public boolean sendTime;
    initial_beliefs:
        (current.perceivedTime = 1);
        (Campanile_Clock.time = 1);
        (current.sendTime = true);
    initial_facts:
        (current.perceivedTime = 1);
        (Campanile_Clock.time = 1);
        (current.sendTime = true);
}
```

# Appendix E

# Java Translation of Brahms Operational Semantics

The two following classes demonstrate the implementation of the Brahms semantics in Java. These classes were shown to researchers at NASA to help them understand how the semantics of Brahms works. These classes uses methods already declared in this appendix as it recycles the Java code used for storing the data of the models for the *PROMELA* translation.

```
/**
*Author: Richard Stocker
*Copyright: University of Liverpool
*Date: Dec 2011
**/

/*
This is the main class which forms the multi-agent system.  This
class stores all the agents, objects, groups etc.

This class relates to the System's Tuple in the semantics i.e.
<Agents, currect agent, Beliefs, Facts, Time>

Agents in the tuple refers to all agents and objects (for simplicity).
Current agent under consideration isn't included as this only happens
during run time.  These data structures only include what comes from
the Brahms code, not any variables which represent "under the hood"
operations such as time.
*/

import java.util.*;
class MultiAgentSystem
{
    // All the data about agents/objects
    Set<agent> agents = new HashSet<agent>();
    Set<object> objects = new HashSet<object>();
    Set<b_class> classes = new HashSet<b_class>();
    Set<group> groups = new HashSet<group>();
```

```
Set<relation> relations = new HashSet<relation>();
Set<activity> activities = new HashSet<activity>();
Set<attribute> attributes = new HashSet<attribute>();
Set<attribute> allAttributes = new HashSet<attribute>();
Set<relation> allRelations = new HashSet<relation>();
Set<belief> beliefs = new HashSet<belief>();
Set<fact> facts = new HashSet<fact>();
Set<guard> guards = new HashSet<guard>();
Set<conclude> concludes = new HashSet<conclude>();
Set<workframe> workframes = new HashSet<workframe>();
Set<thoughtframe> thoughtframes = new HashSet<thoughtframe>();


/***************
*Promela fields*
****************/

String name; // Used to name the agent/object under consideration
// Stores all the identification numbers of agents, objects and
// locations
String identificationNumbers[];

int numberOfAgentsObjects; // Used for array sizes
int numberOfEverything; // Used for array sizes
 // Work around until I make code to count max depth of a workframe
 or thoughtframe

int maxDepth;

Set<locations> locs = new HashSet<locations>();
Set<areaDefs> areaDefinitions = new HashSet<areaDefs>();
Set<path> paths = new HashSet<path>();
int[][] adjacencyMatrix; // Used for calculating paths
int shortestDuration = -1; //Holds the shortest duration


int globalClock = 0; // The global clock


public MultiAgentSystem()
{ }

public MultiAgentSystem(Set new_agents, Set new_objects,
    Set new_classes, Set new_groups, Set new_locs,
    Set new_areaDefinitions, Set new_paths, Set new_facts)
{
    agents = new_agents;
    objects = new_objects;
    classes = new_classes;
```

```
        groups = new_groups;
        locs = new_locs;
        areaDefinitions = new_areaDefinitions;
        paths = new_paths;
        facts = new_facts;
        numberOfAgentsObjects = agents.size() +
            objects.size()+1;
        numberOfEverything = locs.size() +
            numberOfAgentsObjects+1;
        adjacencyMatrix = new int[locs.size()][locs.size()];
        identificationNumbers = new String[numberOfEverything];
        identificationNumbers[0] = "Environment";
        if(locs.size() > 0){
            buildAdjacencyMatrix();
            Dijkstra AdjacencyMatrix =
                new Dijkstra(locs, adjacencyMatrix);
            adjacencyMatrix = AdjacencyMatrix.findShortestPaths();
        }
        initAgents();
        Sch_Star();
    }
    //Create the initial Adjacency matrix
    public void buildAdjacencyMatrix(){
        for(int i = 0; i < locs.size();i++){
            for(int j = 0; j < locs.size();j++){
                adjacencyMatrix[i][j] = 99999;
            }

        }

        int areaID1 = -1;
        int areaID2 = -1;

        //  Generate the adjacency matrix for the geography
        for (Iterator<path> pathit = paths.iterator();
                pathit.hasNext(); ){
            path pt = pathit.next();
            for (Iterator<locations> Locit = locs.iterator();
                    Locit.hasNext(); ){
                locations l = Locit.next();
                if(l.getName().equals(pt.getArea1())){
                    areaID1 = l.getID();
                }
                if(l.getName().equals(pt.getArea2())){
                    areaID2 = l.getID();
                }

            }
            adjacencyMatrix[areaID1][areaID2] = pt.getDist();
        adjacencyMatrix[areaID2][areaID1] = pt.getDist();

    }
}

/*Main method calls this method to distribute workframes,
beliefs etc. down the hierarchy.  So an agent will gain
all the workframes, thoughtframes beliefs etc. from groups
it is a member of*/
public void initAgents(){
    // Find which groups are members of which groups
    System.out.println("Initialising Agents!");
    for(Iterator<group> groupit = groups.iterator();
            groupit.hasNext();){
        group g = groupit.next();
        g.inheritFromMemberOf(groups);
        //Get all facts from every group
        facts.addAll(g.getFacts());
    }
    String indent = "    ";
    for(Iterator<agent> AgIterator = agents.iterator();
            AgIterator.hasNext();){
        agent a = AgIterator.next();
        a.inheritFromMemberOf(groups);
        //Get all facts from every agent
        facts.addAll(a.getFacts());
        a.setIndent(indent);
        indent = indent.concat("    ");
    }
    for(Iterator<object> obIterator = objects.iterator();
            obIterator.hasNext();){
        object o = obIterator.next();
    }

}
// Method holds all Sch_* rules so Sch_run, Sch_rcvd
// and Sch_Term
public void Sch_Star(){
    //System.out.println("Sch_Star rule activated");
    Sch_Run();
    if(shortestDuration != -1){
        Sch_rcvd();//Move clock forward
    }
    else{ /*Sch_term: Quit program*/
        Sch_term();
    }

}
```

```
    public void Sch_Run(){
        /*Sch_WF:  Loop through all agents/objects, tell
        them to start processing and to return the duration
        of the next activity*/
        shortestDuration = -1;//Reset shortest duration
        String indent = "";
        for(Iterator<agent> AgIterator = agents.iterator();
                AgIterator.hasNext();){
            agent a = AgIterator.next();
            // Tell all agents to move to rules of Set_TF
            a.Set_TF(globalClock);
            // Tell all agents to move to rules of Set_*
            // e.g Set_Act or Set_Idle
            int duration = a.Set_WF(globalClock, agents,
                adjacencyMatrix, locs, facts);
            // If a new shortest is found
            if(((duration < shortestDuration) ||
                    shortestDuration == -1) && duration > -1){
                shortestDuration = duration;
            }
            indent = indent.concat("    ");
        }
        for(Iterator<object> obIterator = objects.iterator();
                obIterator.hasNext();){
            object o = obIterator.next();
            // If a new shortest is found
            if(duration > shortestDuration)
                shortestDuration = duration;*/
        }

    }

    public void Sch_rcvd(){
        int tempClock = globalClock;
        globalClock = globalClock + shortestDuration;
        Sch_Star();//Continue cycle

    }

    public void Sch_term(){
        globalClock = -1;
    }


}

/**
*Author: Richard Stocker
*Copyright: University of Liverpool
```

```
*Date: Dec 2011
**/

/* Holds all data about the agents.  Code is almost identical to
objects except during translation object will react on facts and
not beliefs.  In the semantics: Agent's tuple = <agent,
Thoughtframe, Workframe, stage, Befliefs, Facts, Time, Thoughtframes,
Workframes>

Thoughtframe(current thoughtframe), Workframe(current workframe),
Stage(which rules to consider) and Time are not covered in
these data structures because they are purely run time only.
*/

import java.util.*;
class agent
{
    String indent; // used for spacing out the agents output

    String name; // name of the agent
    String display; // Agents display name
    String cost; // cost of agent
    String timeUnit; //
    String location;  // current location of the agent

    // Which group the agent is a member of
    Set<String> memberOf;
    // All relations the agent has.
    Set<relation> relations = new HashSet<relation>();
    // All activites agent has
    Set<activity> activities = new HashSet<activity>();
    // Attributes
    Set<attribute> attributes = new HashSet<attribute>();
    // beliefs
    Set<belief> beliefs = new HashSet<belief>();
    Set<fact> facts = new HashSet<fact>(); // facts
    // workframes
    Set<workframe> workframes = new HashSet<workframe>();
    // thoughtframes
    Set<thoughtframe> thoughtframes = new HashSet<thoughtframe>();

    int[][] adjacencyMatrix;
    Set<locations> locs;

    int ID; // An ID number assigned to the agent

    Set<agent> agents = new HashSet<agent>();
    Set<object> objects = new HashSet<object>();
```

```
Set<b_class> classes = new HashSet<b_class>();
Set<group> groups = new HashSet<group>();

// Number of all objects and agents
int numberOfAgentsObjects;
// Number of agents, objects and locations
int numberOfEverything;
// Holds all the identification numbers for agents.
String identificationNumbers[];
// Detectables to be checked
Set<detectable> agentsDetectables = new HashSet<detectable>();

//  Details of all the locations.
Set<locations> locs = new HashSet<locations>();
 Set<areaDefs> areaDefs = new HashSet<areaDefs>();
 Set<path> paths = new HashSet<path>();

int clock; // agents clock
int globalClock; // as on the tin
int timeRemaining; //Time left on current activity
// The current thoughtframe
thoughtframe currentThoughtframe;
workframe currentWorkframe;
Set<thoughtframe> activeThoughtframes =
    new HashSet<thoughtframe>();
Set<workframe> activeWorkframes = new
    HashSet<workframe>();
double maxPri = 0;

public String initialisePromela = "";
public String toPromela = "";

public agent()
{ }

public agent(
String new_name,
int new_ID,
Set new_memberOf,
String new_display,
String new_cost,
String new_timeUnit,
String new_location,
Set new_relations,
Set new_activities,
Set new_attributes,
Set new_beliefs,
Set new_facts,
Set new_workframes,
Set new_thoughtframes){

    name = new_name;
    ID = new_ID;
    memberOf = new_memberOf;
    display = new_display;
    cost = new_cost;
    timeUnit = new_timeUnit;
    location = new_location;
    relations = new_relations;
    activities = new_activities;
    attributes = new_attributes;
    beliefs = new_beliefs;
    facts = new_facts;
    workframes = new_workframes;
    thoughtframes = new_thoughtframes;

    if(location != null){
        beliefs.add(new belief("current", "location",
            "=", location));
    }

}
// Find which groups the agent is a member of
public void inheritFromMemberOf(Set groups){
    for(Iterator<group> groupit = groups.iterator();
            groupit.hasNext();){
        group g = groupit.next();
        if(memberOf.contains(g.getName())){
            relations.addAll(g.getRelations());
            activities.addAll(g.getActivities());
            attributes.addAll(g.getAttributes());
            beliefs.addAll(g.getBeliefs());
            workframes.addAll(g.getWorkframes());
            thoughtframes.addAll(g.getThoughtframes());
        }
    }
}

public void Set_TF(int gc){
    globalClock = gc;
    /*Check thoughtframes*/
    activeThoughtframes = findActiveThoughtframes();

    if(!activeThoughtframes.isEmpty()){
        Tf_Star();
    }
}
```

```
 /*All rules denoted Set_* */
public int Set_WF(int newGlobalClock,
        Set<agent> new_agents, int[][] new_adjacencyMatrix,
        Set<locations> new_locs, Set<fact> newFacts){
    facts = newFacts;
    agents = new_agents;
    adjacencyMatrix = new_adjacencyMatrix;
    locs = new_locs;
    timeRemaining = -1; // reset time remaining on activity
    globalClock = newGlobalClock;

    /*firstly needs to decide whether or not a thoughtframe
    or workframe is active to chose next rule*/

    /*Check workframes*/
    activeWorkframes = findActiveWorkframes();
    /*Set_Idle*/
    /*If no worframes or thoughtframes available*/
    if(activeWorkframes.isEmpty() &&
            currentWorkframe == null){
        clock = newGlobalClock;
        return -1;
    }
    else{
        /*Set_Act*/
        if(clock == globalClock){
            Wf_Star();//Find active workframe
        }
        else if(currentWorkframe != null){
            Pop_WFStar();
        }
        return timeRemaining;
    }
}

public Set findActiveWorkframes(){
    //Make a copy so original stays untouched
    Set<workframe> tempWorkframes =
        new HashSet<workframe>(workframes);
    activeWorkframes.clear();
    System.out.println(indent+"Active workframes are:");
    for(Iterator<workframe> wfIterator =
            tempWorkframes.iterator();
            wfIterator.hasNext();){
        workframe wf = wfIterator.next();
        /*Pass all beliefs to this class containing a
        thoughtframe, this method will return whether
        or not the thoughtframe is active */
        boolean active = wf.isActive(beliefs, indent);
        /*If active then add to set of all workframes*/
        if(active == true){
            activeWorkframes.add(wf);
        }
    }
    return activeWorkframes;
}

public Set findActiveThoughtframes(){
    //Make a copy so original stays untouched
    Set<thoughtframe> tempThoughtframes = new
        HashSet<thoughtframe>(thoughtframes);
    activeThoughtframes.clear();
    for(Iterator<thoughtframe> tfIterator =
            tempThoughtframes.iterator();
            tfIterator.hasNext();){
        thoughtframe tf = tfIterator.next();
        /*Pass all beliefs to this class containing a
        thoughtframe, this method will return whether
        or not it is active */
        boolean active = tf.isActive(beliefs, indent);
        /*If active then add to set of all thoughtframes*/
        if(active == true){
            activeThoughtframes.add(tf);
        }
    }
    return activeThoughtframes;
}

/*All rules denoted Tf_* */
public void Tf_Star(){
    maxPri = 0; // Maximum priority
    Set<thoughtframe> highestPriThoughtframes =
        new HashSet<thoughtframe>();
    /*Identify thoughtframes with highest priority and
    add to a temporary set*/
    if(highestPriThoughtframes.size() > 1){
        for(Iterator<thoughtframe> tfIterator =
                highestPriThoughtframes.iterator();
                tfIterator.hasNext();){
            thoughtframe tf = tfIterator.next();
            System.out.println(indent+"----"+tf.getName());
        }
    }
    if(!activeThoughtframes.isEmpty()){
        highestPriThoughtframes = findSetOfMaxPriThoughtframes
            (highestPriThoughtframes);
        /*Tf_Select*/
        // if no current thoughtframe
        if(currentThoughtframe == null){
            Tf_Select(highestPriThoughtframes);
        }
        /*Tf_(true/false/once)*/
        /*Tf_true*/
        if(currentThoughtframe.getRepeat().equals("true")){
            //System.out.println(indent+"Selected rule Tf_true");
        }
        /*Tf_once*/
        else if(currentThoughtframe.getRepeat().
                equals("once")){
            // calls method to change repeat value to false
            currentThoughtframe.setRepeat("false");
        }
        /*Tf_false*/
        else{
            // remove thoughtframe from the set
            thoughtframes.remove(currentThoughtframe);
        }
    }
    else{
        System.out.println(indent+"No Active Thoughtframes");
    }
    Pop_TFStar();//Execute thoughtframe
}

/*All rules denoted Wf_* */
public void Wf_Star(){
    maxPri = 0;
    Set<workframe> highestPriWorkframes =
        new HashSet<workframe>();
    /*Identify workframes with highest priority and add to
    temporary set*/
    highestPriWorkframes = findSetOfMaxPriWorkframes
            (highestPriWorkframes);
    /*Wf_Select*/
    // if no current thoughtframe
    if(currentWorkframe == null){
        /*Take top element from set of highest
        priority workframes*/
        Iterator<workframe> wfIterator =
            highestPriWorkframes.iterator();
        workframe tempWork = wfIterator.next();
        // Make new instances of all events and add to a set
        // of events - this is so the event templates won't
        // be changed
        List<event> tempEvents = new ArrayList();
        for(Iterator<event> eventIterator =
                tempWork.getEvents().iterator();
                eventIterator.hasNext();){
            event tempE = eventIterator.next();
            event e;
            //if a conclude needs one constructor
            if(tempE.getType() == eventType.Conc){
                e = new event(tempE.getID(), tempE.getFName(),
                    tempE.getType(), tempE.getAttOwner(),
                    tempE.getAttName(), tempE.getValueOwner(),
                    tempE.getValueOwner2(), tempE.getValue(),
                    tempE.getValueAttr(), tempE.getValueAttr2(),
                    tempE.getValueOperator(), tempE.getBC(),
                    tempE.getFC(), tempE.getVariables());
                e.setIndent(indent);
            }
            // for all other activities another constructor
            else{
                e = new event(tempE.getID(), tempE.getFName(),
                    tempE.getName(), tempE.getWhomWhere(),
                    tempE.getWhomWhere2(), 0,
                    tempE.getActivities(),
                    tempE.getVariables());
                e.setIndent(indent);
            }
            tempEvents.add(e);
        }
        currentWorkframe = new workframe(tempWork.getMas(),
                tempWork.getID(), tempWork.getAgent(),
                tempWork.getName(), tempWork.getRepeat(),
                tempWork.getPriority(), tempWork.getVariables(),
                tempWork.getDetectables(), tempWork.getGuards(),
                tempEvents);

        /*Wf_(true/false/once)*/
        /*Wf_true does nothing*/
        if(tempWork.getRepeat().equals("true")){
            //System.out.println(indent+"Selected rule Wf_true");
        }
        /*Wf_once*/
        else if(tempWork.getRepeat().equals("once")){
            // calls method to change repeat value to false
            currentWorkframe.setRepeat("false");
        }
        /*Wf_false*/
        else if(tempWork.getRepeat().equals("false")){
            // remove thoughtframe from the set
            workframes.remove(tempWork);
        }
    }
    /*WF_Suspend*/
    else if(maxPri > (currentWorkframe.getPriority()+0.3)){
        // make a temp workframe
        workframe temp = new workframe(currentWorkframe.getMas(),
            currentWorkframe.getID(), currentWorkframe.getAgent(),
            currentWorkframe.getName(),
            currentWorkframe.getRepeat(),
            currentWorkframe.getPriority(), currentWorkframe.
                getVariables(),
            currentWorkframe.getDetectables(), currentWorkframe.
                getGuards(),
            currentWorkframe.getEvents());
        double newPri = temp.getPriority() + 0.2;
        temp.setPri(newPri); // assign it a new priority of +0.2
        /*This bit is missing from semantics, it should have been
        included. Basically says to change repeat to false so the
        workframe will be deleted after it has been ran*/
        temp.setRepeat("false");
        // add suspended workframe to set of all workframes
        workframes.add(temp);
        currentWorkframe =null;
        Wf_Star();
    }
    // Decide on what value variables will take
    // needs implementing!
    // Var_Star();
    // Start popping stack. Detectables are checked when
    // an activity is found
    Pop_WFStar();
}

/*The semantics only show Pop_* rules because TF and WF are
identical except for the data structure they access*/
public void Pop_TFStar(){
    List<event> events = new ArrayList<event>
        (currentThoughtframe.getEvents());
    for(Iterator<event> eventIterator = events.iterator();
            eventIterator.hasNext();){
        event e = eventIterator.next();
        /*Rule Pop_TFconc* - Not in semantics but
        same as Pop_WFconc* */
        if(e.getType() == eventType.Conc){
            /*Rule Pop_TfconcB*/
            Pop_TfconcB(e);
        }
    }
    /*Pop_emptyTF: Thoughtframe is empty,
    check for more thoughtframes*/
    // Check if there are more active thoughtframes
    currentThoughtframe = null;
    // populate list of active thoughtframes
    activeThoughtframes = findActiveThoughtframes();
    // if more active then process them
    if(!activeThoughtframes.isEmpty()){
        //Check for more thoughtframes
        Tf_Star();
    }

}

public void Pop_TfconcB(event e){
    Random randomGenerator = new Random();
    //generate a random number between 0..99
    int randomInt = randomGenerator.nextInt(100);
    /*if random number is <= the belief condition and > 0
    or is 100% then update belief*/
    if((e.getBC() > 0 && randomInt <= e.getBC()) ||
            e.getBC() == 100){
        /*find which belief it is*/
        for(Iterator<belief> belIterator = beliefs.iterator();
                belIterator.hasNext();){
            belief b = belIterator.next();
            /*Check if belief matches the event*/
            if(b.getAbout().equals(e.getAttOwner()) &&
                b.getAttribute().equals(e.getAttName())){
                belief tempBelief = b;
                // method assigns the belief the new value
                tempBelief.setValue(e.setValueBelief(beliefs));
                beliefs.remove(b); // remove old belief
                beliefs.add(tempBelief); // insert new belief
                String beliefUpdate = tempBelief.getAbout();
                if(beliefUpdate.equals("current"))
                    beliefUpdate = name;
                beliefUpdate = beliefUpdate.concat("."+
                    tempBelief.getAttribute() +
                    tempBelief.getMathSymbol() +
                    tempBelief.getValue());
                System.out.println(indent+"----Belief Assertion:
                    " + beliefUpdate + " at time " + clock);
                break;
            }
        }
    }
    //Facts not handled in thoughtframes
    // Calls method to remove event from stack
    currentThoughtframe.removeEvent(e);

}
```

```
/*Could probably turn Pop_TFStar and Pop_WFStar into one method
which accepts the list of events*/
public void Pop_WFStar(){
    List events = currentWorkframe.getEvents();
    if(!events.isEmpty()){
        Iterator<event> eventIterator = events.iterator();
        //System.out.println(indent+"Popping event off stack");
        event e = eventIterator.next();
        /*Rule Pop_WFconc* */
        if(e.getType() == eventType.Conc){
            Pop_WfconcB(e);
            Pop_WfconcF(e);
            /*Rule Pop_WfconcBF is not needed when programming
            it this way*/
            // Calls method to remove event from stack
            currentWorkframe.removeEvent(e);
            Pop_WFStar();
        }
        /*Primitive Activity*/
        else if(e.getType() == eventType.PrimAct){
            PopPAStar(e);
        }
        /*Communication activity*/
        else if(e.getType() == eventType.CommAct){
            PopCommStar(e);
        }
        /*Move activity*/
        else if(e.getType() == eventType.Move){
            PopMoveStar(adjacencyMatrix, locs,e);
        }
    }
    /*Check if workframe is empty*/
    else{
        /*Pop_emptyWF: workframe is empty, check for more
        workframes*/
        // Check if there are more active workframes
        currentWorkframe = null;
        // populate list of active workframes
        activeWorkframes = findActiveWorkframes();
        // if more active then process them
        if(!activeWorkframes.isEmpty()){
            Wf_Star();
        }
    }
}

public void Pop_WfconcB(event e){
    System.out.println(indent+"Conclude found:");
    //System.out.println(indent+"Selected rule Pop_WfconcB");
    Random randomGenerator = new Random();
    //generate a random number between 0..99
    int randomInt = randomGenerator.nextInt(100);
    /*if random number is <= the belief condition and > 0 or is
    100% then update belief*/
    if((e.getBC() > 0 && randomInt <= e.getBC()) ||
            e.getBC() == 100){
    /*find which belief it is*/
        for(Iterator<belief> belIterator = beliefs.iterator();
                belIterator.hasNext();){
            belief b = belIterator.next();
            /*Check if belief matches the event*/
            if(b.getAbout().equals(e.getAttOwner()) &&
                b.getAttribute().equals(e.getAttName())){
                belief tempBelief = b;
                // method assigns the belief the new value
                tempBelief.setValue(e.setValueBelief(beliefs));
                beliefs.remove(b); // remove old belief
                beliefs.add(tempBelief); // insert new belief
                String beliefUpdate = tempBelief.getAbout();
                if(beliefUpdate.equals("current"))
                    beliefUpdate = name;
                beliefUpdate = beliefUpdate.concat("."+
                    tempBelief.getAttribute() +
                    tempBelief.getMathSymbol() +
                    tempBelief.getValue());
                System.out.println(indent+"----Belief Assertion:
                    " + beliefUpdate + " at time " + clock);
                break;
            }
        }
    }
}

public void Pop_WfconcF(event e){
    Random randomGenerator = new Random();
    //generate another random number
    int randomInt = randomGenerator.nextInt(100);
    /*if random number is <= the fact condition and > 0 or
    is 100% then update belief*/
    if((e.getFC() > 0 && randomInt <= e.getFC()) ||
        e.getFC() == 100){
        String about = e.getAttOwner();
        if(about.equals("current")){
            about = name;
        }
        /*find which fact it is*/
        if(facts.size() > 0){
            for(Iterator<fact> factIterator = facts.iterator();
                    factIterator.hasNext();){
                fact f = factIterator.next();
                /*Check if belief matches the event*/

                if(f.getAbout().equals(about) && f.getAttribute()
                        .equals(e.getAttName())){
                    fact tempFact = f;
                    // method assigns the fact the new value.
                    tempFact.setValue
                        (e.setValueFact(facts, name));
                    facts.remove(f); // remove old fact
                    facts.add(tempFact); // insert new fact
                    String factUpdate = tempFact.getAbout();

                    if(factUpdate.equals("current"))
                        factUpdate = name;
                    factUpdate = factUpdate.concat("."+
                        tempFact.getAttribute() +
                        tempFact.getMathSymbol() +
                        tempFact.getValue());
                    System.out.println(indent+"Fact Assertion:
                        " + factUpdate);
                    break;
                }
            }
        }
        else{
            fact f = new fact(about, e.getAttName(), "=",
                e.getValue(), name);
        }

    }
}

//Check if detectable active
public void Det_Star(){
    //Get current worframes set of detectables
    Set<detectable> tempDet = currentWorkframe.getDetectables();
    Set<detectable> activeDetectables = new HashSet<detectable>();
    for(Iterator<detectable> detIterator = tempDet.iterator();
            detIterator.hasNext();){
        detectable d = detIterator.next();
        boolean active = d.isActive(facts, indent);
        if(active == true){
            activeDetectables.add(d);
            for(Iterator<belief> belIterator = beliefs.iterator();
                    belIterator.hasNext();){
                belief b = belIterator.next();
                /*Check if belief matches the event*/
                if(b.getAbout().equals(d.getLeftOwner()) &&
                        b.getAttribute().equals
                        (d.getLeftAttribute())){
                    beliefs.remove(b); // remove old belief
                    break;
                }
            }
            //Find the fact and add as a belief
            for(Iterator<fact> factIterator = facts.iterator();
                    factIterator.hasNext();){
                fact f = factIterator.next();
                if(f.getAbout().equals(d.getLeftOwner()) &&
                        f.getAttribute().equals
                        (d.getLeftAttribute())){
                    String tempAbout = f.getAbout();
                    if(tempAbout.equals(name))
                        tempAbout = "current";
                    String tempAtt = f.getAttribute();
                    String tempMath = f.getMathSymbol();
                    String tempValue = f.getValue();

                    belief newBelief = new belief(tempAbout,
                        tempAtt, tempMath, tempValue);
                    beliefs.add(newBelief);
                    System.out.println(indent+"------------
                        Belief Assertion: "+
                        tempAbout + "." + tempAtt + " "
                        + tempMath + " " + tempValue);
                    break;
                }
            }
        }
        else{
            System.out.println(indent+"--------Detectable "
                + d.getName() + " is inactive");
        }
    }
    if(!activeDetectables.isEmpty()){
        System.out.println(indent+"Processing the detectables");
        // 4 = Abort; 3 = Impasse; 2 = Complete; 1 = Continue.
        // Used to determine which has highest priority.
        int detType = 0;

        // Loop decides which action to perform when more than
        // 1 detectable is active
        for(Iterator<detectable> detIterator =
                activeDetectables.iterator();
                detIterator.hasNext();){
           detectable d = detIterator.next();
```

```
            if(d.getType().equals("abort"))
                detType = 4;
            if(d.getType().equals("impasse")&& detType <= 3)
                detType = 3;
            if(d.getType().equals("complete")&& detType <= 2)
                detType = 2;
            if(d.getType().equals("continue")&& detType <= 1)
                detType = 1;
        }
        System.out.println(indent+"detType = " + detType);
        switch (detType) { // Decide which rule to call
            case 1:  Det_Continue();
                     break;
            case 2: Det_Complete();
                     break;
            case 3: Det_Impasse();
                     break;
            case 4: Det_Abort();
                     break;
        }
    }
}

public void Det_Abort(){
    // Check if there are more active workframes
    currentWorkframe = null;
    // populate list of active workframes
    activeWorkframes = findActiveWorkframes();
    // if more active then process them
    if(!activeWorkframes.isEmpty()){
        Wf_Star();
    }
}
public void Det_Continue(){
    System.out.println(indent+"Continue detectable
        activated, carrying on with workframe");
}
public void Det_Complete(){
    System.out.println(indent+"Found a
        Complete detectable!!!");
}
public void Det_Impasse(){
    System.out.println(indent+"Impassing workframe
        through detectable");
}

public void PopPAStar(event e){
    System.out.println(indent+"Primitive activity found:");
    Det_Star(); // Check detectables
```

```
        /*Pop_PASend*/
        if(clock == globalClock){
            // set time remaining to duration of activity
            timeRemaining = e.getDuration();
            /*Method now ends and timeRemaining will be returned
            to MultiAgentSystem.java */
        }
        else{
            /*Find out duration of current activity after time update*/
            // how much time to deduct from activity
            int timeDifference = globalClock - clock;
            // find new duration of the activity
            int newDuration = e.getDuration() - timeDifference;
            /*Pop_PA(t>0): if new duration > 0*/
            if(newDuration > 0){
                int tempDur = e.getDuration();
                // call a method to update the duration
                e.setDuration(newDuration);
                // Set time to clock
                clock = globalClock;
                timeRemaining = newDuration;

            }
            /*Pop_PA(t=0): if new duration = 0*/
            else if(newDuration == 0){
                // remove event from stack
                currentWorkframe.removeEvent(e);
                clock = globalClock; // Set time to clock
                // Set time remaining as zero so system can
                // update after the activity
                timeRemaining = 0;
            }

        }
    }

    public void PopCommStar(event e){
        System.out.println(indent+"Communication activity found");
        Det_Star(); // Check detectables
        /*Pop_commSend*/
        if(clock == globalClock){
            // set time remaining to duration of activity
            timeRemaining = e.getDuration();
            System.out.println(indent+"----
                activity duration  is " + timeRemaining);
        }
            /*Method now ends and timeRemaining will be returned
                to MultiAgentSystem.java */
        else{
```

```
/*Find out duration of current activity
    after time update*/
// how much time to deduct from activity
int timeDifference = globalClock - clock;
// find new duration of the activity
int newDuration = e.getDuration() - timeDifference;
clock = globalClock; // Set time to clock
/*Pop_comm(t>0): if new duration > 0*/
if(newDuration > 0){
    int tempDur = e.getDuration();
    // call a method to update the duration
    e.setDuration(newDuration);
    // Set time to clock
    clock = globalClock;
    // to be returned to MultiAgentSystem.java
    timeRemaining = newDuration;
}
/*Pop_comm(t=0): if new duration = 0
    Looks very different to the semantics because
    the semantics simplifies this greatly.*/
else if(newDuration == 0){
    Set<messages> mess = e.getMessages();
    /*Cycle through to find which agent message
    is to be sent to, in the event it is an agent*/
    for(Iterator<agent> AgIterator = agents.iterator();
            AgIterator.hasNext();){
        agent a = AgIterator.next();
        /*if agent is agent message is to be sent to*/
        if(e.getWhomWhere().equals(a.getName())){
            /*Cycle through the messages*/
            for(Iterator<messages> messIterator =
                    mess.iterator();
                    messIterator.hasNext();){
                messages m = messIterator.next();
                /*Find which belief is being sent*/
                int counter = 0; //If agent doesnt have
                for(Iterator<belief> belIterator =
                        beliefs.iterator();
                        belIterator.hasNext();){
                    belief b = belIterator.next();
                    /*if belief matches message*/
                    if(b.getAttribute().equals
                    (m.getMessAtt()) &&
                    b.getAbout().equals
                    (m.getMessAbout())){
                        belief tempB;
                        String tempAbout =
                            m.getMessAbout();
                        if(tempAbout.equals("current")){
```

```
                        tempAbout = name;
                    }
                    if(e.getWhomWhere() == tempAbout){
                        tempB = new belief("current",
                            b.getAttribute(),
                            b.getMathSymbol(),
                            b.getValue());
                    }
                    else{
                        tempB = new belief(tempAbout,
                            b.getAttribute(),
                            b.getMathSymbol(),
                            b.getValue());
                    }
                    // Send belief
                    a.transferMessage(tempB, clock);
                }
            }
        }
    }
}
/*Cycle through to find which object message is to be
sent to, in the event it is an object*/
for(Iterator<object> obIterator = objects.iterator();
        obIterator.hasNext();){
    object o = obIterator.next();
    if(e.getWhomWhere().equals(o.getName())){
        /*Cycle through the messages*/
        for(Iterator<messages> messIterator =
                mess.iterator();
                messIterator.hasNext();){
            messages m = messIterator.next();
            /*Find which belief is being sent*/
            for(Iterator<belief> belIterator =
                    beliefs.iterator();
                    belIterator.hasNext();){
                belief b = belIterator.next();
                /*if belief matches message*/
                if(b.getAttribute().
                        equals(m.getMessAtt())
                        && b.getAbout().equals
                        (m.getMessAbout()))
                     // Send belief
                    transferMessage(b, clock);
            }
        }
    }
}
// remove event from stack
```

```
            currentWorkframe.removeEvent(e);
            // Set time remaining as zero so system can
            // update after the activity
            timeRemaining = 0;
        }
    }
}

public void PopMoveStar(int [][] adjacencyMatrix,
        Set<locations> locs, event e){
    Det_Star(); // Check detectables
    /*Pop_PASend*/
    if(clock == globalClock){
        //Find point in matrix where current location lies
        int from = -1; //Location moving from
        int to = -1;//Location moving to
        if(e.getDuration() == 0){
            for (Iterator<locations> Locit =
                    locs.iterator(); Locit.hasNext(); ){
                locations l = Locit.next();
                if(location.equals(l.getName())){
                    from = l.getID();
                    break;
                }
            }
            //Find point in matrix where target location lies
            for (Iterator<locations> Locit = locs.iterator();
                    Locit.hasNext(); ){
                locations l = Locit.next();
                if(e.getWhomWhere().equals(l.getName())){
                    to = l.getID();
                    break;
                }
            }
            e.setDuration(adjacencyMatrix[from][to]);
            timeRemaining = e.getDuration();
        }
        /*Method now ends and timeRemaining will be returned to
        MultiAgentSystem.java */
    }
    else{
        /*Find out duration of current activity after time
        update*/
        // how much time to deduct from activity
        int timeDifference = globalClock - clock;
        // find new duration of the activity
        int newDuration = e.getDuration() - timeDifference;
        /*Pop_PA(t>0): if new duration > 0*/
        if(newDuration > 0){
            // call a method to update the duration
            e.setDuration(newDuration);
            clock = globalClock; // Set time to clock
            // to be returned to MultiAgentSystem.java
            timeRemaining = newDuration;
        }
        /*Pop_PA(t=0): if new duration = 0*/
        else if(newDuration == 0){
            location = e.getWhomWhere();
            // remove event from stack
            currentWorkframe.removeEvent(e);
            clock = globalClock; // Set time to clock
            // Set time remaining as zero so system can
            // update after the activity
            timeRemaining = 0;
        }

    }
}

public void Tf_Select(Set<thoughtframe>
        highestPriThoughtframes){
    /*Take top element from set of highest priority
    thoughtframes*/
    Iterator<thoughtframe> tfIterator =
        highestPriThoughtframes.iterator();
    thoughtframe tempThought = tfIterator.next();

    // Make new instances of all events and add to a set
    // of events, this is so the event templates won't be
    // changed
    List<event> tempEvents = new ArrayList();
    for(Iterator<event> eventIterator =
            tempThought.getEvents().iterator();
            eventIterator.hasNext();){
        event tempE = eventIterator.next();
        event e;
        //if a conclude needs one constructor
        if(tempE.getType() == eventType.Conc){
            e = new event(tempE.getID(), tempE.getFName(),
            tempE.getType(), tempE.getAttOwner(),
            tempE.getAttName(), tempE.getValueOwner(),
            tempE.getValueOwner2(), tempE.getValue(),
            tempE.getValueAttr(), tempE.getValueAttr2(),
            tempE.getValueOperator(), tempE.getBC(),
            tempE.getFC(), tempE.getVariables());
            e.setIndent(indent);
        }
        else{ // for all other activities another constructor
            e = new event(tempE.getID(), tempE.getFName(),
                tempE.getName(), tempE.getWhomWhere(),
                tempE.getWhomWhere2(), 0, tempE.getActivities(),
                tempE.getVariables());
            e.setIndent(indent);
        }
        tempEvents.add(e);
    }
    currentThoughtframe = new thoughtframe(
        tempThought.getMas(), tempThought.getID(),
        tempThought.getAgent(), tempThought.getName(),
        tempThought.getRepeat(), tempThought.getPriority(),
        tempThought.getVariables(), tempThought.getGuards(),
        tempEvents);
}
public Set findSetOfMaxPriThoughtframes
        (Set<thoughtframe> hpt){
    maxPri = 0;
    for(Iterator<thoughtframe> tfIterator =
            activeThoughtframes.iterator();
            tfIterator.hasNext();){
        thoughtframe tf = tfIterator.next();
        double pri = tf.getPriority();
        if(pri == maxPri) // if equal to max add to set
            hpt.add(tf);
        // if greater than max, clear the set and add
        else if(pri > maxPri){
            hpt.clear();
            hpt.add(tf);
        }
    }
    return hpt;
}

public Set findSetOfMaxPriWorkframes(Set<workframe> hpw){
    maxPri = 0;
    for(Iterator<workframe> wfIterator =
            activeWorkframes.iterator();
            wfIterator.hasNext();){
        workframe wf = wfIterator.next();
        double pri = wf.getPriority();
        // if equal to max add to set
        if(pri == maxPri)
            hpw.add(wf);
        // if greater than max, clear the set and add
        else if(pri > maxPri){
            hpw.clear();
            hpw.add(wf);
            maxPri = pri;
        }
    }
    return hpw;

}

/*Method takes belief from the messages,
removes previous belief and inserts new*/
public void transferMessage(belief b, int c){
    /*Loop through all beliefs*/
    for(Iterator<belief> belIterator = beliefs.iterator();
            belIterator.hasNext();){
        belief b2 = belIterator.next();
        //If the belief matches
        if(b.getAbout().equals(b2.getAbout()) &&
                b.getAttribute().equals(b2.getAttribute())){
            beliefs.remove(b2);
            break;
        }
    }
    // Add the belief to the belief base i.e. message received
    beliefs.add(b);
    String beliefUpdate = b.getAbout();
    //if(beliefUpdate.equals("current"))
        //beliefUpdate = name;
    beliefUpdate = beliefUpdate.concat("."+b.getAttribute() +
        b.getMathSymbol() + b.getValue());
    System.out.println(indent+"--------Agent "+name+"
        message update: "
        + beliefUpdate + " at time " + c);
}
public String getName(){
    return name;
}
public Set getFacts(){
    return facts;
}

/*public String detectableBeliefUpdate(detectable d){
    String newValue = "";
    for(Iterator<fact> factIterator = facts.iterator();
            factIterator.hasNext();){
        fact f = factIterator.next();
        if(f.getAbout().equals(d.getLeftOwner()) &&
                f.getAttribute().equals(d.getLeftAttribute())){
            newValue = f.getValue();
        }
    }
    return newValue;
}*/
public void setIndent(String in){
    indent = in;
}
}
```

# Bibliography

[1] G. Ali, S. Khan, N. A. Zafar, and F. Ahmad. Formal modeling towards a dynamic organization of multi-agent systems using communicating X-machine and Z-notation. In *Indian Journal of Science and Technology*, volume 5, pages 2972–2977, 2012.

[2] C. Baier and J.-P. Katoen. *Principles of model checking.* MIT Press, 2008.

[3] J. Balasubramaniyan, J. Garcia-Fernandez, D. Isacoff, E. Spafford, and D. Zamboni. An architecture for intrusion detection using autonomous agents. In *Computer Security Applications Conference, 1998. Proceedings. 14th Annual*, pages 13–24. IEEE, 1998.

[4] M. Ball, V. Callaghan, M. Gardner, and D. Trossen. Achieving human-agent teamwork in eHealth based pervasive intelligent environments. In *4th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2010*, pages 1–8. IEEE, 2010.

[5] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. 20(3):207–226, 1983.

[6] R. Bordini, L. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 69–78. IEEE Computer Society, 2008.

[7] R. Bordini, L. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 69–78. IEEE Computer Society, 2008.

[8] R. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking agentspeak. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 409–416. ACM, 2003.

[9] R. Bordini, M. Fisher, and M. Sierhuis. Analysing human-agent teamwork. In *10th ESA Workshop on Advanced Space Technologies for Robotics and Automation (ASTRA 2008), Noordwijk, The Netherlands*, 2008.

[10] R. H. Bordini, M. Dastani, and A. E. F. Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications*, volume 2. Springer, 2009.

[11] R. H. Bordini, M. Fisher, and M. Sierhuis. Formal verification of human-robot teamwork. In *Proceedings of the 4th ACM/IEEE International Conference on*

*Human Robot Interaction, HRI 2009, La Jolla, California, USA, March 9-13, 2009*, pages 267–268. ACM Press, 2009.

[12] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model checking rational agents. In *IEEE Intelligent Systems*, volume 19, pages 46–52. IEEE, 2004.

[13] R. H. Bordini, J. F. Hübner, and R. Vieira. Jason and the golden fleece of agent-oriented programming. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 3–37. Springer, 2005.

[14] J. Bradshaw, A. Acquisti, J. Allen, M. Breedy, L. Bunch, N. Chambers, P. Feltovich, L. Galescu, M. Goodrich, and R. Jeffers. Teamwork-centered autonomy for extended human-agent interaction in space applications. In *AAAI 2004 Spring Symposium*, pages 22–24. AAAI Press, 2004.

[15] J. Bradshaw, P. Feltovich, M. Johnson, L. Bunch, M. Breedy, T. Eskridge, H. Jung, J. Lott, and A. Uszok. Coordination in human-agent-robot teamwork. In *International Symposium on Collaborative Technologies and Systems, 2008. CTS 2008*, pages 467–476. IEEE, 2008.

[16] J. Bradshaw, M. Sierhuis, A. Acquisti, P. Feltovich, R. Hoffman, R. Jeffers, D. Prescott, N. Suri, A. Uszok, and R. Van Hoof. Adjustable autonomy and human-agent teamwork in practice: An interim report on space applications. In *Agent Autonomy*, pages 243–280. Springer, 2003.

[17] J. M. Bradshaw, P. J. Feltovich, M. Johnson, M. R. Breedy, L. Bunch, T. C. Eskridge, H. Jung, J. Lott, A. Uszok, and J. van Diggelen. From tools to teammates: Joint activity in human-agent-robot teams. In M. Kurosu, editor, *HCI (10)*, volume 5619 of *Lecture Notes in Computer Science*, pages 935–944. Springer, 2009.

[18] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. In *Computational intelligence*, volume 4, pages 349–355. Wiley Online Library, 2007.

[19] J. P. Burgess and Y. Gurevich. The decision problem for linear temporal logic. In *Notre Dame Journal of Formal Logic Notre-Dame, Ind.*, volume 26, pages 115–128, 1985.

[20] L. Cavedon, A. S. Rao, L. Sonenberg, and G. Tidhar. Teamwork via team plans in intelligent autonomous agent systems. In T. Masuda, Y. Masunaga, and M. Tsukamoto, editors, *Wolrd Wide Computing and its Applications*, volume 1274 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 1997.

[21] W. Clancey, M. Sierhuis, C. Kaskiris, and R. Hoof. Advantages of Brahms for specifying and implementing a multiagent human-robotic exploration system. In *Proc. 16th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 7–11. AAAI Press, 2003.

[22] W. Clancey, M. Sierhuis, C. Seah, C. Buckley, F. Reynolds, T. Hall, and M. Scott. Multi-agent simulation to implementation: a practical engineering methodology for designing space flight operations. In *Engineering Societies in the Agents World VIII*, pages 108–123. Springer, 2008.

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 8, pages 244–263. ACM, 1986.

[24] P. R. Cohen and H. J. Levesque. Teamwork. In *Noús, Vol. 25, No. 4. Special Issue on Cognitive Science and Artificial Intelligence*, pages 487–512. Blackwell Publishing, 1991.

[25] F. S. de Boer, K. V. Hindriks, W. van der Hoek, and J.-J. C. Meyer. A verification framework for agent programming with declarative goals. In *Journal of Applied Logic*, volume 5, pages 277–302. Elsevier, 2007.

[26] S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: a missing link in automatic termination analysis. In *Proceedings of the 1993 international symposium on Logic programming*, pages 420–436. MIT Press, 1993.

[27] L. Dennis, M. Fisher, M. Webster, and R. Bordini. Model checking agent programming languages. In *Automated Software Engineering*, volume 19, pages 5–63. Springer, 2012.

[28] L. A. Dennis, B. Farwer, R. H. Bordini, and M. Fisher. A flexible framework for verifying agent programs. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pages 1303–1306. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[29] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini. Model checking agent programming languages. In *Automated Software Engineering*, volume 19, pages 5–63. Springer, 2012.

[30] R. Drechsler. *Advanced formal verification.* Springer, 2004.

[31] J. Dyer. Team research and team training: A state-of-the-art review. In *Human factors review*, volume 1983, pages 285–3. Santa Monica, CA: Human Factors Society, 1984.

[32] M. Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic.* Wiley, 2011.

[33] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. *SAT solving for termination analysis with polynomial interpretations.* Springer, 2007.

[34] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6. 1 International Symposium on Protocol Specification, Testing and Verification.* International Federation for Information Processing (IFIP), 1995.

[35] K. Haigh and M. Veloso. Planning, execution and learning in a robotic agent. In *Proceedings of the Fourth International Conference on AI Planning Systems*, pages 120–127. AAAI Press, 1998.

[36] K. Haveland and G. Rosu. Monitoring programs using rewriting. In *Proceedings of 16th IEEE International Conference on Automated Software Engineering(ASE)*, pages 135–143. IEEE Computer Society Press, 2001.

[37] K. Havelund and T. Pressburger. Model checking Java programs using Java Pathfinder. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 2, pages 366–381. Springer, 2000.

[38] H. Hexmoor and G. Beavers. Measuring team effectiveness. In *Applied Informatics-Proceedings*, pages 338–343. Citeseer, 2002.

[39] K. V. Hindriks, F. S. De Boer, W. Van der Hoek, and J. J. C. Meyer. Agent programming in 3APL. In *Autonomous Agents and Multi-Agent Systems*, volume 2, pages 357–401. Springer, 1999.

[40] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[41] G. J. Holzmann. The model checker Spin. In *IEEE Trans. Software Eng*, volume 23, pages 279–295, 1997.

[42] R. v. Hoof. Agent isolutions - website. `http://www.agentisolutions.com`.

[43] M. Huhns and M. Singh. *Readings in agents.* Morgan Kaufmann Pub, 1998.

[44] N. Jennings and M. Wooldridge. *Agent technology: foundations, applications, and markets.* springer, 2002.

[45] S.-S. Jongmans, K. Hindriks, and M. van Riemsdijk. Model checking agent programs by using the program interpreter. In J. Dix, J. a. Leite, G. Governatori, and W. Jamroga, editors, *Computational Logic in Multi-Agent Systems*, volume 6245 of *Lecture Notes in Computer Science*, pages 219–237. Springer, 2010.

[46] R. Kaivola and N. Narasimhan. Formal verification of the Pentium® 4 floating-point multiplier. In *Proceedings of the conference on Design, automation and test in Europe*, pages 20–27. IEEE Computer Society, 2002.

[47] G. Kaminka, M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. Marshall, A. Scholer, and S. Tejada. Gamebots: a flexible test bed for multiagent team research. In *Communications of the ACM*, volume 45, pages 43–45. ACM, 2002.

[48] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Information and Computation*, volume 86, pages 43–68. Elsevier, 1990.

[49] Y. Kesten, A. Pnueli, and L.-o. Raviv. Algorithmic verification of linear temporal logic specifications. In *Automata, Languages and Programming*, pages 1–16. Springer, 1998.

[50] F. Koch, J.-J. C. Meyer, F. Dignum, and I. Rahwan. Programming deliberative agents for mobile services: the 3APL-M platform. In *Programming Multi-Agent Systems*, pages 222–235. Springer, 2006.

[51] T. Kropf. *Introduction to formal hardware verification.* Springer, 1999.

[52] C. Lenz, S. Nair, M. Rickert, A. Knoll, W. Rosel, J. Gast, and A. Bannat. Joint-action for Humans and Industrial Robots for Assembly Tasks. In *Proc. 17th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 2008)*, pages 130–135. IEEE Robotic and Automation Society, 2008.

[53] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of. In *Logic Programming: Proceedings of the Fourteenth International Conference on Logic Programming*, page 63. The MIT Press, 1997.

[54] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Computer Aided Verification*, pages 682–688. Springer, 2009.

[55] K. Marriott and P. Stuckey. *Programming with constraints: an introduction.* MIT press, 1998.

[56] M. McCallum., W. Vasconcelos, and T. Norman. Verification and analysis of organisational change. In *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, pages 48–63. Springer, 2006.

[57] M. Montemerlo, J. Pineau, N. Roy, S. Thrun, and V. Verma. Experiences with a mobile robotic guide for the elderly. In *Eighteenth national conference on Artificial intelligence*, pages 587–592, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[58] N. Muscettola, C. Fry, K. Rajan, B. Smith, S. Chien, G. Rabideau, and D. Yan. On-board planning for new millennium deep space one autonomy. In *Aerospace Conference, 1997. Proceedings., IEEE*, volume 1, pages 303 –318 vol.1, 1997.

[59] NASA. Deep space one - website. `http://science.nasa.gov/missions/deep-space-1/`.

[60] R. Netzer and B. Miller. What are race conditions?: Some issues and formalizations. In *ACM Letters on Programming Languages and Systems (LOPLAS)*, volume 1, pages 74–88. ACM, 1992.

[61] S. I. of Computer Science. SICStus prolog - website. `http://www.sics.se/isl/sicstuswww/site/index.html`.

[62] U. of the West of England. CHRIS: Cooperative Human Robot Interaction Systems Project - Website. `http://www.chrisfp7.eu`.

[63] D. Padua. *Encyclopedia of parallel computing*, volume 4. Springer, 2011.

[64] T. Parr. *The definitive ANTLR reference: building domain-specific languages.* Pragmatic Bookshelf, 2007.

[65] J. Pineau, M. Montemerlo, M. Pollack, N. Roy, and S. Thrun. Towards robotic assistants in nursing homes: Challenges and results. In *Robotics and Autonomous Systems*, volume 42, pages 271–281. Elsevier, 2003.

[66] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[67] A. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, pages 42–55. Springer, 1996.

[68] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In *KR*, pages 473–484, 1991.

[69] G. M. P. Rao Anand S. Bdi agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319. San Francisco, 1995.

[70] M. v. Riemsdijk, F. de Boer, M. Dastani, and J. Meyer. Prototyping 3APL in the Maude term rewriting language. In *Computational Logic in Multi-Agent Systems*, pages 95–114. Springer, 2007.

[71] N. Rungta, F. Raimondi, J. Hunter, and R. Stocker. A synergistic and extensible framework for multi-agent system verification. In *The Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI-13, accepted for publication)*, 2013.

[72] E. Salas, D. Sims, and C. Burke. Is there a big five in teamwork? In *Small group research*, volume 36, pages 555–599. Sage Publications, 2005.

[73] M. Sierhuis. *Modeling and Simulating Work Practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design.* PhD thesis, Social Science and Informatics (SWI), University of Amsterdam, SIKS Dissertation Series No. 2001-10, Amsterdam, The Netherlands, 2001.

[74] M. Sierhuis. Multiagent Modeling and Simulation in Human-Robot Mission Operations. (See `http://ic.arc.nasa.gov/ic/publications`), 2006.

[75] M. Sierhuis, J. Bradshaw, J. Acquisti, R. V. Hoof, R. Jeffers, and A. Uszok. Human-agent teamwork and adjustable autonomy in practice. In *Proceedings of the Seventh International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, 2003.

[76] M. Sierhuis, W. J. Clancey, and M. H. Sims. Multiagent modeling and simulation in human-robot mission operations work system design. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pages 191–200. IEEE, 2002.

[77] M. Sierhuis, W. J. Clancey, R. J. v. Hoof, C. H. Seah, M. S. Scott, R. A. Nado, S. F. Blumenberg, M. G. Shafto, B. L. Anderson, A. C. Bruins, C. B. Buckley, T. E. Diegelman, T. A. Hall, D. Hood, F. F. Reynolds, J. R. Toschlog, and T. Tucker. NASAs OCA mirroring system an application of multiagent systems in mission control. In *Autonomous Agents and Multi Agent Systems Conference (AAMAS)*, 2009.

[78] R. Stocker, L. A. Dennis, C. Dixon, and M. Fisher. Verifying brahms human-robot teamwork models. In L. F. del Cerro, A. Herzig, and J. Mengin, editors, *Joint European Conference on Logics in Artificial Intelligence (JELIA)*, volume 7519 of *Lecture Notes in Computer Science*, pages 385–397. Springer, 2012.

[79] R. Stocker, M. Sierhuis, L. Dennis, C. Dixon, and M. Fisher. A Formal Semantics for Brahms. In *Proc. 12th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*, volume 6814 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2011.

[80] K. Sycara. Integrating agents into human teams. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 46, pages 413–417. SAGE Publications, 2002.

[81] D. Traum, J. Rickel, J. Gratch, and S. Marsella. Negotiation over tasks in hybrid human-agent teams for simulation-based training. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 441–448. ACM, 2003.

[82] D. University of Technology. GOAL - Website. `http://mmi.tudelft.nl/trac/goal`.

[83] M. Winikoff. Implementing commitment-based interactions. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 128. ACM, 2007.

[84] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[85] S. R. Wolfe, M. Sierhuis, and P. A. Jarvis. To BDI, or not to BDI: design choices in an agent-based traffic flow management simulation. In H. Rajaei, G. A. Wainer, and M. J. Chinni, editors, *SpringSim*, pages 63–70. SCS/ACM, 2008.

[86] M. Wooldridge, M.-P. Huget, M. Fisher, and S. Parsons. Model checking for multiagent systems: the Mable language and its applications. In *International Journal on Artificial Intelligence Tools*, volume 15, pages 195–226. World Scientific, 2006.

[87] M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: a survey. In *Intelligent agents*, pages 1–39. Springer, 1995.

[88] M. J. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.

[89] M. J. Wooldridge. *An Introduction to MultiAgent Systems (2. ed.)*. Wiley, 2009.

[90] W. Yeung. Behavioral modeling and verification of multi-agent systems for manufacturing control. In *Expert Systems with Applications*, volume 38, pages 13555–13562. Elsevier, 2011.