Formal Verification of Human-Robot Teamwork

Richard Stocker[a], Louise A. Dennis[a], Clare Dixon[a,*], Michael Fisher[a]

[a]*Department of Computer Science, University of Liverpool, UK*

Abstract

Close collaboration between humans and robots is likely to become increasingly important in scientific, industrial and domestic settings; for example, pilots and autopilots that collaborate together to fly a plane, teams of robot and human builders working on construction projects, and the introduction of autonomous vehicles on public roads. However, before robots can be fully utilised in such situations, a comprehensive analysis of their safety is necessary. The focus of our work is on developing formal verification techniques to analyse key aspects, such as safety, of collaborative activities between humans and robots. We use Brahms, a human-agent-robot teamwork modelling language, to describe such teamwork and develop a formal operational semantics allowing us to translate this into the input language of a standard model checker in order to formally verify these Brahms models. To illustrate our approach we define, translate and verify a hospital scenario in which multiple personal digital assistant agents and robotic helpers assist human doctors and human nurses.

Keywords: teamwork, human-robot interaction, formal verification

1. Introduction

Autonomous systems will soon be used in many areas of everyday life such as in industry, in the home, and in scientific settings. Consequently, there is an increasing need for such systems to interact and cooperate with humans. These systems are now expanding away from simple sensors and embedded hardware to more pervasive machines such as the automated vacuum cleaners commonly available. More sophisticated examples we can expect in the future include manufacturing robots developed to assist humans in the construction of complex artifacts [21], and robot 'helpers' to assist the elderly and disabled in their homes [24, 25]. This cooperation and interaction between humans and robots always raises questions over whether the teamwork between the humans and robots can achieve the desired goals, while ensuring that no bad outcome ever occurs, for example, the human's safety is never compromised. Therefore it is vital to evaluate

*Corresponding author, Tel: +44 151 795 4280, Fax: +44 151 795 4235
Email addresses: `R.S.Stocker@liverpool.ac.uk` (Richard Stocker), `L.A.Dennis@liverpool.ac.uk` (Louise A. Dennis), `CLDixon@liverpool.ac.uk` (Clare Dixon), `MFisher@liverpool.ac.uk` (Michael Fisher)

and analyse such scenarios to ensure that the desired goals are indeed achievable and that safety is maintained. The *challenges* involved with such analyses are:

- to accurately describe high-level human and robot behaviours;
- to exhaustively assess all possible interactions within the team;
- and to carry this analysis out in a correct and (at least partly) effective way.

In this paper, we describe our approach which aims to match a set of requirements (possibly involving safety, capabilities, or interactions) against scenarios involving collaborations between humans, robots, and software agents.

To address the first challenge above we need a way of representing human and robot behaviours and look to existing representations to do this. Our aim is to analyse high level behaviours rather than low level interactions. Many programming languages have been developed to describe the interactions of multiple agents including many that describe agents in terms of mental states and attitudes, the so-called Belief-Desire-Intention (BDI) style of programming language, for example [18, 36, 6]. However, we preferred to focus on a framework that had been explicitly *developed* and *used* for human-robot teamwork. This led us to the Brahms framework [28]. Brahms is a simulation/modelling language (rather than a programming language) in which complex human-agent work patterns can be described and is based on the concept of rational agents. The system has been extensively and successfully used within NASA for the modelling of astronaut-robot planetary exploration teams [9, 30, 29]. Thus, we utilise Brahms to capture the key interactions and behaviours of any human-robot-agent scenario and assume that informal requirements (or required properties of the scenario) have been provided through previous analysis/modelling.

We will employ formal verification, in particular *model checking*, to ensure that high-level requirements are satisfied by the Brahms scenarios in question. Model checking [11, 19] is an automated, algorithmic technique that takes as input a model of the system together with the formal requirements and checks that these requirements are satisfied on all paths through the model. We provide a formal semantics for Brahms and, using this, develop a tool that translates Brahms models into an intermediate representation that can be translated into input to a number of model checkers. Our tool currently translates this intermediate representation into `Promela`, the input language of the model checker `Spin` [19].

Figure 1 demonstrates our process for translating and verifying a Brahms model of a human-agent-robot scenario. Here, the Brahms model is interpreted using the formal semantics we developed in [34] to generate a `Java` representation of the semantic structures relevant to this scenario. These `Java` data structures can then be used to generate `Promela` processes for each human, robot, and agent in the scenario which are suitable for input into the `Spin` model checker [19]. We also translate our requirements into `Promela` properties, and then are able to apply the `Spin` model checker to verify that the required properties hold. This tool thus provides us with a mechanism for formally verifying properties of human-agent-robot teamwork scenarios modelled using Brahms.

The contribution of the work is to provide a viable tool for verifying human agent teamwork via the Brahms language. Specifically our contribution involves:

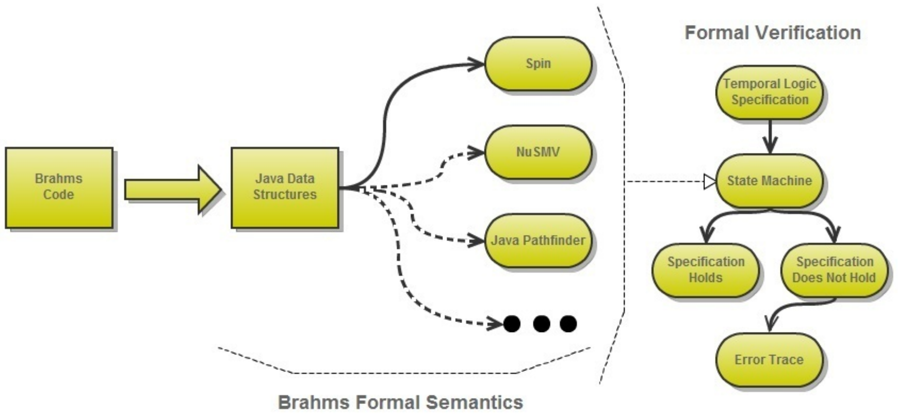

Figure 1: The overall translation and verification process for Brahms.

- the first formal semantics for Brahms;
- the first formal verification tool for the Brahms simulation framework;
- the first formal verification tool specifically for human-agent teamwork;
- a case study demonstrating the system involving the formal verification teamwork between humans, robots and agents in a hospital scenario.

A subset of the formal semantics rules and their operation was illustrated using a simple scenario in [34]. Here we provide the complete set of rules for the formal semantics (Appendix A). In [33] we provided a case study relating to a robot assistant in a smart house and used it to verify some simple examples. Here we provide, and formally verify, a larger, more complex, health-care scenario set in a hospital, demonstrating more features of Brahms, involving more agents, and requiring more teamwork. Full details of the approach can be found in [32]. The significance of the work is that we are now able to formally analyse and assess key high-level requirements of autonomous systems such as robotic assistants that are being developed to work in teams with humans in the workplace or at home.

The rest of this paper describes our framework and demonstrates its use on a specific hospital scenario, where a helper robot and digital agent assistants work together with a number of nurses and a doctor to care for patients. In Section 2 we provide background to Brahms and the model checker `Spin`. In Section 3 we describe the *Digital Nurse Scenario* we use to demonstrate the approach. Section 4 describes the Brahms Formal Semantics used as a basis for the formal verification and Section 5 provides details of the tool that translates from Brahms models to the input to the `Spin` model checker via an intermediate representation. Section 6 discusses the correctness of the system and Section 7 gives details of the verification performed on the Digital Nurse scenario. Related work is presented in Section 8 and conclusions are provided in Section 9.

2. Background

We first provide some background concerning the variety of teamwork we consider, the Brahms modelling language and then the `Spin` model checker.

2.1. Human-Robot Teamwork

Teamwork is a research topic in itself, see for example [15, 27], with differing views about what constitutes teamwork. The need for human-agent or human-robot teamwork occurs in many applications, for example, pervasive systems operating together with humans in health care and at home; collaborative manufacturing; simulation based training; evacuation during emergencies; gamebots and players in video games and space related scenarios with astronauts collaborating with robots or agents. In such scenarios humans and agents/robots need to work together to form a team in order to effectively complete their tasks, but there are many challenges in achieving this. Techniques for overcoming theses challenges include the notions of joint activity [7] and joint intentions [12] (plans or actions to achieve goals), and team plans [8]. We could consider the views of teamwork on a sliding scale; one extreme being a multi-agent system where agents' individual goals are closely linked (such as two individuals sweeping floors in two separate branches of a company), and the opposite end of the spectrum where the agents are involved in joint actions and with joint intentions. Our work, determined by the representation of teamwork in Brahms, lies between these two extremes. We model teamwork that involves communication 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. Here we consider a health care scenario where a robot assists humans with tasks such as turning patients and digital assistants help with scheduling tasks and sending reminders.

2.2. Brahms

Brahms is a multi-agent modelling, simulation and development environment devised by Sierhuis [28] and subsequently developed at NASA Ames Research Center. Brahms is designed to model both human and robotic activity using *rational agents*. Rational agents characterise autonomous entities, able to make their own choices and carry out actions in a *rational* and *explainable* way [37]. As Brahms was developed in order to represent human activities in real-world contexts, it also allows the representation of artifacts, data, and concepts in the form of classes and objects. Both agents, representing autonomous entities, and objects can be located in a model of the world giving agents the ability to detect both objects and other agents, to have beliefs about the objects/agents, and to move between locations. When Brahms executes, it produces a simulation of the humans and agents interacting within some model. Among other things it tracks the time taken over tasks.

For a more detailed description of the Brahms language we refer the reader to [28, 29], but here we highlight the *key aspects* of the language. We provide examples taken from our Digital Nurse scenario (see Section 3) which models aspects of a hospital situation where nurses are equipped with digital assistants.

Agents and Objects. Agents and objects are at the core of every Brahms simulation. Agents model autonomous entities and objects model inanimate objects and sensors, etc. Objects are similar to agents, however they react to external factors while agents react based on their internal *beliefs*. In the Digital Nurse scenario, examples of agents are particular doctors, nurses and digital nurses for example `Doctor_one`, `Nurse_one`, `Digital_Nurse_one` and `Patient_one`, while examples of objects are `Monitor` (a patient monitor) and `Campanile_Clock` (a clock).

Groups and Classes. Groups in Brahms form the hierarchical structure for agents, where agents can be members of groups and groups can also be members of other groups. Groups also provide a template for an agent, so if an agent is a member of a group then it will inherit all the beliefs of the group. It also inherits the group workframes and thoughtframes which govern the behaviour of an agent based on its beliefs. In the Digital Nurse scenario examples of groups are `Doctors`, `Nurse`, `Digital_Nurse` and `Patient`.

Attributes, Relations, Beliefs and Facts. Agents and objects can have their own attributes, relations and beliefs. Attributes are characteristics of the agent such as a nurse agent having Boolean attributes `turnDuty`, `foodDuty` and `break` to denote that the nurse is assigned to turn the patients, to feed patients, and is having a break, respectively. Relations define links between agents and objects, for example "`Digital_Nurse hasDN`" states that the current agent has a relationship called '`hasDN`' with a `Digital_Nurse` agent (in this case, the current agent has a particular digital organiser to assist with nursing duties). Beliefs and facts are tied to attributes and relations, and every belief or fact has to contain either an attribute or a relation, for example an agent could have the belief "`current hasDN Digital_Nurse_One`" which represents that the current agent has a digital nurse assistant called '`Digital_Nurse_One`'. Beliefs and facts differ in that facts represent the real value the attribute or relation has in the simulation, while beliefs represent what the agent believes this value to be.

Geography. In Brahms the agent's world is described using a 'geography' model. Here the world is organised hierarchically, where an area can be conceptual (an *areaDef*, for example `hospital`, `room` or `bed`) or a physical location (*area*, for example `wardOne`, `staffRoom`, `bedTwo` etc). These *area* and *areaDef* descriptions are used to form a hierarchy where: an *area* can be an *instanceof* an *areaDef*; an *areaDef* can *extend* another *areaDef*; and an *area* can be *partof* another *area* (and the distance between two areas is described using a *path*). For example `area bedTwo instanceof bed partof wardOne` denotes that `bedTwo` is a `bed` that is in `wardOne`. Agents are assigned an initial location within the geography, for example, `location:staffRoom` denotes that the agent is initially located in the `staffRoom`.

Workframes and Thoughtframes. Workframes and Thoughtframes govern how agents, objects and the world vary over time. A workframe contains a sequence of activities/actions and belief updates which the agent/object will perform, whereas a thoughtframe only contains sequences of belief updates; a thoughtframe is therefore a restricted workframe which is unable to undertake activities. Workframes can detect (using *detectables*) changes in the environment, update an agent's beliefs accordingly and then decide whether or not to continue executing. Workframes represent the work processes involved in completing a task and thoughtframes represent the reasoning process carried out on the current beliefs. Example workframes are provided in Figures 2 and 3.

Executing Workframes and Thoughtframes: Activities and Concludes. Agents are able to perform activities and *concludes* (belief/fact updates). These are executed via workframes and thoughtframes. Workframes and thoughtframes are very similar. Crucially thoughtframes can only update beliefs. They do not update facts nor perform activities. There are three main types of activity: *primitive*, *move*, and *communication*.

Primitive activities These describe basic actions. Since Brahms is a simulation rather than execution framework, primitive activities have no effect beyond progressing the simulation time. The name assigned to the activity explains what the agent was doing, e.g., the *primitive* activity `haveBreak()` has a duration of 1800 time units and the name implies that the agent was having a break. Representing the effect of the activity would require a 'conclude' to update the agent's beliefs and overall facts (e.g., by setting the attribute `break` to true just before the activity starts and false after it completes).

Move activities Move activities are performed to change an agent's location. As with *primitive* activities they are assigned a duration, which is calculated from the Brahms geography model. When a *move* activity is performed, several things occur: the clock that tracks how much time has elapsed is advanced; the agent's location is changed appropriately; all other agents in the previous location have their beliefs about the agent's location deleted; and the all the agents in the new location recognise this agent has joined them. An example is `move moveToBed(bed b)` as part of the `Nurse` agent which results in the location of the nurse being updated to `b`.

Communication activities These are used for passing messages between agents. For example `communicate requestBreakfastChoice(Patient pat)` in the `Nurse` agent involves a communication between the nurse and a patient about their breakfast food choice. Again, these communications are assigned a duration and, once this duration is over, the beliefs of the other agents are updated corresponding to this communication. However, an agent can only communicate beliefs it already holds.

Detectables. Detectables occur within workframes and can only be executed if their workframe is currently active. When a detectable is executed it imports the fact it "detected" into the agent's belief base and then undertakes one of:

1. *abort* - deletes all elements from the workframe's remaining stack of deeds (activities and belief or fact updates to be carried out);
2. *continue* - carries on executing the workframe;
3. *complete* - deletes only activities from the workframe's remaining stack of deeds (leaving belief or fact updates to be carried out); or

4. *impasse* - suspends the workframe until the detectable's guard is no longer satisfied.

Variables. These allow varieties of quantification within Brahms. 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 of them to work on; *foreach* — work on all, one after another; or *collectall* — work on all at once. An example from our scenario is `collectall(Patient) pat;` where the variable `pat` is used to range over all the `Patient` agents.

A Brahms simulation contains a set of agents (representing robots, humans or actual agents) and a scheduling system which manages a clock recording global time in the simulation. Since agent actions have durations, the scheduler will examine each agent to see how much longer any action the agent is performing will take and then advance the clock to the next significant point in time, typically when the agent with the shortest duration action finishes. By doing this the scheduler maintains synchronicity throughout a simulation, ensuring that the order in which the agents execute does not affect the outcome of the simulation.

Finally, we note that Brahms has now moved to the industrial arena and is able to describe complex human-agent-robot teamwork. In particular, it has been used by NASA in describing and analysing prospective astronaut-robot teamwork on Mars [9].

2.3. Formal Verification, `Promela` *and* `Spin`

Formal verification represents a family of techniques aimed at assessing whether a system satisfies its specification. We particularly use a fully automated, algorithmic technique known as *model checking* [11]. A model checker takes a description of the system together with some requirement expressed in a formal logic. The model checker exhaustively checks the formal requirement against *all* paths through the system. If a path is found in which the property does *not* hold then a trace of that path is provided.

In this paper we use the `Spin` model checker [19]. `Promela` (Process/Protocol Meta Language), the input language for `Spin`, was designed to be a simple, high-level, multi-process language. Processes are a key part of `Promela`, being asynchronous by default. `Promela` provides three basic control flow constructs: case selection; repetition; and unconditional jump.

`Spin` itself is an on-the-fly reachability analysis system [19]. It accepts specifications in the form of *linear temporal logic* properties, which are translated into *Büchi automata* — finite automata over infinite input sequences [16]. `Spin` then examines all possible runs through the `Promela` program, running the *Büchi automaton* in parallel in order to assess whether the temporal requirements are satisfied [17].

3. Digital Nurses: An Example of Brahms Model

This scenario was developed to demonstrate and verify increasingly complex human-agent teamwork. Additional non-determinism is represented through the event of a heart attack so the protocols in the case study could be tested in an emergency

3.1. Overview of Scenario

The Digital Nurse scenario was created to have multiple agents (some virtual) and humans working together as part of a team. The scenario involves two nurses, one doctor, three digital nurses, one robot and a patient monitor. The goal of the scenario is to take care of five patients who need to be given food, water, medicine and be turned in their beds. Additionally, certain patients are at risk of a heart attack, which provides an emergency situation within the scenario. The nurses have the duty 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 (secondary) nurse covers the active nurse in the simulation during the (primary) 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 partly acts as a scheduler for the nurses, reminding them when to perform their duties and informing them of emergencies. The doctor in the scenario checks the patients, prescribes medication and responds to emergencies. The robot assists 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.

3.2. Brahms Representation

The scenario is modelled in Brahms using twelve Brahms agents and two Brahms objects. The agents are: two nurses (modelled as humans), one doctor (modelled as a human), three digital nurses (one for each nurse, and one for the doctor, all modelled as software agents), five patients (modelled as humans) and one robot (modelled as a hardware/robotic agent). The objects are: a *monitor* to monitor the patients and a *clock*.

The patients drink water every so often, make a breakfast choice when prompted and can, unexpectedly, suffer a heart attack. The patient will only recover from a heart attack if the doctor, robot and a nurse all respond to treat them.

The `Monitor` object keeps track of all the agents. It can detect when an patient's water supply is low and communicates this to the appropriate digital nurse. The `Monitor` also checks the patients' vital signs and so knows when one has a heart attack, dies or is no longer having a heart attack. The `Monitor` is also used for counting durations e.g., it notes when a heart attack occurs and increments a counter until the patient is either dead or has been resuscitated.

One of the nurses (`Nurse_one`) is assigned the responsibility of turning the patients but cannot do so until the robot is there to assist. We use the implementation of "turning the patients" as an example of teamwork in this model and illustrate how this is represented in Brahms. To do this two workframes are used: '`wf_turnOne`' (see Figure 2) and '`wf_turnTwo`' (see Figure 3). Workframe `wf_turnOne` identifies a patient and prepares them for turning by assembling the appropriate resources. Once this has happened then `wf_turnTwo` is executed to actually turn the patient. The workframe'`wf_turnOne`' has two quantifiers, `forone(Patient) pat` and `forone(bed) b`, and these variables find a patient and a bed that match the guard conditions and assign them to the variables `pat` and `b`. The guard conditions are specified using the `when` token, which ensures that the nurse must believe the time is 8 (`current.perceivedTime = 8`) and that the nurse in question is on turn duty (`current.turnDuty = true`). We refer to the other guard conditions as "binding conditions", i.e., they are used to bind objects/agents to this workframe. These specify that the agent selected requires turning (`pat.needTurning = true`), has not already been turned (`pat.turned = false`), has not been prepared for turning (`pat.readyToBeTurned = false`), or that the bed is in the location of the patient selected (`pat.location = b`). If there are no agents that meet all these conditions then the workframe is *not* flagged as 'active'. If the workframe is selected for execution then the execution is performed through the activities and concludes in the `do` section. Here the nurse:

- performs a move location to the location of bed `b` first (`moveToBed(b)`);
- concludes that the patient is ready to be turned (`conclude((pat.readyToBeTurned = true))`);
- informs the robot which patient is to be turned (`patientToTurn(pat)`);
- waits for the robot to arrive (`waitToTurn()`); and
- if the robot doesn't arrive then the nurse concludes that the patient is not ready to be turned (`pat.readyToBeTurned = false`), this will then leave the workframe open for execution again since the repeat variable is set to true (`repeat: true`).

If the robot arrives at the bed while the nurse is waiting then the detectable `waitForRobot` will 'fire'; this is fired when the nurse 'detects' the robot's location is the same as the patient's (`detect((pat.location = Robot.location))`). When this detectable condition is true the action that is performed is `abort`, meaning the workframe will terminate and the conclude (`conclude((pat.readyToBeTurned)) = false` which indicates that the patient is not actually ready for turning) will not fire.

If workframe `wf_turnOne` is aborted then the workframe `wf_turnTwo` can become active. Workframe `wf_turnTwo` will be bound to the patient used in `wf_turnOne` by the `readyToBeTurned` flag which was set to true in `wf_turnOne` and which appears in the workframe guard. When executing this workframe a `turnPatient()` primitive activity is called to model the nurse turning the patient, a 'conclude' is performed so the nurse believes the patient has been turned, and two other 'concludes' are used to reset the patient's state.

As well as turning patients, described in detail above, `Nurse_one` responds to resuscitate patients when they have had a heart attack, asks patients what they want for breakfast and places their orders.

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 on messages, and so has workframes:

1. reminding the nurse when it is time for breakfast;
2. sending breakfast orders to the robot;

```
workframe wf_turnOne {
  repeat: true;
  priority: 1;
  variables:
   forone(Patient) pat; /* Variable bound to patient that needs turning */
   forone(bed) b;       /* Variable bound to patient’s location */

  detectables:          /* Detect if robot is at 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 {
    moveToBed(b);                           /* Move to location identified */
    conclude((pat.readyToBeTurned = true)); /* Flag for wf_turnTwo to true */
    patientToTurn(pat);              /* Inform robot which patient to turn */
    waitToTurn();                          /* Wait for the robot to arrive */
    conclude((pat.readyToBeTurned = false)); }
  }               /* Robot has not arrived so set wf_turnTwo flag to false */
```

Figure 2: `wf_turnOne`

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

  /* Guards check that both wf_turnTwo flag is true and
      this nurse is responsible for turning patients */
  when(knownval(pat.readyToBeTurned = true) and knownval(current.turnDuty = true))
  do {
    turnPatient();                  /* Turn the patients */
    conclude((pat.turned = true)); /* Reset all values and flags */
    conclude((pat.readyToBeTurned = false));
    conclude((pat.timeSinceTurned = 0)); } }
```

Figure 3: `wf_turnTwo`

3. informing the robot of medication that the doctor has prescribed;
4. informing the nurse when it is break time;
5. arranging another nurse to cover a nurse's break; and
6. informing the nurses and doctor of when a patient has a heart attack.

All the agents have thoughtframes for managing 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. An emergency event is provided in the form of a "heart attack". This is encoded using a Boolean attribute `heartAttackRisk` and associated workframe and thoughtframe as part of the group `Patient`.

4. Brahms Formal Semantics

Next we describe a formal *operational semantics* for Brahms which provides the theoretical basis for our verification. (A summary of this was provided in [34].) The full formal semantics is set out in Appendix A. A Brahms semantic model is represented as a 5-tuple:

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

Where Ags is the set of all agents, ag_i is the agent currently under consideration, B_ξ is the belief base of the system (used to synchronise the agents, e.g. agent i's next event finishes in 1000 seconds), F is the set of facts in the environment (e.g. this nurse is using `Digital_Nurse_One`) and T_ξ is the current time of the system.

The agents (Ags, and ag_i), in turn, have a 9-tuple representation:

$$\langle ag_i, \mathcal{T}, \mathcal{W}, stage, B, F, T, \mathit{TF}, \mathit{WF} \rangle$$

Here ag_i is the identification of the agent; $\mathcal{T}$, the current thoughtframe; $\mathcal{W}$, the current workframe; $stage$, the current stage of the agent's reasoning cycle; B, the agent's beliefs; F, the set of facts about the world; T, the agent's internal time; TF the agent's set of thoughtframes; and WF, the agent's set of workframes. Here $stage$ controls which rules in the operational semantics are currently applicable to the agent or if the agent is in a 'finish' (*fin*) or 'idle' ($idle$) stage.

The (operational) semantics is then represented as a set of transition rules of the form

$$\langle \mathit{StartingTuple} \rangle \xrightarrow[\mathit{ConditionsRequiredForActions}]{\mathit{ActionsPerformed}} \langle \mathit{ResultingTuple} \rangle$$

Here, '$\mathit{ConditionsRequiredForActions}$' refers to conditions which must hold before this rule can be applied, while '$\mathit{ActionsPerformed}$' represents changes to the agent, object or system state which, for presentational reasons, can not be easily represented in the resulting tuple. Finally, it is assumed that all agents and objects can see, and access, everything in the overall system's tuple, e.g. T_ξ.

The semantics is split into two groups of rules: the first group concerns the global system and represents the functioning of the scheduler; the other group acts upon individual agents. Rules for the scheduler act as global arbiters, instructing agents when to start, suspend, or terminate. Rules for the individual agents choose activities and update beliefs, etc. An agent first processes *thoughtframes*, then *detectables* (both of which may update the beliefs), and then *workframes* which may initiate activities. For example, there are rules informing an agent how to select a thoughtframe based on whether its beliefs match the thoughtframe guard conditions and whether the thoughtframe's priority is sufficiently high. The rules governing activities communicate with the system to inform it of the activity's duration. When no agent can apply any more operational rules, control returns to the scheduler which examines all the agents' activities to determine which will conclude first and at what time it will finish. The scheduler then moves the global (simulation) clock forward accordingly, and hands control to the rules governing the behaviour of the individual agents once more.

Figure 4 shows a simplified flow of control for the semantic rules for the scheduler while Figure 5 shows a simplified flow of the semantics for an agent. In Figure 5 the rectangles labelled A6, A11, A14 and A15 are shaded to represent the non-determinism in the system. The scheduler, in Figure 4, starts off by initialising everything, for example, agents, objects, etc., in state S1. During this initialisation the scheduler tells the agents to begin execution. In S2 the scheduler waits for a response from all the agents about the duration of their activity.

The agents, in Figure 5, 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. In box A6 a thoughtframe is chosen with the highest priority. If several thoughtframes share the highest priority then one is chosen at random. In A7 a set of all active workframes is selected. If this set is empty then the agent moves to A9 and flags itself as idle. If there are active workframes then the agent moves to state A11 where it randomly selects one of these workframes. 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. Because belief and fact updates are associated with certainty values two random numbers are generated. A14A compares the first random number against the conclude's *fact certainty*. If the *fact certainty* is less than the random number then the fact is updated in A14B; if it is greater the fact is not updated. The agent then moves to A14C where the second random number is compared to the conclude's *belief certainty* and the relevant belief may be updated. The agent then returns to A10 to check if there are more elements left on the workframe deed stack. 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 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 S3 and calculates the shortest duration. 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 S6 and S7 from state S4 to terminate the simulation. If the scheduler did find a duration greater than -1 in S4 then it moves its clock forward

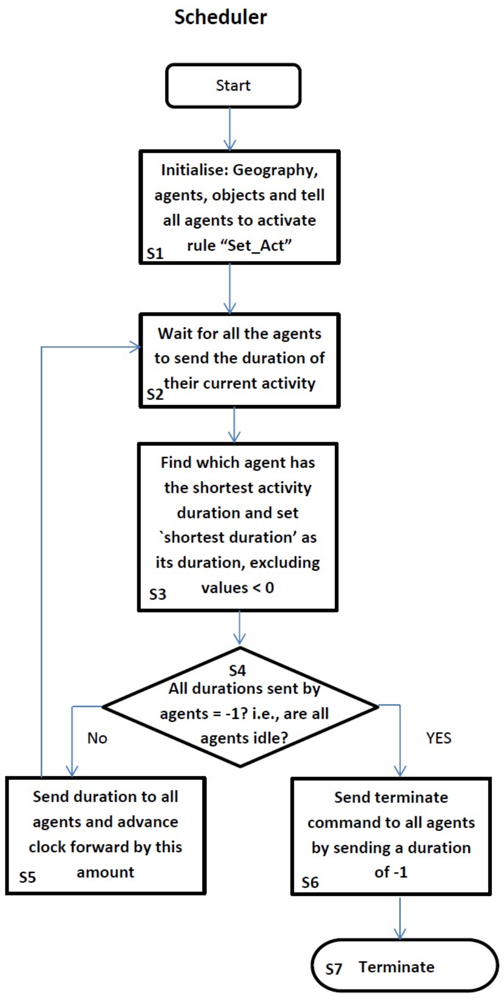
Scheduler
Start
Initialise: Geography, agents, objects and tell all agents to activate rule "Set_Act"
S1
Wait for all the agents to send the duration of their current activity
S2
Find which agent has the shortest activity duration and set `shortest duration' as its duration, excluding values < 0
S3
S4
All durations sent by agents = -1? i.e., are all agents idle?
No
YES
Send duration to all agents and advance clock forward by this amount
S5
Send terminate command to all agents by sending a duration of -1
S6
S7 Terminate

Figure 4: Overview of the Scheduler's Semantics

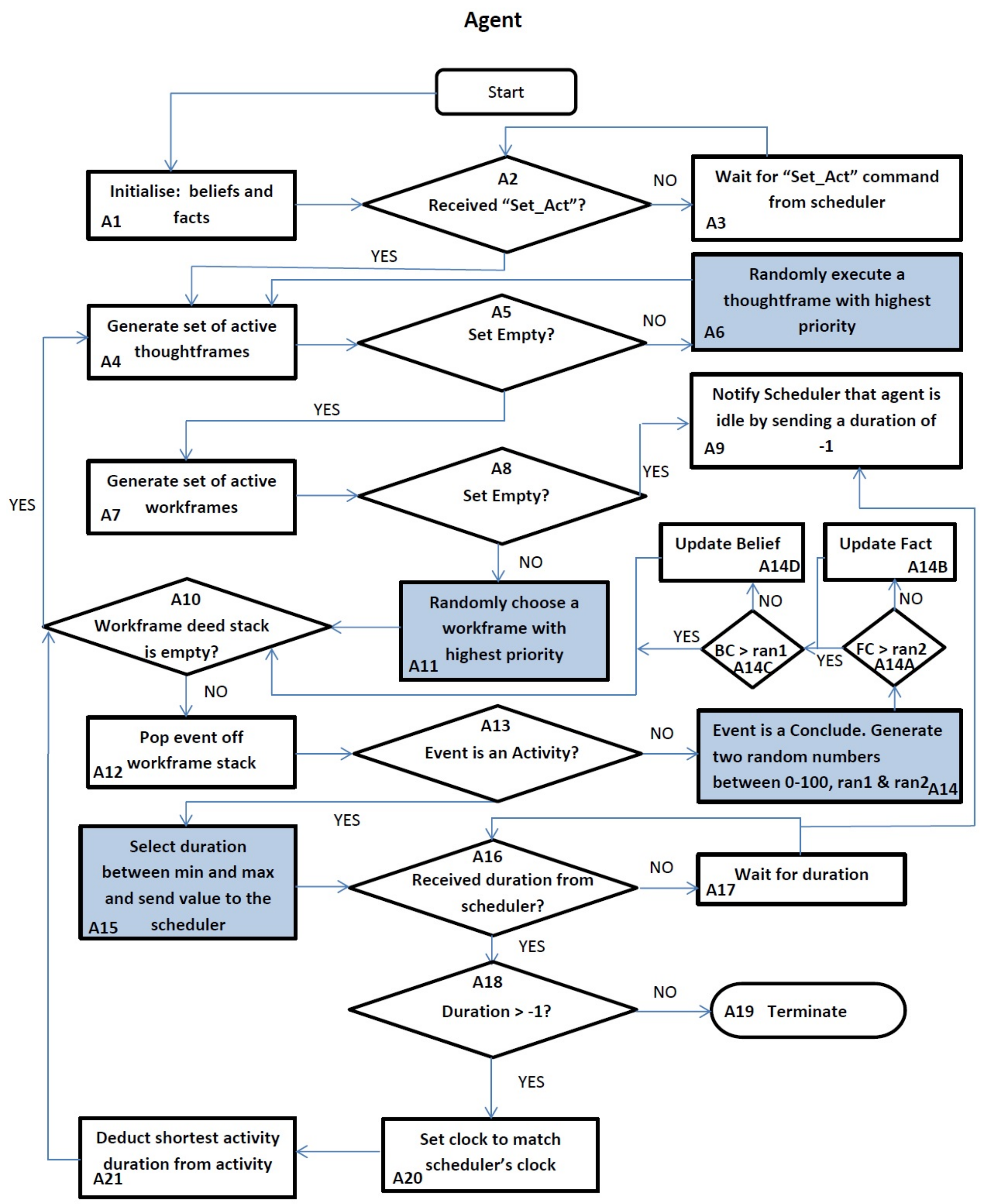
Agent
Start
A1 Initialise: beliefs and facts
A2 Received "Set_Act"?
NO
YES
A3 Wait for "Set_Act" command from scheduler
A4 Generate set of active thoughtframes
A5 Set Empty?
A6 Randomly execute a thoughtframe with highest priority
A7 Generate set of active workframes
A8 Set Empty?
A9 Notify Scheduler that agent is idle by sending a duration of -1
A10 Workframe deed stack is empty?
A11 Randomly choose a workframe with highest priority
Update Belief A14D
Update Fact A14B
BC > ran1 A14C
FC > ran2 A14A
A12 Pop event off workframe stack
A13 Event is an Activity?
A14 Event is a Conclude. Generate two random numbers between 0-100, ran1 & ran2
A15 Select duration between min and max and send value to the scheduler
A16 Received duration from scheduler?
A17 Wait for duration
A18 Duration > -1?
A19 Terminate
A20 Set clock to match scheduler's clock
A21 Deduct shortest activity duration from activity

Figure 5: Overview of a Brahms Agent's Semantics

by this duration calculated in state S3 and moves back to waiting for a response 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 states A20 and then A21 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.

5. Brahms Verification

Here we show how the formal semantics of Brahms can be used to generate input to a model checker. In particular, we develop an intermediate representation from which we can potentially generate input to a variety of different model checkers.

5.1. Our Approach

We decided to use `Java` to create an intermediate model of the Brahms program prior to translation into the input language of any model checker. An intermediate representation of the formal model in `Java` opens up the option of creating a number of translators into the input languages of a variety of model checkers. This would allow different model checking formalisms to be used; potentially allowing us to check probabilistic and epistemic properties as well as purely temporal ones.

The initial target chosen was the `Spin` model checker as this is a widely used, effective and stable system. In addition, `Spin`'s input language, `Promela`, is a higher level language than most other model checker input languages, making it easier to represent the Brahms agents/objects. `Spin` also has the ability to run `Promela` code as a simulation, making it possible to compare the output of the `Promela` version with the output of the Brahms simulation.

Figure 1 describes our process for verifying Brahms scenarios and highlights the potential for flexible translation to many different model checkers.

5.2. Java Representation

The `Java` classes developed capture the structural aspects of the Brahms models required by the operational semantics of Brahms. This is a syntactic transformation of the Brahms model and its underlying elements into Java data structures. The Brahms syntax is stored in the following classes representing different aspects of Brahms: *MultiAgentSystem*; *agents*; *groups*; *classes*; *objects*; *workframes*; *thoughtframes*; *attributes*; *relations*; *beliefs*; *facts*; *activities*; *events*; *concludes*; *communication messages*; *detectables*; *guard conditions*; *variables*; *geography: areas*; *geography: locations*; *geography: paths*; and *geography: path lengths*. The `Java` class `MultiAgentSystem` is used to represent the 5-tuple from the operational semantics for Brahms models. The structure and relationship between the `Java` classes represents the Brahms syntax as required by the operational syntax. For example the `agent` class represents the 9-tuple from the operational semantics for agents contains instance variables relating to (sets of) *beliefs*, *facts workframes* and *thoughtframes* etc. A parser was created to read Brahms code and export the data from the simulation into the `Java` data structures. The `main` class reads in the Brahms code and then executes the parser. The parser uses the `MultiAgentSystem` class as the base class which instantiates all the other classes and begins the translation process.

The dynamic aspects of the operational semantics, for example, the current *beliefs* and *facts* of an *agent*, the current *agent* under consideration, or the time cannot be modelled at this stage as these issues depend on the execution. The code for parser and the twenty-two developed classes can be found in [32].

5.3. `Promela` *Representation*

Although `Promela` is an appropriate input language for model checking it has more restrictive data types and control structures than a general purpose programming language such as `Java`. As such, providing a one to one correspondence between the `Java` data structures and associated rules in the operational semantics and `Promela` was not possible. Arrays are the main data structure available in `Promela` and so an array-based representation was required for most of the `Java` data structures. The intermediate representation was static and so did not implement the transitions in our semantics, however these did need to be implemented in the translation to `Promela`. These were implemented using if statements and loops. We provide an example of the translation the agent data structures into `Promela` processes below.

The `Promela` translation has a separate process (termed *proctype* in `Promela`) for each agent named `proc_` followed by the agent's name (e.g.,`proc_Nurse_one`). The components of the 9-tuple in the Brahms semantics that represent an agent are primarily represented by arrays. For instance, the agent's current workframe is represented as a one-dimensional array and treated as a stack. The array is labelled `wf_stack` followed by the agent's name, e.g., `wf_stackNurse_one`.

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

Figure 6: Array structure for workframes

For an example, see Figure 6. The first six indices 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 either belief updates or activities; see Figure 6. The current thoughtframe is represented in a similar fashion.

Beliefs and facts in Brahms are tied to the attributes and relations of an agent, e.g., the agent `Nurse_one` believes `Patient_one` needs turning (`needTuring = true`). To model this in `Promela` every agent is assigned a belief about every attribute, even if it does not own that attribute. This is modelled in `Promela` using a one dimensional array for each attribute. Facts are modelled in a similar way.

Sets of thoughtframes or workframes for an agent are two dimensional arrays. One dimension captures the details of a workframe (or thoughtframe) as shown in Figure 6 and the other dimension shows the workframes for that agent. Relationships between agents or objects are also modelled using two dimensional arrays.

6. Correctness Issues

We claim to verify Brahms program, however we are not using the *actual* Brahms interpreter as part of our verification but are instead converting Brahms programs first into an intermediate Java representation and then into `Promela` code. This naturally raises the question of whether a program that has been declared correct by our system would actually behave correctly if executed in the existing Brahms simulation engine.

There are several aspects to this question: the correctness of our Brahms *semantics*; the correctness of our *translation* from the Brahms semantics into Java data structures; and the correctness of the *translation* into `Promela`. These are considered below.

6.1. Correctness of the Brahms Semantics

As mentioned previously we first had to develop a (first) formal semantics for Brahms. For confidentiality reasons, we had no access to the Brahms implementation code, so the development of the formal semantics was carried out by a combination of reading existing Brahms documentation, and experimentation using Brahms to observe the behaviour of various Brahms constructs in the simulator. This was carried out in in collaboration with the designer of Brahms and the accuracy of the resulting semantics was confirmed by NASA engineers who used Brahms regularly. Additionally the semantics we developed were used by NASA in their own work [20], further confirming their accuracy. Given that it was impossible to access the actual implemented semantics of the Brahms language this approach brings us as close as feasibly possible to an accurate semantics. We note that, as we consulted with NASA engineers, our formal semantics at least corresponds to the *intended* semantics for Brahms. Furthermore, given that the primary purpose of Brahms is to model situations in order to assess their correctness we have, at the very least, provided a semantics for an agent-based modelling language which serves the same purpose.

To summarise, the semantics have been seen and closely examined by those who know the framework the best and know what it was designed to do. They have studied the rules and also used them in the development of their own verification techniques, thus confirming that they accurately represent Brahms' intended purpose.

6.2. Correctness of the Translation into Java Data Structures and then into `Promela`

As mentioned in Section 5.2 the translation takes the components of the formal semantics and translates them into `Java` data structures. The `Java` classes closely model the key aspects of the Brahms language capturing the elements of the Brahms semantics. The two main parts are the translation of the 5-tuple representing Brahms semantics into the `MultiAgentSystem` class and the translation of the 9-tuple representing agents into the `agent` class. These classes were then translated into a `Promela` model. No formal validation was performed at these points. However the code was reviewed in order to provide an informal justification of correctness.

6.3. Correctness of the Final System

In order to assess the overall correctness of our system, we performed a direct comparison between the outputs of the `Promela` model against the existing Brahms framework. The `Spin` model checker can execute `Promela` code in simulation as using it to create 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 run in the Brahms framework. As each transition rule in the operational semantics was implemented, several tests were run using Brahms programs that required the additional functionality supplied by the rule. The observable behaviour of these programs (updates and activities) was compared to their behaviour in the Brahms framework and the rule and/or system code adjusted if necessary to ensure identical behaviour.

Second, although we intend the system to verify the construction of Brahms models, the model-checking process can also be used to validate the implementation of the system. If a property that is expected to hold can not be verified using `Spin` (or conversely if a property that is not expected to hold can be verified) then this result can be examined and an explanation sought for the discrepancy. In the early stages of the system development, this process revealed issues with the system implementation rather than with the Brahms model itself. As the implementation progressed and new Brahms models were investigated, discrepancies were more frequently attributable to errors in the model until, with the final few models investigated, no discrepancies arose because of errors in the underlying system implementation.

7. Digital Nurse Scenario:Verification

The `Promela` representation of the scenario was automatically generated using our tool, which took a Brahms description of this scenario and parsed it into `Java` data structures, identifying the agents, objects, workframes, thoughtframes, activities, etc. From these data structures `Promela` code was automatically generated. The `Promela` code has separate processes for the scheduler and for each agent and object in the simulation, e.g., a process for every nurse, doctor, patients, digital assistant and robot. The `Promela` code can then be run in simulation mode, which is useful for comparing the output against a Brahms simulation, or in verification mode.

Once we had `Promela` code representing the scenario, we performed verification using the `Spin` model checker. We proved both safety and liveness properties where we ensure something will always happen (or always not happen) or that something will eventually happen (or eventually not happen). In a scenario such as in a hospital, time is crucial so many requirements involve time bounds e.g., if someone has a heart attack then they will receive assistance within a certain time.

7.1. Sample Properties to Prove

We used a range of logical properties for the scenario; note that in *temporal logic*, $\Diamond\phi$ means that "ϕ will be true at *some* moment in the future", while $\Box\phi$ means that "ϕ 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 properties are all based on the beliefs of the agents or facts in the system.

Example Patient Properties.

P1: This property was designed to evaluate 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 means "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, and when they do so we consider the resuscitation rate to be 100%. If some members do not 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 temporal logic formula is $\Box(a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge a_5)$ where a_i means "patient i is alive".

P3: One of the roles of the nurse is to turn those 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 a patient ends 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 be part of the scenario but more of an addition to aid verification. The property, explained in English, is: *the patients do not wait more than 1 hour when they need to be turned.* The property, as a temporal logic formula is: $\Box(a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge a_5)$ where a_i means "the time patient i waits to be turned is less than 1 hour".

Example NURSE Properties.

N1: A requirement of the scenario is that when the nurse on duty (i.e., `Nurse_one`) takes 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 represents "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 nurse takes a break. The property, explained in English, is: *there is always a nurse on duty*. The property, as a temporal logic formula is: $\Box(a_1 \vee a_2)$ where a_i represents "nurse i 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 the nurse, the doctor and the robot will resuscitate the patient within 4 minutes*. The property, as a temporal logic formula is: $\Box(a \Rightarrow \Diamond(n \wedge d \wedge r \wedge e))$ where

a = "patient_one has a heart attack"
$n/d/r$ = "the nurse/doctor/robot performs resuscitation workframe"
e = "the time since heart attack is less than 4 minutes"

Example Robot Properties.

R1: One of the robot's duties is to ensure the patient's water jug always contains water. In the scenario we have a sensor attached to the object monitoring the patient 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 important 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 represents "the 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: $\Box((a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge (a_3 \vee b_3) \wedge (a_4 \vee b_4) \wedge (a_5 \vee b_5))$

a_i = "the robot has no belief about patient i's medication"
b_i = "the robot's belief about patient i's medication matches the doctor's belief"

Example Digital Nurse Properties.

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: *the digital nurse will notify the nurse it is breakfast time within 1 hour*. The property, as a temporal logic formula is: $\Diamond(a \land b)$ where a represents "breakfast has been announced" and b represents "the 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 temporal logic property is: $\Box(a \Rightarrow \Diamond(n \land r \land d \land e))$ where

$$\begin{aligned} a &= \text{“the patient has a heart attack”} \\ n/r/d &= \text{“the nurse/robot/doctor believes the patient has had a heart attack”} \\ e &= \text{“the time since heart attack is less than 2 minutes”} \end{aligned}$$

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 nurse_one to take a break within 1 hour of when it is due*. The property, as a temporal logic formula is is: $\Diamond(a \land b)$ where a represents "the digital nurse believes the nurse has gone on a break" and b represents "the time is at least 10 but not 11".

7.2. Verification Results

All the properties above were verified using our tool, in tandem with `Spin`. The verification results for these properties can be found in Figure 7 which shows the verification time and number of states stored for each property specified in Section 7.1.

7.3. Digital Nurse Scenario Discussion

This Digital Nurse Scenario was created to show a simulation of robots and digital assistants aiding nurses and doctors in their daily duties. Although simple by comparison to a normal day in a hospital, is aimed at demonstrating how we intend our tool to be used for verifying human-agent-robot team work. The Digital Nurse Scenario took some small key aspects of day-to-day activities in a hospital, with a robot to fetch and assist, and digital assistants to remind and inform doctors and nurses. Examples of activities simulated are turning patients, administering medication, and feeding the patients. A non-deterministic, emergency element was a patient having a heart attack, which could happen at a set number of intervals. The aim of the verification was

Property	Time (seconds)	States
P1	63.6	103,618
P2	148	103,618
P3	150	230,864
N1	64.8	101,594
N2	154	230,864
R1	148	230,864
R2	146	230,864
DN1	76.8	114,534
DN2	149	230,864
DN3	72.6	101,184

Figure 7: Digital Nurse scenario's verification performance

to prove that the critical requirements would be met no matter when the emergency occurred. Requirements, such as patients receiving medication at the correct times, patients not waiting too long to be turned, and an emergency are resolved in a timely fashion. The Digital Nurse Scenario was first implemented in Brahms and manually analysed for correctness, and a translation to equivalent `Promela` code was produced using our tool. The properties specified in Section 7.1 were implemented in `Spin` and verified. All properties were successfully verified and the performance results can be found in Figure 7.

8. Related Work

Next we consider related work. This is split into three main areas: other applications of Brahms; the application of the research developed in this paper to other areas; and verification of multi-agent systems.

Brahms has been used at NASA for modelling astronaut-robot planetary exploration teams [9] and for developing a human-behavioural agent model for communications between Mission Control and the International Space Station [10, 31]. The latter has been used to develop a multi-agent software system that has taken over all the routine tasks from a human (about 80% of the workload). Brahms has also been used to describe a possible scenario of human-agent teamwork during a Mars exploration mission in [4].

The Brahms to `Promela` tool described in this paper has also been used to verify an autonomous robotic assistant in [35]. The robot (a Care-O-bot®) is located in the University of Hertfordshire's Robot House, a suburban three bedroom house. The house is equipped with sensors to detect the location of the occupants, open doors, door bell rings etc. The robot can perform tasks such as checking and answering the doorbell, carrying drinks, providing reminders about medication etc. The high level robot behaviour is programmed using a set of if-then rules. As these rules are similar to the constructs in the Brahms language the scenario was modelled using Brahms and the Brahms to `Promela` tool applied. Several properties were checked relating to the robot following instructions from the person and reminders to take medication.

[20] develops an alternative tool for verifying Brahms models. This tools uses the Brahms semantics presented here implemented in full (i.e., including the transitions) in `Java`. A Brahms model for a multi-agent system is translated into `Java` and then executed in the Java Pathfinder (JPF) model checker to produce a *state model* of the Brahms program. This process of converting a Brahms model into state model is referred to as a *MAS connector*. Extensible plugins in JPF, such as customised choice points, allowed the efficient reduction of the state space. This state model represents all the possible states and actions of the MAS. Additionally the *MAS connector* can gather 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 reusable for different verifiers such as probabilistic and on-the-fly safety properties. The state model can then be converted into the input format of mainstream verification tools such as `Spin`, `NuSMV` and `Prism`, allowing verification of linear time temporal logic or branching-time temporal logic properties, probabilities, time bounds and cost. This work converts our static intermediate representation into an executable form and then uses JPF to produce a generic state model where we export the static representation to `Promela` code which then produces a state model specifically for `Spin`. As far as we are aware the tool developed in [20] is not publicly available for the verification of Human-Agent Teamwork models, hence why it was not used in work such as that reported in [35].

The verification of agents and multi-agent systems is an on-going research area, with papers such as [5, 2, 26, 14, 1, 38] developing methods for verification of various agent programming languages. We next discuss some examples model checking agent-based systems.

An approach to verifying multi-agent systems represented using one of a number of agent programming languages is via the Agent Infrastructure Layer (AIL) toolkit [14]. The AIL is a collection of Java classes developed to unify a variety of modelling formalisms, particularly agent programming languages. The collection of Java classes within the AIL contain clear and adaptable semantics and are able to implement interpreters for various agent languages. Programs interpreted by the AIL can then be model checked by Agent Java Pathfinder (AJPF); an extension of the Java Pathfinder model checker customised to support AIL-based interpreters. AJPF has been extended, based on [20] the so that it can be used to generate models for input to other (non-agent) model checkers such as `Spin` and `Prism` [13]. Thus AJPF can be used as a link between BDI-type programming languages and standard model checking tools. We did not use AIL in this work since it was not well developed when the work started, but our formalisation and implementation were guided by concepts using in the AIL. At present the AIL continues to focus on BDI-based programming languages rather than Human-agent teamwork modelling formalisms such as Brahms.

Bordini et al. [14] 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 GOAL [18] and SAAPL (Simple Abstract Agent Programming Language) [36].

De Boer et al. [2] produced a verification framework for goal orientated agents, specifically for agents programmed in the language GOAL. In [2] 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 [3] 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. [23] 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. [23] produce their own system for modelling agents using organisations, roles, actions, and obligations using Sicstus Prolog. To verify such models they assess the model with a constraint solver [22] which determines whether all the agents' obligations can be fulfilled.

9. Concluding Remarks

The work in this paper 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 aim to extend its usefulness by allowing all possible routes to be explored, thus ensuring that undesirable outcomes cannot arise within the model.

In this work we have developed the first formal operational semantics for Brahms 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` [19] or agent model checkers such as AJPF [14]. 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.

A scenario was used to demonstrate and analyse the verification produced by our tool. This scenario relates to a hospital example involving: two nurses, five 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 presented in [33] demonstrating the same key features of the Brahms semantics but also requiring the use of Brahms variables. The main difference is that this scenario uses more agents and requires more teamwork as agents and humans must perform tasks together.

9.1. Performance

The aim of this paper was to produce the first tool for the formal verification of human-agent-robot teamwork modelled in Brahms. However an important part of the evaluation of such a tool is the performance. Since this is the prototype, and was the first such tool, we have identified some aspects of the implementation which could be modified to reduce the size of the models and improve the verification speed. One such modification would be to re-implement the `Promela` translation so that all the agents and objects are represented in a single process. This modification would remove code which halts an agent while another executes. This halting of agents affects how code can be compressed into a smaller number of states using the `Promela` function called `d_step` which compresses multiple lines of code into a single state. Additionally, unnecessary states are created to implement the operational semantics as well as states for the scenario. For example, our tool will generate many states to check the guard conditions of a workframe, where new states are only needed if the workframe is active.

Despite some inefficiency, the tool we developed was still able to successfully verify all the properties in the Digital Nurse scenario and the Home Helper scenario in [33]. A performance comparison was made between our tool and the tool developed in [20], where the times taken to verify all the properties for the Home Helper scenario were compared. The latter utilised the Brahms semantics we developed and had the benefit of our experience when developing their tool. The implementation from [20] took 1 minute to verify all the properties, whereas our implementation took 5 minutes. The implementation in [20] uses Java Pathfinder which allows the user to define points of non-determinism in the code (such as selecting a workframe) and variables of interest (such as beliefs and facts). Java Pathfinder then generates a finite state machine creating states only when a defined point of non-determinism is encountered or a variable of interest changes value, which is then passed to a model checker to analyse. The generation of the finite state machine creates an overhead when using an *on the fly* model checker, such as `Spin`, meaning the model will need to be created and then explored, where an on the fly model checker explores a model as it is being created.

We believe the future improvements to our tool suggested earlier to reduce the state space as well as this disadvantage of the JPF tool would make our tool comparable if not better than the JPF based tool.

9.2. Wider Application

In this paper we used our tool to formally verify properties of a Digital Nurse scenario. The tool was also used in [33] to verify properties of a Home Helper scenario. Although our tool was developed to verify properties of human-agent-robot teamwork models, it is not restricted to these scenarios. Our tool can translate the majority of the functions implemented in the Brahms language, including all the key functions such as workframes, throughtframes, variables, detectables, locations etc. allowing, our tool to be used to verify most Brahms simulations.

There are limitations to the size of programs that our tool can practically verify. Large models and models with multiple highly active agents tend to contain too many states for feasible model checking. It is difficult to define the limit on the scenarios we can verify, because multiple factors can combine to cause state space explosion during verification. The main contributors are the number of agents, the number of attributes the agents have, the duration of activities, the length of the simulation, and the number of non-deterministic events. The Digital Nurse scenario we produced was at the limit of the size of model we can currently verify.

9.3. Further Development

The tool created in this work for the verification of 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. Some ways of increasing efficiency includes:

1. identifying a communication issue which results in an exponential rise in states when performing multiple communications; and
2. structuring the Brahms translation for efficient use of deterministic wrappers to limit the number of surplus states, particularly by reducing the number of processes.

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).

The functionality of the tool's user interface could also be improved as currently it is only executable from the command line. `Spin` never-claims (used to express specifications) 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 single 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.

Finally, a more formalised justification of the correctness argument sketched in Section 6 might be possible, though this would certainly take considerable work and would still require appeal to unformalised elements throughout.

Acknowledgements

The research reported in this paper was partially funded by EPSRC grants EP/K006193 (*Trustworthy Robotic Assistants*), EP/J011770 (*Reconfigurable Autonomy*), and EP/F033567 (*Verifying Interoperability Requirements in Pervasive Systems*). We would like to thank researchers at NASA who have commented on the developed Brahms semantics and researchers at PARC who provided input to the Digital Nurse Scenario.

References

[1] G. Ali, S. Khan, N. Zafar, and F. Ahmad. Formal Modeling Towards a Dynamic Organization of Multi-agent Systems using Communicating X-machine and Z-notation. *Indian Journal of Science and Technology*, 5(7):2972–2977, 2012.

[2] F. de Boer, K. Hindriks, W. van der Hoek, and J-J. Meyer. A Verification Framework for Agent Programming with Declarative Goals. *Journal of Applied Logic*, 5(2):277–302, 2007.

[3] R. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model Checking Agentspeak. In *Proc. 2nd Int. Conf. Autonomous Agents and Multiagent Systems (AAMAS)*, pp409–416. ACM, 2003.

[4] R. Bordini, M. Fisher, and M. Sierhuis. Analysing Human-agent Teamwork. In *Proc. 10th ESA Workshop on Advanced Space Technologies for Robotics and Automation (ASTRA)*, Noordwijk, The Netherlands, 2008.

[5] R. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model Checking Rational Agents. In *IEEE Intelligent Systems*, volume 19, pp46–52. IEEE, 2004.

[6] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley, 2007.

[7] J. Bradshaw, P. Feltovich, M. Johnson, M. Breedy, L. Bunch, T. Eskridge, H. Jung, J. Lott, A. Uszok, and J. van Diggelen. From Tools to Teammates: Joint Activity in Human-Agent-Robot Teams. In *Proc. HCI (10)*, volume 5619 of *LNCS*, pp935–944. Springer, 2009.

[8] L. Cavedon, A. Rao, L. Sonenberg, and G. Tidhar. Teamwork via Team Plans in Intelligent Autonomous Agent Systems. In *World Wide Computing and its Applications*, volume 1274 of *LNCS*, pp106–121. Springer, 1997.

[9] W. Clancey, M. Sierhuis, C. Kaskiris, and R. van Hoof. Advantages of Brahms for Specifying and Implementing a Multiagent Human-Robotic Exploration System. In *Proc. 16th Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pp7–11. AAAI Press, 2003.

[10] 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.

[11] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[12] P. Cohen and H. Levesque. Teamwork. In *Noús*, 25(4):487–512. *Special Issue on Cognitive Science and Artificial Intelligence*. Blackwell Publishing, 1991.

[13] L. Dennis, M. Fisher, and M. Webster. Two-Stage Agent Program Verification. To appear in *Journal of Logic and Computation*, 2015.

[14] L. Dennis, M. Fisher, M. Webster, and R. H. Bordini. Model Checking Agent Programming Languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[15] J. Dyer. Team Research and Team Training: A State-of-the-Art Review. In *Human factors review*, volume 1983, pp285–3. Human Factors Society, 1984.

[16] E. Emerson. The Role of Buchi's Automata in Computing Science. In *The Collected Works of J. Richard Büchi*. Springer, 1990.

[17] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly Automatic Verifi-

cation of Linear Temporal Logic. In *Proc. 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*. IFIP, 1995.

[18] GOAL — Website. `http://mmi.tudelft.nl/trac/goal`.

[19] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[20] J. Hunter, F. Raimondi, N. Rungta, and R. Stocker. A Synergistic and Extensible Framework for Multi-Agent System Verification. In *Proc. Int. Conf. Autonomous Agents and Multi-Agent Systems (AAMAS)*, pp869–876. IFAAMAS, 2013.

[21] 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)*, pp130–135. IEEE Robotic and Automation Society, 2008.

[22] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

[23] M. McCallum., W. Vasconcelos, and T. Norman. Verification and Analysis of Organisational Change. In *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, pp48–63. Springer, 2006.

[24] M. Montemerlo, J. Pineau, N. Roy, S. Thrun, and V. Verma. Experiences with a Mobile Robotic Guide for the Elderly. In *Proc. 18th National Conference on Artificial Intelligence (AAAI)*, pp587–592. AAAI Press, 2002.

[25] J. Pineau, M. Montemerlo, M. Pollack, N. Roy, and S. Thrun. Towards Robotic Assistants in Nursing Homes: Challenges and Results. *Robotics and Autonomous Systems*, 42(3-4):271–281, 2003.

[26] M. van Riemsdijk, F. de Boer, M. Dastani, and J-J. Meyer. Prototyping 3APL in the Maude Term Rewriting Language. In *Proc. Workshop on Computational Logic in Multi-Agent Systems*, pp95–114. Springer, 2007.

[27] E. Salas, D. Sims, and C. Burke. Is there a “Big Five” in Teamwork? *Small Group Research*, 36(5):555–599, 2005.

[28] 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.

[29] M. Sierhuis. Multiagent Modeling and Simulation in Human-Robot Mission Operations. (`http://ti.arc.nasa.gov/pub-archive/2006`), 2006.

[30] M. Sierhuis, J. Bradshaw, J. Acquisti, R. van Hoof, R. Jeffers, and A. Uszok. Human-agent Teamwork and Adjustable Autonomy in Practice. In *Proc. 7th Int. Symp. Artificial Intelligence, Robotics and Automation in Space*, 2003.

[31] M. Sierhuis, W. Clancey, R. van Hoof, C. Seah, M. Scott, R. Nado, S. Blumenberg, M. Shafto, B. Anderson, A. Bruins, C. Buckley, T. Diegelman, T. Hall, D. Hood, F. Reynolds, J. Toschlog, and T. Tucker. NASA’s OCA Mirroring System — An Application of Multiagent Systems in Mission Control. In *Proc. Int. Conf. Autonomous Agents and Multi Agent Systems (AAMAS)*, 2009.

[32] R. Stocker. *Towards the Formal Verification of Human-Agent-Robot Teamwork*. PhD thesis, Department of Computer Science, University of Liverpool, 2013.

[33] R. Stocker, L. Dennis, C. Dixon, and M. Fisher. Verifying Brahms Human-Robot Teamwork Models. In *Proc. European Conf. Logics in Artificial Intelligence*

(JELIA), volume 7519 of *LNCS*, pp385–397. Springer, 2012.

[34] R. Stocker, M. Sierhuis, L. Dennis, C. Dixon, and M. Fisher. A Formal Semantics for Brahms. In *Proc. 12th Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*, volume 6814 of *LNCS*, pp259–274. Springer, 2011.

[35] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. Koay, and K. Dautenhahn. Formal Verification of Robotic Assistants for Home-Healthcare Environments. In *Proc. Workshop on Formal Verification in Human Machine Systems (FVHMS)*. AAAI, 2014.

[36] M. Winikoff. Implementing Commitment-based Interactions. In *Proc. Int. Conf. Autonomous Agents and Multiagent Systems*, pp868–875. ACM, 2007.

[37] M. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2009.

[38] W. Yeung. Behavioral Modeling and Verification of Multi-agent Systems for Manufacturing Control. *Expert Systems with Applications*, 38(11), 2011.

Appendix A. Semantic Rules

Appendix A.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. ξ represents the scheduler, showing that all the agents and the scheduler move as one from time point to time point.

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

Appendix A.2. 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.

RULE: Sch_run

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

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

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

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_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.

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_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.

RULE: Sch_Term

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

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

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

Appendix A.3. Agent Semantics

The Brahms system operates on a simple cycle of handling:

$$Thoughtframes \rightarrow Detectables \rightarrow Workframes$$

In the following we present groups of rules relating to Set_* rules thoughtframes (Tf_* rules) workframes (Wf_* rules) detectables (Det_* rules) variables (Var_* rules) popstack (Pop_* rules)

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.

RULE: 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.

RULE: Set_Idle

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

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

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

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 are any thoughtframes with preconditions met then one is selected based on the thoughtframe's priority using the Max_pri method. The agent can not have a current thoughtframe but might have an active workframe. 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).

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

RULE: 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.

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

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

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

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.

<u>RULE:</u> Wf_Select

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

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 β_d represents the workframe's deed stack and β_{ins} refers to the workframe's instructions, such as the workframe's repeat values, etc.

<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_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_*'

<u>RULE:</u> Wf_true

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

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.

<u>RULE:</u> Wf_once

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

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.

<u>RULE:</u> Wf_false

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

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 continues.

<u>RULE:</u> 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, β^D is the workframe β'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.

<u>RULE:</u> 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 β.

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).

<u>RULE:</u> 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.

<u>RULE:</u> 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_*'.

<u>RULE:</u> Det_empty

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

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

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

Here β^V represents the variables contained within workframe β.

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 β with a deed stack d, a set of empty workframe instances and the workframe's set of instructions β_{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

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

'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.

<u>RULE:</u> Var_each

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

Var_all. The 'collectall' variable 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.

<u>RULE:</u> Var_all

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

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

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'.

RULE: Pop_WfconcF

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

$$\xrightarrow[f\in F\wedge\beta=\langle\beta_d, conclude(f')^{fact};\beta_{ins}\rangle]{F=(F/f)\cup f'}$$

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

RULE: 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.

RULE: 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.

RULE: Pop_notConc

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

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

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

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. The agents use this rule 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)

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

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

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

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). 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

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