A Formal Semantics for Brahms*

Richard Stocker[1], Maarten Sierhuis[2,3], Louise Dennis[1], Clare Dixon[1], Michael Fisher[1]

[1] Department of Computer Science, University of Liverpool, UK
[2] PARC, Palo Alto, USA
[3] Man-Machine Interaction, Delft University of Technology, Delft, NL
Contact: Richard Stocker (R.S.Stocker@liverpool.ac.uk)

Abstract. The formal analysis of computational processes is by now a well-established field. However, in practical scenarios, the problem of how we can formally verify interactions with humans still remains. In this paper we are concerned with addressing this problem. Our overall goal is to provide formal verification techniques for human-agent teamwork, particularly astronaut-robot teamwork on future space missions and human-robot interactions in health-care scenarios. However, in order to carry out our formal verification, we must first have some formal basis for this activity. In this paper we provide this by detailing a formal operational semantics for Brahms, a modelling/simulation framework for human-agent teamwork that has been developed and extensively used within NASA. This provides a first, but important, step towards our overall goal by establishing a formal basis for describing human-agent teamwork, which can then lead on to verification techniques.

1 Introduction

Computational devices often need to interact with humans. These devices can range from mobile phones or domestic appliances, all the way to fully autonomous robots. In many cases all that the users care about is that the device works well most of the time. However, in mission critical scenarios we clearly require a more formal, and consequently much deeper, analysis. Specifically, as various space agencies plan missions to the Moon and Mars which involve robots and astronauts collaborating, then we surely need some form of *formal verification* for astronaut-robot teamwork. This is needed at least for astronaut safety (e.g. "the astronaut will never be out of contact with the base") but also for mission targets (e.g. "three robots and two astronauts can together build the shelter within one hour"). But: how are we to go about this? How can we possibly verify human behaviour? And how can we analyze teamwork?

In [2] a formal approach to the problem of human-agent (and therefore astronaut-robot) analysis has been proposed, suggesting the model-checking of *Brahms* models [10, 18, 14]. Brahms is a simulation/modelling language in which complex human-agent work patterns can be described. Importantly for our purposes, Brahms is based on the concept of *rational agents* and the system continues to be successfully used

* Work partially funded in the UK through EPSRC grants EP/F033567 and EP/F037201.

within NASA for the sophisticated modelling of astronaut-robot planetary exploration teams [4, 13, 11].

Thus, it seems natural to want to formally verify Brahms models [2] but, until now, the Brahms language had no *formal* semantics (unless you count the implementation code as this). So, this paper describes the first formal operational semantics [16] for the Brahms language; in current work we are using the formal semantics to develop and apply model checking to Brahms.

2 Brahms

Brahms is a multi-agent modelling, simulation and development environment devised by Sierhuis [10] and subsequently developed at NASA Ames Research Center. Brahms is a modelling language designed to model human activity using *rational agents*.

An agent [19] essentially captures the idea of an autonomous entity, being able to make its own choices and carry out its own actions. Beyond simple autonomy, *rational agents* are increasingly used as a high-level abstraction/metaphor for building complex/autonomous space systems [6]. Rational agents can be seen as agents that make their decisions in a *rational* and *explainable* way (rather than, for example, purely randomly). The central aspect of the rational agent metaphor is that such agents are autonomous, but can react to changes in circumstance and can choose what to do based on their own agenda. In assessing such systems it may not be sufficient to consider *what* the agent will do, but we must often also consider *why* it chooses to do it. The predominant view of rational agents is that provided by the BDI (beliefs-desires-intentions) model [9, 8] in which we describe the goals the agent has and the choices it makes. Thus, in modelling a system in terms of rational agents, we typically describe each agent's *beliefs* and *goals* (*desires*), which in turn determine the agent's *intentions*.

Brahms follows a similar rational agent approach but, because it was developed in order to represent people's activities in real-world contexts, it also allows the representation of artifacts, data, and concepts in the form of classes and objects. Both agents and objects can be located in a model of the world (the geography model) giving agents the ability to detect objects and other agents in the world and have beliefs about the objects. Agents can move from one location in the world to another by executing a *move* activity, simulating the movement of people. For a more detailed description of the Brahms language we refer the reader to [10] and [11]. The *key aspects* of Brahms are:

- *activities*: actions an agent can perform, which typically consume simulation time;
- *facts*: state of the environment (which every agent/object can observe through the use of "detectables");
- *beliefs*: each agent's own personal perceptions;
- *detectables*: bring facts into the an agent's belief base and determine how the agent will react in response;
- *workframes*: sequences of events required to complete a task, together with any belief updates resulting from the task completion;
- *thoughtframes*: reasoning/thought processes, e.g. "I believe it is raining therefore I believe I need an umbrella";

– *time*: central to Brahms as the output is represented in the form of a time-line displaying every belief change and event that occurs.

In summary, the Brahms language was originally devised to model the contextual situated activity behaviour of groups of people. It has now evolved into a language for modelling both people *and* robots/agents. As such it is ideal for describing human-agent/robot teamwork.

2.1 Brahms Example

Orbital Communications Adaptor (OCA) officer flight controllers in NASA's International Space Station Mission Control Center use different computer systems to uplink, downlink, mirror, archive, and deliver files to and from the International Space Station (ISS) in real time. The OCA Mirroring System (OCAMS) is a multi-agent software system operational in NASA's Mission Control Center [15], replacing the OCA officer flight controller with an agent system that is based on the behavior of the human operator. NASA researchers developed a detailed human-behavioral agent model of the OCA officers' work practice behaviour in Brahms. The agent model was based on work practice observations of the OCA officers and the observed decision-making involved with the current way of doing the work. In the system design and implementation phases, this model of the human work practice behaviour was made part of the OCAMS multi-agent system, enabling the system to behave and make decisions as if it were an OCA officer. Here is a short scenario of how the OCAMS system is used in mission control:

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

The OCAMS system has been extended over three years [5]. With the latest release the OCAMS system will have completely taken over all routine tasks from the OCA officer, about 80% of the workload. Other flight controllers in mission control will interact with the OCAMS agent as if it were an OCA officer. With every new release (indeed with every increase in functionality) the system developers are required to perform complete testing of the system. Increases in functionality mean that there is now not enough time to test every possible case. The ability to carry out formal verification and validation of this human-agent system would enable more comprehensive analysis.

3 Overview of Semantics

Rather than presenting the full semantics in detail (see [17]), we consider the core elements of the semantics here and then work through an example Brahms scenario in Section 4.

3.1 Time keeping and Scheduling

An important aspect of Brahms is a shared system clock; this creates the simulation time line and is used as a global arbiter of when activities start and end. Agents are not explicitly aware of the system clock even though it controls the duration of activities and can be referred to in the selection of workframes and thoughtframes. As a result, many applications also involve a clock "object" that agents are explicitly aware of.

The Brahms execution model involves updating the system clock, then examining each agent in turn to see what internal state changes take place at that time step, and then updating the system clock once more. The system clock does not update by a fixed amount in each cycle but makes a judgment about the next time something "interesting" is going to happen in the system and jumps forward to that point.

3.2 Running Workframes and Thoughtframes

Workframes and *thoughtframes* represent the plans and thought processes in Brahms. A workframe contains a sequence of activities and belief/fact updates which the agent/object will perform and a guard which determines whether the workframe is applicable or not. A thoughtframe is a restricted version of this which only contains sequences of belief/fact updates. Workframes may take time to complete (depending on the activities involved) while thoughtframes are assumed to run instantaneously. A workframe can also detect changes in its environment (facts), bring these changes into the agent's belief base and then decide whether or not to continue executing in the light of the changes. Essentially, workframes represent the work processes involved in completing a task and thoughtframes represent the reasoning process upon the current beliefs, e.g. "I perform a workframe to go the shops; on leaving the house I detect it is raining so I suspend my workframe and update my belief that it is raining, which then triggers a thoughtframe stating that, since it is raining and I want to go the shops, then I need a raincoat".

3.3 Priority and Suspension of Workframes and Thoughtframes

Workframes and thoughtframes have a priority (which can be assigned by the programmer or derived from their list of activities). Thoughtframe and workframe priorities are independent, but an agent will execute all thoughtframes first in any given time step before moving on to examine workframes. At any point in time, the workframe that an agent is currently working on can be suspended if another, higher priority, workframe becomes available. Thoughtframes are *never* suspended because they have no duration and so always complete before any higher priority thoughtframe becomes available. When several workframes have the same priority, Brahms considers the currently executing workframe to have the highest priority, any other suspended workframes have second highest, impassed[1] workframes will have third highest, and then any other workframe will follow. Priorities in Brahms are integers but to model this priority order in our semantics we assign suspended workframes an increased priority of "0.2" and impassed workframes "0.1". Workframes and thoughtframes of equal status and with joint

[1] Workframes suspended because of changes in detectable facts.

highest priority will be selected at random. When a workframe is suspended, everything is stored, even the duration into its current activity, so the agent can resume exactly from where it left off.

3.4 Executing plans: Activities and Communication

When an agent is assigned a workframe/thoughtframe all the instructions contained within it are stored on one of two stacks; one for the current thoughtframe and one for the current workframe. When an agent executes an instruction it is 'popped' off the top of the stack. *Primitive* activities and *move* activities all have time associated with them — when they are at the top of a stack the duration of the task is decreased by an appropriate amount each time the system clock updates. If the duration remaining reaches zero then the activity is finished and popped off the stack. When the activity is a *move* activity the belief base of the agent, together with the global facts, are changed to reflect the new position of the agent/object; this update occurs when the time of the activity reaches zero.

Communications are also activities and may have a duration. When the communication ends, a belief update is performed on the target agent's (the receiver's) belief set.

3.5 Detectables

Detectables are contained within workframes and can only be executed if the workframe is currently active. They detect changes in facts then either: abort, impasse, continue or complete the workframe. Detected facts are imported into the agent's belief base and then either: *aborts* - deletes all elements from the workframe's stack; *impasses* - suspends the current workframe, *continues* - carries on regardless; or *completes* - deletes only activities from the workframe's stack but allows it to make all (instantaneous) belief updates.

3.6 Variables

Variables provide a method of quantification within Brahms. If there are multiple objects or agents which can match the specifications in a work- or thoughtframe's guard condition then the variable can either perform: *forone* — just select one; *foreach* — work on all, one after another; or *collectall* — work on all simultaneously. This is handled by recursive sets of workframes, e.g.

$$\text{“Set of Workframes”} = \{\text{W1,W2,W3:}\{\text{W3.1,W3.2}\}\text{,W4}\}$$

where W3 is a workframe with variables and W3.1 and W3.2 are instantiations of W3 but with objects/agents in the place of the variables. When a workframe with variables is empty e.g. W3:{} Brahms will invoke a selection function to make instantiations based on the conditions.

3.7 Brahms Syntax

Brahms has a complex syntax for creating systems, agents and objects, although there is no space here to cover the full specification [2]. As an example Fig. 1 shows the definition of an agent's workframe showing where variables, detectables and the main body of the workframe are placed. Guards are specified by `precondition-decl`.

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

Fig. 1. Example of Brahms Syntax Specification

3.8 Semantics: Notation

In the rest of this paper we use the following conventions to refer to components of the system, and agent and object states.

Agents: ag represents one *agent*, while Ag represents the *set* of all agents.
Beliefs: b represents one *belief*, while B represents a *set* of beliefs. In Brahms the overall system may have beliefs which are represented by B_ξ.
Facts: f represents one *fact*, while F represents a *set* of facts.
Workframes: β represents the *current workframe* being executed, WF represents a *set* of workframes, while *W* is any arbitrary workframe.
Thoughtframes: $\mathcal{T}$ represents any arbitrary *thoughtframe*, while TF represents a *set* of thoughtframes.
Activities: $\texttt{Prim\_Act}^t$ is a primitive activity of duration t.
Environment: ξ represents the *environment*
Time: T represents the time in general, while a specific duration for an activity is represented by t. The time maintained by the system clock is T_ξ.
Stage: The semantics are organised into "stages". Stages refer to the names of the operational semantic rules that may be applicable at that time, wild cards ($*$) are used to refer to multiple rules with identical prefixes. There is also a "*fin*" stage which indicates an agent/object is ready for the next cycle, and an "$idle$" stage which means it currently has no applicable thoughtframes or workframes.

[2] For the full syntax (with an informal semantics) see [12]

Scripts Superscripts are used in these semantics to extract elements of a structure and subscripts are used to identify the owner of the component, e.g. ag_i^{stage} would refer to the "stage" of the agent i.

Since the data structures for workframes are fairly complex we will treat these as a tuple, $\langle W_d, W_{ins} \rangle$ where W_d is *workframe header data* and W_{ins} is the *workframe instruction stack*. The *workframe header data* includes

- W^r is the workframe's repeat variable.
- W^{pri} is the workframe's priority.
- W^V is the variable declaration.
- W^D is the workframe's detectables
- W^g is the workframe's guard.

Here we are considering any possible workframe (W), for the current workframe we would use β (or the name of the workframe). Thoughtframes are structured in a similar way.

3.9 Semantics: Structure

The system configuration is a 5-tuple description: the first element of the tuple is the set of all agents; the second is the current agent under consideration; the third is the belief base of the system, used to synchronise the agents (not used is simulations) e.g. agent i's next event finishes in 1000 seconds; the fourth is the set of facts in the environment, e.g. temperature is 20 degrees celsius; and the fifth is the current time of the system:

System's tuple = $\langle Ags, ag_i, B_\xi, F, T_\xi \rangle$

The agents and objects within a system have a 9-tuple representation: the first is the identification of the agent; second is the current thoughtframe; third is the current workframe; fourth is the stage the agent is at; fifth is the set of beliefs the agent has; sixth is the set of facts of the world; seventh is the time of the agent; eighth is the set of thoughtframes the agent has; and the ninth is the agent's set of workframes. The fourth element of the tuple, the stage, explains which set of rules the agent is currently considering or if the agent is in a finish (*fin*) or idle ($idle$) stage.

Agent's tuple = $\langle ag_i, \mathcal{T}, W, stage, B, F, T, TF, WF \rangle$

The semantics are represented as a set of transition rules of the form

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

Here, '$ConditionsRequiredForActions$' refers to any conditions which must hold before the rule can be applied. '$ActionsPerformed$' is used to represent change to the agent, object or system state which, for presentational reasons, can not be easily represented in the tuple.

Finally, it is assumed that all agents and objects can see and access everything in the environment's tuple, e.g. T_ξ.

4 Running Example of Brahms Semantics

As explained above, rather than giving all the semantics in detail (see [17] for the full semantics), we here work through a small Brahms scenario. Though simple, this scenario involves many of the aspects available within Brahms and so utilises a wide variety of semantic rules.

This example scenario is based on that provided in the Brahms tutorial which can be downloaded from `http://www.agentisolutions.com/download`. The scenario models two students: *Alex*, who will sit and study in the library until he becomes hungry; and *Bob* who sits idly until the other student suggests going for food. When Alex becomes hungry he will decide to message Bob and venture out for food. Once Alex arrives at the restaurant he will wait for Bob to arrive and then he will eat, pay for the food and return to the library to study. The scenario also contains an explicit hourly clock object (separate from the system clock) which announces how many hours have passed, providing Alex with his beliefs about the current time.

Initialisation and parsing of the program code assigns the agents all their initial beliefs, determines the initial world facts and the geography of the area (distances between each location etc.). The agent Bob will be ignored in the discussion until his role becomes active. The initial beliefs of our student (Alex) are:

$$\left\{ \begin{array}{l} hungry = 15, \quad Loc = Library, \quad Bob.Loc = Home, \\ perceivedtime = 0, \quad Clock.time = 0, \quad desiredRestaurant = \emptyset \end{array} \right\} \tag{1}$$

Alex starts in the "*fin*" (finish) stage. Starting in the finished stage appears counterintuitive, however this "*fin*" indicates the agents have finished their previous events and are ready for the next cycle. On system initiaition we assume the agents previous events have been completed, even though they were empty.

Below is example Brahms code showing one of Alex's thoughtframes and one of his workframes. The thoughtframe represents Alex's thought process for increasing his hunger as time progresses (Campanile_Clock refers to the clock object) and Alex becomes hungrier when he sees that the clock object's time is later than he believes it is. He then updates his internal belief about the time and his hunger. The workframe tells Alex to perform the study activity when both the time and his level of hunger are less than 20.

```
agent Alex
{
  ...
  thoughtframes:
    thoughframe tf_FeelHungry
    {
       when(knownval(Campanile_Clock.time >
            current.perceivedtime)
        do
        {
            conclude((current.perceivedtime =
                      Campanile_Clock.time), bc: 100);
```

```
            conclude((current.hungry =
                      current.hungry + 3), bc: 100);
        }
    }

  workframes:
    workframe wf_Study
    {
        repeat: true;
        priority: 1;
        when(knownval(Campanile_Clock.time < 20)
             and current.hungry < 20)
        do
        {
            Study();
        }
    }
}
```

Here, "`bc: 100`" describes a percentage probability value associated with the belief. For simplicity we omit further discussion of this in the remainder of the paper.

In later representations of thoughtframes and workframes we will use in the semantics, the above tf_FeelHungry will appear as

$$\langle\ FeelHungry_d,\\ [conclude(perceivedtime = Clock.time);\ conclude(hungry = hungry + 3)]\rangle$$

where the thoughtframe data $FeelHungry_d$ contains the guard, $FeelHungry^g$ with the value $Clock.time > perceivedtime$. Similarly, the `wf_Study` thoughtframe above will later appear as $\langle Study_d, [\texttt{Prim\_Act}^{3000}]\rangle$. Note that `Study()` is a primitive activity with duration 3000 seconds and we translate it directly into the primitive activity representation in the workframe's instruction stack. $Study_d$ includes the repeat variable $Study^r = true$, the priority $Study^{pri} = 1$ and the guard $Study^g = (Clock.time < 20) \wedge (hungry < 20)$

The **thoughtframes** Alex uses are:

$tf\_FeelHungry$ - increases Alex's hunger;
$tf\_ChooseBlakes$ - tells Alex to choose Blakes restaurant;
$tf\_ChooseRaleighs$ - tells Alex to choose Raleighs restaurant.

The **workframes** Alex uses are:

wf_Study - tells Alex to study;
$wf\_MoveToRestaurant$ - tells Alex to move to his desired restaurant;
wf_Wait - tells Alex to do nothing if he is in the restaurant and Bob isn't present;
wf_Eat - tells Alex to order and eat his food.

The **workframes** the Clock uses are:

$wf\_AsTimeGoesBy$ - increases the clock's time by 1 hour.

4.1 System Initiation

Initially the tuples for the Alex agent and the Clock object are

$$\begin{array}{ll} \text{Alex:} & \langle ag_{Alex}, \emptyset, \emptyset, \mathit{fin}, B_{Alex}, F, 0, TF_{Alex}, WF_{Alex} \rangle \\ \text{Clock:} & \langle ob_{Clock}, \emptyset, \emptyset, \mathit{fin}, B_{Clock}, F, 0, \emptyset, WF_{Clock} \rangle \end{array}$$

where B_{Alex} is as in (1) and

$$\begin{aligned} TF_{Alex} &= \{tf\_FeelHungry, tf\_ChooseBlakes, tf\_ChooseRaleighs\} \\ WF_{Alex} &= \{wf\_Study, wf\_MoveToRestaurant, wf\_Wait, wf\_Eat\} \\ WF_{Clock} &= \{wf\_AsTimeGoesBy\} \\ B_{Clock} &= \{time = 0\} \\ F &= \{Alex.Loc = library, Bob.Loc = Home, Clock.time = 0\} \end{aligned}$$

4.2 Scheduler Rules

All the semantic rules used by the scheduler have the prefix 'Sch_'. The scheduler acts as a mediator between agents keeping them synchronized. It tells all the agents when the system has started, finished and when to move to the next part of the system cycle.

The scheduler is initiated first in any run of the system. It checks if all agents' current stage is either "active" or "finished". Since, in our example, it is the beginning of the system and the default setting for the agents' stage is finished then the system updates the stage of each agent to Set_Act. This will cause all the agents to start processing. This system action is expressed with the Sch_run rule.

RULE: Sch_run

$$\langle Ags, ag_i, B_\xi, F, T_\xi \rangle \xrightarrow[\forall ag_i \in Ags | a_i^{stage} = \mathit{fin} \cup idle, (T_\xi \neq -1)]{ag_i'^{stage} = Set\_Act} \langle Ags, ag_i', B_\xi, F, T_\xi \rangle$$

So, after this rule is executed Alex's state becomes:

$$\langle ag_{Alex}, \emptyset, \emptyset, Set\_Act, B_{Alex}, F, 0, TF_{Alex}, WF_{Alex} \rangle$$

While there is an agent in an active (not idle or finished) stage the scheduler waits. If all the agents are idle the system terminates:

RULE: Sch_Term

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

Agents and objects have their own internal clock but the scheduler manages the global clock which all agents/objects synchronize with. When agents or objects perform an activity they inform the scheduler of the time the activity will conclude. Once all agents are either idle or engaged in an activity (a set of stages marked Pop_CA*, Pop_MA* and Pop_CA* where "$*$" is a wild card) the scheduler then finds the smallest of all these

times and updates the global clock to this time. This is achieved by the 'Sch_rcvd' rule:

RULE: Sch_rcvd

$$\langle Ags, ag_i, B_\xi, F, T_\xi \rangle \xrightarrow[\forall ag_i \in Ags | stage = Pop\_(PA*/MA*/CA*) \cup idle, (T_\xi \neq -1)]{T'_\xi = T_\xi + MinTime(B_\xi(\forall ag_i(T_i)))} \langle Ags, ag_i, B_\xi, F, T'_\xi \rangle$$

4.3 Agents and objects are now invoked

The agents and objects are invoked in order and each processes one rule in turn. In the Set_Act stage the agents run the Set_Act rule which, in this case, moves them all on to examining their thoughtframes for applicability. This is the 'Tf_*' stage which indicates that they will be looking at all the rules beginning with 'Tf_'. These are rules for processing thoughtframes. In our simple example there are currently no thoughtframes available for any objects or agents, so 'Tf_Exit' is selected, which moves Alex and the Clock on to checking for detectables.

In our example, no object or agent has a current workframe which checks detectables and so 'Det_Empty' is invoked which passes them on to checking workframes. This is the basic cycle of Brahms processing: thoughtframes, followed by detectables, followed by workframes.

Running workframes. $wf\_AsTimeGoesBy$ is the only workframe the Clock can process. This contains an activity of 3600 seconds duration and then the Clock increases its time by 1 hour. The guard on this workframe is that the current value of the Clock's time attribute is less than 20, which is currently true so the workframe is put forward for selection. Since there are no other workframes, the Clock selects this workframe as current and stores the list of instructions contained within the workframe in a stack.

Meanwhile Alex also selects a workframe for execution using Wf_Select.

RULE: Wf_Select

$$\langle ag_i, \emptyset, \emptyset, Wf\_*, B_i, F, T_i, TF_i, WF_i \rangle \xrightarrow[\exists W \in WF_i | B_i \models W^g]{\beta = Max_{pri}(W \in WF_i | B \models W^g)} \langle ag_i, \emptyset, \beta, Wf\_(true/false/once), B_i, F, T_i, TF_i, WF_i \rangle$$

This selects Alex's workframe (wf_Study) and restricts his rule choice to 'Wf_true', 'Wf_false' or 'Wf_once', depending on the workframe's repeat variable. The body of the workframe contains a single primitive activity, $Study$.

Alex's state is now

$$\langle\ ag_{Alex},\ \emptyset,\ \langle Study_d,\ [\texttt{Prim\_Act}^{3000}]\rangle,\ Wf\_(true/false/once),\ B_{Alex},\ F,\ 0,\ TF_{Alex},\ WF_{Alex}\rangle$$

The next semantic rule selected depends on the repeat variable of the current workframe. Here, wf_Study has repeat set to "true" (always repeat) so rule '**Wf_True**' is applicable. This does nothing more than pass the agent to the next set of rules used for handling variables (denoted Var_*). There are two other repeat rules: '**Wf_False**' (never repeat) means the workframe would be deleted from the set of workframes when finished; and '**Wf_Once**' (repeat only one more time) sets the repeat variable in the workframe to false from this point onward.

Both the Clock and Alex now move to pop elements off the stack associated with the current workframe. These rules are denoted with '**Pop_***'.

Popping the stack. Both the Clock object and our Alex agent are now processing the elements on their current workframe's stack of activities, using the rules denoted by '**Pop_**'. wf_Study tells Alex to perform a primitive activity (essentially a wait) called $Study$ with duration 3000 seconds. Meanwhile the Clock has an activity to wait 3600 seconds before updating the time. For convenience we will simply show the current workframe's instruction stack in the following tuples, not the full workframe.

The current states of Alex and the Clock are:

$$\langle ag_{Alex}, \emptyset, \langle Study_d, [\texttt{Prim\_Act}^{3000}]\rangle, Pop\_*, B_{Alex}, F, 0, TF_{Alex}, WF_{Alex}\rangle$$

$$\begin{array}{l}\langle\ ob_{Clock}, \emptyset, \langle AsTimeGoesBy_d, [\texttt{Prim\_Act}^{3600}]\rangle, Pop\_*,\\ \ B_{Clock}, F, 0, TF_{Clock}, WF_{Clock}\rangle\end{array}$$

The Clock and Alex communicate the duration of their current activity to the scheduler by updating the system's beliefs about the time they are due to finish their next event. This is done using the rule '**Pop_PASend**'.

RULE: **Pop_PASend**

$$\langle ag_i, \emptyset, \langle \beta_d, [\texttt{Prim\_Act}^t; \beta_{ins}]\rangle, Pop\_*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[T_\xi = T_i]{B'_\xi = B_\xi \cup (T_i = T_i + t)}$$
$$\langle ag_i, \emptyset, \langle \beta_d, [\texttt{Prim\_Act}^t; \beta_{ins}]\rangle, Pop\_PA*, B_i, F, T_i, TF_i, WF_i\rangle$$

(Recall that individual agents can still act upon the main system state, e.g. T_ξ and B_ξ here.) Once all the agents have communicated their duration (excluding idle agents) the scheduler compares their durations to find the shortest activity and updates its internal clock using '**Sch_rcvd**'. In this case it updates the time from 0 to 3000 which is when Alex's activity will finish.

Both Alex's and the Clock's time remain at 0 but the global clock is now at 3000. The '**Pop_PA***' rules all have a time difference as a guard and act to decrease the remaining duration of the current activity and update the agent/object's internal time keeping.

In this situation the Clock object's primitive activity duration is decreased to 600. Alex's activity will have finished (since it is 3000) and a different rule, '**Pop_PA(t=0)**' is invoked:

RULE: Pop_PA(t=0)

$$\langle ag_i, \emptyset, \langle \beta_d, [\texttt{Prim\_Act}^t; \beta_{ins}]\rangle, Pop\_PA*, B_i, F, T_i, TF_i, WF_i\rangle$$
$$\xrightarrow[\neg(T_\xi = T_i), T_i + t - T_\xi = 0]{T_i' = T_\xi}$$
$$\langle ag_i, \emptyset, \langle \beta_d, \beta_{ins}\rangle, Pop\_concWf*, B, F, T_i', TF_i, WF_i\rangle$$

Alex has now moved on to a stage where he will only perform 'conclude' actions in a workframe stack (denoted by stage 'Pop_concWf*'). There are no conclude actions on the stack so he is transferred to the workframe rules 'Wf_*' once more.

4.4 The Cycle Continues

The simulation continues to run and the Clock's time attribute is updated when its primitive activity finishes. This means Alex's belief about his perceived time no longer matches the Clock's time. Alex's thoughtframe, *tf_FeelHungry*, becomes active. This places two "conclude" instructions on Alex's current thoughtframe stack. As mentioned above, thoughtframes act like workframes but only involve belief updates (conclude instructions) and so take no time. The rule 'Pop_concTf' updates an agent's belief using a thoughtframe.

RULE: Pop_concTf

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

After applying 'Pop_concTf' twice to conclude first *perceivedtime* = *Clock.time* and then *hungry* = *hungry* + *3* Alex's beliefs are:

$$\left\{ \begin{array}{l} hungry = 18, Loc = Library, \\ Bob.Loc = Home, percievedtime = 1, \\ Clock.time = 1, \quad desiredRestaurant = \emptyset \end{array} \right\}$$

Alex continues to study. The cycle of Alex and the Clock counting time, studying and increasing hunger continues until the point where Alex's hunger level is 21 or above and he decides it is time to find some food.

The situation is as follows: the simulation time is 10800; Alex's hunger is 21; the Clock's time attribute is 2 and it has just completed a primitive activity causing Alex's *tf_FeelHungry* thoughtframe to execute; Alex has an active workframe with 1200 seconds remaining of study time. However, a thoughtframe has now been activated and has updated Alex's beliefs to conclude that his intended restaurant is Raleigh's. In our tuple representation the states of the Alex agent and the Clock object are:

$$\langle ag_{Alex}, \emptyset, \langle Study_d, [\texttt{Prim\_Act}^{1200}]\rangle, Pop\_*, B_{Alex}, F, 10800, TF_{Alex}, WF_{Alex}\rangle$$

$$\langle ob_{Clock}, \emptyset, \emptyset, fin, B_{Clock}, F, 10800, TF_{Clock}, WF_{Clock} \rangle$$

Alex is now hungry. Although Alex has a currently active workframe, the workframe $wf\_MoveToRestaurant$ is now applicable and has a higher priority. This requires that the current workframe is suspended. This is achieved by creating a new workframe which stores wf_Study's remaining instructions. This is achieved by 'Wf_Suspend'.

RULE: Wf_Suspend

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

where we use the notation $\beta[\beta^{pri}/(\beta^{pri}+0.2)]$ to indicate that the priority value of β has been replaced by $\beta^{pri}+0.2$.

After applying this rule Alex's set of workframes becomes

$$WF_{Alex} = \left\{ \begin{array}{lll} \langle Study_d, [\texttt{Prim\_Act}^{1200}] \rangle, & wf\_MoveToRestaurant, \\ wf\_Study, & wf\_Wait, & wf\_Eat \end{array} \right\}$$

Alex calls Bob to meet. A simple communication activity is performed by Alex during a workframe which sends a message to Bob indicating that he wishes to meet for food. This communication works like a primitive activity followed by a simple belief update, the primitive activity would represent the duration of the communication. This communication will change the beliefs Bob has about Alex such that Bob will now believe (for simplicity) `meetAlex = true`, `Alex.desiredRestaurant = Raleigh`. Bob will then perform actions, similar to Alex's in the following, in order to get to Raleigh's restaurant.

Alex goes out for food. Alex has now selected a workframe to move to Raleigh's restaurant, $wf\_MoveToRestaurant$. This workframe is different to those we have seen previously because it has a *move* activity, which acts like a primitive activity followed by a conclude. The primitive activity has the duration dependant on journey time (calculated via pre-processing) and the conclude updates beliefs and facts of the our agent's location which, in this case, will be Raleigh's restaurant.

Waiting for Bob. Alex has arrived before Bob, so Alex initiates a workframe to wait for Bob. This workframe contains a detectable which detects when Bobs location is `Telegraph_Av_2405`. When Bob eventually arrives, an external belief (a fact) is updated which activates the detectable. The detectable on Alex's workframe which is of type 'Abort'. This abortion will cancel the current workframe and any activities Alex

is working on but will also update his beliefs to match the fact (Bob's new location). The abort is is handled by 'Det_Abort':

RULE: Det_Abort

$$\langle ag_i, \emptyset, \langle \beta_d, \beta_{ins} \rangle, Det\_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[\exists d \in \beta^D | \exists F' \subseteq F | = d^g \,\&\, d^{type} = Abort]{B_i' = B_i \cup F'}$$

$$\langle ag_i, \emptyset, \emptyset, Det\_*, B_i', T_i, F, TF_i, WF_i \rangle$$

where d is the current detectable, β^D is the set of all detectables for workframe β, d^g is the *guard* condition of the detectable d, and d^{type} is the detectable type: *Impasse*; *Complete*; *Continue*; or *Abort*.

Scenario Conclusion. Alex's waiting has now been terminated and guards are satisfied for him to start the workframe to eat with Bob. The scenario finally terminates when the Clock object's time attribute has reached 20 hours, this is an additional condition in every single workframe/thoughtframe. After 20 hours each entity will no longer have any frames active so they all enter an idle state prompting the scheduler to use the 'Sch_Term' rule.

5 Concluding Remarks and Future Work

In this paper we have outlined the first formal semantics for the Brahms language. While the full semantics is given in the associated technical report [17], the worked scenario described above demonstrates much of the semantics of Brahms, including the most important aspects: *selection* of workframes and thoughtframes; *suspension* of workframes when a more important (higher priority) workframe becomes active; *detection* of facts; *performance* of 'concludes', primitive activities, 'move' activities and communication activities; and the use of the scheduler.

This formal semantics 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` [7] or agent model checkers such as `AJPF` [1]. Currently we are developing a translator for Brahms which, via the transition rules in our semantics, will then be able to generate input for a range of model checkers such as `Spin`, `AJPF` or `NuSMV` [3].

As well as developing a model checking tool, we are also currently identifying a suite of example Brahms scenarios (together with their required properties) for evaluating this tool. For the small Brahms example developed in Section 4, we might wish to verify that: "Alex will never starve"; or "Alex will eventually reach Raleigh's". Brahms is an important language. It has been used to model very extensive applications in agent-human teamwork. While we have emphasized applications in space exploration, Brahms is equally at home in describing more mundane applications in home healthcare or human-robot interaction. As such the formal verification of this language would be very useful for assessing human safety; the operational semantics developed here are a necessary first step towards this.

References

1. R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated Verification of Multi-Agent Programs. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 69–78, 2008.
2. R. H. Bordini, M. Fisher, and M. Sierhuis. Formal Verification of Human-Robot Teamwork. In *Proc. 4th ACM/IEEE International Conference on Human Robot Interaction (HRI)*, pages 267–268. ACM Press, 2009.
3. A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri. *NUSMV: A New Symbolic Model Verifier*, 1999 Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, Springer, 1999.
4. 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 (FLAIRS)*, pages 7–11. AAAI Press, 2003.
5. W. J. Clancey, M. Sierhuis, C. Seah, F. Reynolds, T. Hall, and M. Scott. Multi-Agent Simulation to Implementation: A Practical Engineering Methodology for Designing Space Flight Operations. In *Proc. 8th Annual International Workshop on Engineering Societies in the Agents World (ESAW)*, LNAI, Springer, 2008.
6. L. A. Dennis, M. Fisher, A. Lisitsa, N. Lincoln, and S. M. Veres. Satellite Control Using Rational Agent Programming. *IEEE Intelligent Systems*, 25(3):92–97, 2010.
7. G.J. Holzmann *Software model checking with SPIN*. Advances in Computers, 2005.
8. A. S. Rao and M. Georgeff. BDI Agents: From Theory to Practice. In *Proc. 1st International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, USA, 1995.
9. A. S. Rao and M. P. Georgeff. Modeling Agents within a BDI-Architecture. In *Proc. Conference on Knowledge Representation & Reasoning (KR)*. Morgan Kaufmann, 1991.
10. 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, The Netherlands, 2001.
11. M. Sierhuis. Multiagent Modeling and Simulation in Human-Robot Mission Operations. (See `http://ic.arc.nasa.gov/ic/publications`), 2006.
12. M. Sierhuis. Brahms Language Specification. (See `http://www.agentisolutions.com/documentation/language/LanguageSpecificationV3.0F.pdf`).
13. M. Sierhuis, J. M. Bradshaw, A. Acquisti, R. V. Hoof, R. Jeffers, and A. Uszok. Human-Agent Teamwork and Adjustable Autonomy in Practice. In *Proc. 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, 2003.
14. M. Sierhuis and W. J. Clancey. Modeling and Simulating Work Practice: A Human-Centered Method for Work Systems Design. *IEEE Intelligent Systems*, 17(5), 2002.
15. M. Sierhuis, W. J. Clancey, R. J. v. Hoof, C. H. Seah, M. S. Scott, R. A. Nado, S. F. Blumenberg, M. G. Shafto, B. L. Anderson, A. C. Bruins, C. B. Buckley, T. E. Diegelman, T. A. Hall, D. Hood, F. F. Reynolds, J. R. Toschlog, and T. Tucker. NASA's OCA Mirroring System: An application of multiagent systems in Mission Control, 2009.
16. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.
17. R. Stocker, M. Fisher, L. Dennis, and C. Dixon. A Formal Semantics for the Brahms Language. (See `http://www.csc.liv.ac.uk/ rss/publications`), 2011.
18. R. van Hoof. Brahms website: `http://www.agentisolutions.com`, 2000.
19. M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.