

A Formal Semantics for Brahms Technical Report*

Richard Stocker¹, Maarten Sierhuis², Louise Dennis¹, Clare Dixon¹, Michael Fisher¹

¹ Department of Computer Science, University of Liverpool, UK

² PARC, Palo Alto, USA

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

1 Introduction

Brahms is a multi-agent modelling, simulation and development environment devised by Sierhuis [1] and subsequently developed at NASA Ames Research Center. Brahms is a modelling language designed to model human activity using *rational agents* 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 [1] and [2].

2 Basic Anatomy of Brahms

2.1 Workframes and Thoughtframes

Workframes and thoughtframes represent the work and thought practise in Brahms; a thoughtframe is simply a restricted workframe. A workframe contains a sequence of activities and belief updates which the agent/object will perform, whereas a thoughtframe only contains sequences of belief updates. Workframes can also detect (using *detectables*) changes in their environment, bring this change into the agent's belief base and then decide whether or not it should continue executing. Essentially, workframes represent the work processes involved in completing a task and thoughtframes represent the reasoning process upon the current beliefs, e.g. "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 thought process (thoughtframe) stating that, since it is raining and I want to go the shops, then I need a raincoat".

2.2 Executing plans: Activities and Concludes

Agents are able to perform activities and concludes (belief/fact updates), these are executed via workframes and thoughtframes (concludes only) which decide when they should be performed. *Primitive* activities, *move* activities and *communication* activities are the three main types of activities.

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

Primitive activities are conceptual activities in the sense that they do absolutely nothing except spend simulation time, whilst the assigned name infers what the agent was doing. e.g. *primitive* activity ‘dig_hole’ has a duration of 400, so takes 400 simulation seconds to perform where no belief or fact updates are made but from the name it can be assumed the agent was digging a hole. To confirm a hole was dug the workframe would require a conclude to update the beliefs and the facts that a hole now exists.

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

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

2.3 Detectables

Detectables are contained within workframes and can only be executed if their workframe is currently active. They can detect changes in facts and can: abort, continue or complete the workframe. When a detectable is executed it imports the fact it “detected” into the agent’s belief base and then either: *aborts* - deletes all elements from the workframe’s stack; *continues* - carries on regardless; or *completes* - deletes only activities from the workframe’s stack.

2.4 Variables

Variables provide a method of quantification within Brahms. So, if there are multiple objects or agents which can match the specifications in a guard condition then the variable can either perform: *forone* - select one; *foreach* - work on all, one after another; or *collectall* - work on all at once.

2.5 Brahms Syntax

Brahms has a complex syntax for creating systems, agents and objects, although there is no space here to cover the full specification ¹. 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`.

¹ For the full syntax (with an informal semantics) see [?]

```

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 Brahms Code

3.1 Geography

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

```

area AtmGeography instanceof World { }
areadef University extends BaseAreaDef { }
areadef UniversityHall extends Building { }
areadef BankBranch extends Building { }
areadef Restaurant extends Building { }
area Berkeley instanceof City partof AtmGeography { }
area UCB instanceof University partof Berkeley { }
area SouthHall instanceof UniversityHall partof UCB { }
area Telegraph_Av.2405 instanceof Restaurant partof Berkeley { }
area Bancroft_Av.77 instanceof BankBranch partof Berkeley { }

path StH.to_from_RB {
  area1: SouthHall;
  area2: Telegraph_Av.2405;
  distance: 400;
}
path StH.to_from_WF {
  area1: SouthHall;

```

```

    area2: Bancroft_Av_77;
    distance: 200;
}

```

3.2 Group

Groups in Brahms form the hierarchical structure of agents, where agents can be members of groups and groups can also be members of other groups. Groups form a template for an agent, so if an agent is a member of a group then it will inherit all the beliefs, workframes and thoughtframes declared in the group. The following Brahms codes describes a group called student, any agent who is a member of student will inherit all the attributes, relations, beliefs etc.

```

group Student {
  attributes:
    public boolean male;
    public double howHungry;
    public int perceivedtime;
    public Diner chosenDiner;
  relations:
    public Account hasAccount;
    public Cash hasCash;
  initial_beliefs:
    (current.checkedDiner = false);
    (BlakesDiner.location = Telegraph.AV-2134);
  activities:
    moveToLocation(Building loc) {
      location: loc;
    }
    primitive_activity study() {
      max_duration: 3000;
    }
  workframes:
    workframe wf_moveToRestaurant {
      repeat: true;
      variables:
        forone(Diner) dn;
        forone(Building) bd;
      when(knownval(current.howHungry > 20.00) and
        knownval(current.chosenDiner = dn) and
        not(current.location = dn.location) and
        knownval(dn.location = bd))
      do {
        moveToLocation(bd);
        conclude((current.readyToLeaveRestaurant = false)
          , bc:100,fc:0);
      }
    }
}

```

```

workframe wfeat {
  repeat: true;
  variables:
    forone(Cash) cs;
    forone(Diner) dn;
  when(knownval(current hasCash cs) and
    knownval(current.location = dn.location))
  do {
    eat();
    conclude((current.howHungry = current.howHungry - 3.00)
      , bc:100, fc:0);
    conclude((cs.amount = cs.amount - dn.foodcost)
      , bc:100, fc:100);
    conclude((current.readyToLeaveRestaurant = true)
      , bc:100, fc:0);
  }
}
thoughtframes:
  thoughtframe tf_chooseBlakes {
    repeat: true;
    variables:
      forone(Cash) cs;
    when(knownval(current hasCash cs) and
      knownval(cs.amount > 15.00) and
      knownval(current.checkedDiner = false) and
      knownval(Campanile_Clock.time < 20))
    do {
      conclude((current.chosenDiner = Blakes_Diner), bc:100);
      conclude((current.checkedDiner = true), bc:100);
    }
  }
}

```

3.3 Agent

Agents are defined in exactly the same way as groups with the exception that the agent code can define the agent's location. Agent code tends to be much smaller than group code because most of the key features (activities, workframes and thoughtframes) have been defined in all the groups the agent is a member of. Agent code is for specifically describing what is personal to that agent. The following Brahms codes represents a person called Alex who is a member of the previously described group called student. Since all Alex's activities etc. have been described in the group student then all that needs to be defined is Alex's location and personal beliefs/facts.

```

agent Alex_Agent memberof Student {
  location: SouthHall;
  initial_beliefs:
    (current contains Alex_Cash);
}

```

```

        (current contains Alex_BankCard);
        (Alex.Account.balance = 20.00);
initial_facts:
        (current.male = true);
        (current contains Alex_Cash);
        (current contains Alex_BankCard);

```

4 Semantic Framework

4.1 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 Brahm's 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: α represents the *current thoughtframe*, \mathcal{T} represents any arbitrary *thoughtframe*, while TF represents a *set* of thoughtframes.

Activities: 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.

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*. This includes

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

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

4.2 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; the fourth is the set of facts in the environment; and the fifth is the current time of the system:

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

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.

$$\text{Agents tuple} = \langle ag_i, \mathcal{T}, W, stage, B, F, T, TF, WF \rangle$$

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

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

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 .

4.3 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. Workframes that 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.

$$\begin{array}{c} A_0 \xrightarrow{\text{LocalClock}+t} X, X \xrightarrow{\text{Choice}} A'_0 \\ \vdots \\ A_n \xrightarrow{\text{LocalClock}+t} X, X \xrightarrow{\text{Choice}} A'_n \\ \hline \xi \xrightarrow{\text{LocalClock}+t} \xi' \end{array}$$

5 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 clock central 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 conditions 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_**’ semantic rules where ‘***’ is a wild card.

RULE: Sch_run

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

Sch_rcvd. Received activity durations from all agents. This rule identifies when the Scheduler has received all the durations from all agents. States 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.

RULE: Sch_rcvd

$$\langle Ags, ag_i, B_\xi, F, T_\xi \rangle \xrightarrow[\forall ag_i \in Ags | stage = Pop_.(PA*/MA*/CA*) \cup idle, (T_\xi \neq -1)]{T_\xi' = T_\xi + MinTime(B_\xi(\forall ag_i(T_i)))} \langle Ags, ag_i, B_\xi, F, T_\xi' \rangle$$

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

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

6 Agent Semantics

The Brahms systems operates on a simple cycle of:

$$Thoughtframes \rightarrow Detectables \rightarrow Workframes$$

6.1 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 conditions 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

$$\frac{\langle ag_i, \emptyset, \emptyset \cup \beta, Set_*, B_i, F, T_i, TF_i, WF_i \rangle}{\exists (\mathcal{T} \cup W) \in (TF_i \cup WF_i) | B_i \models \mathcal{T}_i^g \cup W_i^g} \rightarrow \langle ag_i, \emptyset, \emptyset \cup \beta, Tf_*, B_i, F, T_i, TF_i, WF_i \rangle$$

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

$$\frac{\langle ag_i, \emptyset, \emptyset \cup \beta, Set_*, B_i, F, T_i, TF_i, WF_i \rangle}{\neg (\exists (\mathcal{T} \cup W) \in (TF_i \cup WF_i) | B_i \models \mathcal{T}_i^g \cup W_i^g)} \rightarrow \langle ag_i, \emptyset, \emptyset \cup \beta, idle, B_i, F, T_i, TF_i, WF_i \rangle$$

6.2 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 it will randomly (but fairly) choose.

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

RULE: Tf_Select

$$\frac{\langle ag_i, \emptyset, \beta \cup \emptyset, Tf_*, B_i, F, T_i, TF_i, WF_i \rangle}{\frac{\alpha = Maxpri(\mathcal{T} \in TF_i | B_i \models \mathcal{T}^g)}{\exists \mathcal{T} \in TF_i | B_i \models \mathcal{T}^g}} \rightarrow \langle ag_i, \alpha, \beta \cup \emptyset, Tf_ (true/false/once), B_i, T_i, F, TF_i, WF_i \rangle$$

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

$$\langle ag_i, \alpha^{r=true}, \beta \cup \emptyset, Tf_ (true/false/once), B_i, F, T_i, WF_i, TF_i \rangle$$

→

$$\langle ag_i, \alpha^{r=true}, \beta \cup \emptyset, Pop_Tf^*, B_i, T_i, F, TF_i, WF_i \rangle$$

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

RULE: Tf_once

$$\begin{aligned} & \langle ag_i, \alpha^{r=once}, \beta \cup \emptyset, Tf_-(true/false/once), B_i, F, T_i, TF_i, WF_i \rangle \\ & \xrightarrow{\alpha' = \alpha[\alpha^{r=once} / (\alpha^{r=false}), TF'_i = (TF_i - \alpha) \cup \alpha']} \\ & \langle ag_i, \alpha'^{r=false}, \beta \cup \emptyset, Pop_Tf^*, B_i, F, T_i, TF'_i, WF_i \rangle \end{aligned}$$

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

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

RULE: Tf_false

$$\begin{aligned} & \langle ag_i, \alpha^{r=false}, \beta \cup \emptyset, Tf_-(true/false/once), B_i, F, T_i, TF_i, WF_i \rangle \\ & \xrightarrow{TF'_i = TF_i - \alpha} \\ & \langle ag_i, \alpha^{r=false}, \beta \cup \emptyset, Pop_Tf^*, B_i, F, TF_i = TF'_i, T_i, TF_i, WF_i \rangle \end{aligned}$$

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

RULE: Tf_exit

$$\begin{aligned} & \langle ag_i, \emptyset, \beta \cup \emptyset, Tf_*, B_i, F, T_i, TF_i, WF_i \rangle \\ & \xrightarrow{\neg(\exists T \in TF_i) | B_i \models T^g} \\ & \langle ag_i, \emptyset, \beta \cup \emptyset, Det_*, B_i, F, T_i, TF_i, WF_i \rangle \end{aligned}$$

6.3 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, but if there is more than one it will randomly (but fairly) choose. The fairness includes that the current workframe has higher priority than all others.

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

RULE: Wf_Select

$$\begin{aligned} & \langle ag_i, \emptyset, \emptyset, Wf_*, B_i, F, T_i, TF_i, WF_i \rangle \\ & \xrightarrow{\begin{array}{l} \beta = Max_{pri}(W \in WF_i | F \models W^g) \\ \exists W \in WF_i | B_i \models W^g \end{array}} \\ & \langle ag_i, \emptyset, \beta, Wf_-(true/false/once), B_i, F, T_i, TF_i, WF_i \rangle \end{aligned}$$

Wf_suspend. If an agent is currently working on a workframe, but there exists a workframe with its guard condition met and 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.

RULE: Wf_Suspend

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

$$\frac{\beta' = \beta[\beta^{pri} / (\beta^{pri} + 0.2)], WF'_i = WF_i \cup \beta'}{\exists W \in WF_i | B_i \models Wg \& W^{pri} > (\beta^{pri} + 0.3)}$$

$$\langle ag_i, \emptyset, \emptyset, Wf_{-*}, B_i, F, T_i, TF_i, WF'_i \rangle$$

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 isn't 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_*'

RULE: Wf_true

$$\langle ag_i, \emptyset, \beta^r = true, Wf_{-(true/false/once)}, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\frac{}{\neg(\exists W \in WF_i | B_i \models Wg \& W^{pri} > \beta^{pri} + 0.3)}$$

$$\langle ag_i, \emptyset, \beta^r = true, Var_{-*}, B_i, F, T_i, TF_i, WF_i \rangle$$

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.

RULE: Wf_once

$$\langle ag_i, \emptyset, \beta^r = once, Wf_{-(true/false/once)}, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\frac{\beta' = \beta[\beta^r = once / (\beta^r = false)], WF'_i = (WF_i - \beta) \cup \beta'}{\neg(\exists W \in WF_i | B_i \models Wg \& W^{pri} > \beta^{pri} + 0.3)}$$

$$\langle ag_i, \emptyset, \beta^r = false, Var_{-*}, B_i, F, T_i, TF_i, WF'_i \rangle$$

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.

RULE: Wf_false

$$\langle ag_i, \emptyset, \beta^r = false, Wf_{-(true/false/once)}, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\frac{WF'_i = WF_i - \beta}{\neg(\exists W \in WF_i | B_i \models Wg \& W^{pri} > \beta^{pri} + 0.3)}$$

$$\langle ag_i, \emptyset, \beta^r = false, Var_{-*}, B_i, F, T_i, TF_i, WF'_i \rangle$$

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

RULE: Det_cont

$$\frac{\langle ag_i, \emptyset, \beta, Det_*, B_i, F, T_i, TF_i, WF_i \rangle \quad B'_i = B_i \cup F'}{(\exists d \in \beta^D | (\exists F' \subseteq F) \models dg \& d^{type} = continue), \neg(\exists d \in \beta^D | (\exists F' \subseteq F) \models dg \& d^{type} = impasse \cup abort \cup complete)} \rightarrow \langle ag_i, \emptyset, \beta, Wf_*, B'_i, F, T_i, TF_i, WF_i \rangle$$

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

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

RULE: Det_comp

$$\frac{\langle ag_i, \emptyset, \beta, Det_*, B_i, F, T_i, TF_i, WF_i \rangle \quad B'_i = B_i \cup F', \beta' = \beta[\beta_{ins} / \beta^{Concludes}]}{(\exists d \in \beta^D | (\exists F' \subseteq F) \models dg \& d^{type} = complete), \neg(\exists d \in \beta^D | (\exists F' \subseteq F) \models dg \& d^{type} = impasse \cup abort)} \rightarrow \langle ag_i, \emptyset, \beta', Wf_*, B'_i, T_i, TF_i, WF_i \rangle$$

$\beta^{Concludes}$ 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).

RULE: Det_impasse

$$\frac{\langle ag_i, \emptyset, \beta, Det_*, B_i, F, T_i, TF_i, WF_i \rangle \quad B'_i = B_i \cup F', \beta' = \beta[\beta_{pri} / (\beta_{pri} + 0.1), \beta^g / (\beta^g \wedge \neg d^g)], WF'_i = WF_i \cup \beta'}{(\exists d \in \beta^D | (\exists F' \subseteq F) \models dg \& d^{type} = impasse), \neg(\exists d \in \beta^D | (\exists F' \subseteq F) \models dg \& d^{type} = abort)} \rightarrow \langle ag_i, \emptyset, \emptyset, Wf_*, B'_i, F, T_i, TF_i, WF'_i \rangle$$

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

RULE: Det_abort

$$\langle ag_i, \emptyset, \beta, Det_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[\exists d \in \beta^D | (\exists F' \subseteq F) \models d^g \& d^{type} = abort]{B'_i = B_i \cup F'} \langle ag_i, \emptyset, \emptyset, Wf_*, B'_i, T_i, TF_i, WF_i \rangle$$

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

RULE: Det_empty

$$\langle ag_i, \emptyset, \beta, Det_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[\neg(\exists d \in \beta^D | (\exists F' \subseteq F) \models d^g)]{} \langle ag_i, \emptyset, \beta, Wf_*, B_i, F, T_i, TF_i, WF_i \rangle$$

6.5 Var_* ruks (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.

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

RULE: Var_empty

$$\langle ag_i, \emptyset, \beta, Var_*, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[\beta^V = \emptyset]{} \langle ag_i, \emptyset, \beta, Pop_*, B_i, F, T_i, TF_i, WF_i \rangle$$

Var_set. Workframes with variables have an additional stack. This additional stack stores instances of the workframe with the differing variations 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.

RULE: Var_set

$$\langle ag_i, \emptyset, \langle \beta_d, [\emptyset], [\beta_{ins}] \rangle, Var_*, B_i, F, T_i, TF_i, WF_i \rangle$$

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

Var_one. When the variable is of type ‘forone’ and the subset of the workframe is none empty then the first workframe is pulled from the subset and set as the current workframe. The subset of variables in the workframe are then deleted. This is how Brahms performs unification.

RULE: **Var_one**

$$\langle ag_i, \emptyset, < \beta_d, [W_0 \dots W_n], [\beta_{ins}] >, Var_{-*}, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[\exists v \in \beta^V \mid v^{type} = forone]{\beta' = Random(W_0 \dots W_n)}$$

$$\langle ag_i, \emptyset, \beta', Pop_{-*}, B_i, F, T_i, TF_i, WF_i \rangle$$

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

Var_each. When the variable is of type ‘foreach’ and the subset of the workframe is none empty then the instances of the workframes are added to the set of workframes and the first instance is set as the current workframe. The instances are given a slightly increased priority and a repeat value of false so they will never be repeated. This represents Brahms operating on a multitude of tasks sequentially.

RULE: **Var_each**

$$\langle ag_i, \emptyset, < \beta_d, [W_0 \dots W_n], [\beta_{ins}] >, Var_{-*}, B_i, T_i, F, TF_i, WF_i \rangle$$

$$\xrightarrow[\exists v \in \beta^V \mid v^{type} = foreach]{WF'_i = WF_i \cup (W_0[W_0^{pri}/(\beta^{pri} + 0.1), W_0^r = false] \dots W_n[W_n^{pri}/(\beta^{pri} + 0.1), W_n^r = false])}$$

$$\langle ag_i, \emptyset, W_0, Pop_{-*}, B_i, F, T_i, TF_i, WF'_i \rangle$$

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

RULE: **Var_all**

$$\langle ag_i, \emptyset, < \beta_d, [W_0 \dots W_n], [\beta_{ins}] >, Var_{-*}, B_i, F, T_i, TF_i, WF_i \rangle$$

$$\xrightarrow[\exists v \in \beta^V \mid v^{type} = forall]{\beta' = W_0 \cup concludes(W_1 \dots W_n)}$$

$$\langle ag_i, \emptyset, \beta', Pop_{-*}, B_i, F, T_i, TF_i, WF_i \rangle$$

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

6.6 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. We represent three different rules for concludes: one for updating *beliefs*; one for *facts*; and one for both. Brahms additionally has probabilities that beliefs will be updated, these probabilities have not been taken into account in these semantics. With concludes we only describe the rules for workframes because performing concludes in thoughtframes are identical but on a different stack.

RULE: Pop_WfconcB

$$\begin{array}{c} \langle ag_i, \emptyset, < \beta_d, [conclude(b = v)^{belief}; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle \\ \xrightarrow[b = v' \in \beta_i]{B'_i = B_i / b = v' \cup b = v} \\ \langle ag_i, \emptyset, \beta, Pop_*, B'_i, T_i, F, TF_i, WF_i \rangle \end{array}$$

Note. The “belief” superscript on the conclude is to show the conclude is for updating beliefs only.

RULE: Pop_WfconcF

$$\begin{array}{c} \langle ag_i, \emptyset, < \beta_d, [conclude(b = v)^{fact}; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle \\ \xrightarrow[b = v' \in \beta_i]{F' = F / b = v' \cup b = v} \\ \langle ag_i, \emptyset, \beta, Pop_*, B_i, T_i, F', TF_i, WF_i \rangle \end{array}$$

RULE: Pop_WfconcBF

$$\begin{array}{c} \langle ag_i, \emptyset, < \beta_d, [conclude(b = v)^{belief/fact}; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle \\ \xrightarrow[b = v' \in \beta_i]{B'_i = B_i / b = v' \cup b = v, F' = F / b = v' \cup b = v} \\ \langle ag_i, \emptyset, \beta, Pop_*, B'_i, T_i, F', TF_i, WF_i \rangle \end{array}$$

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

RULE: Pop_concWfB

$$\langle ag_i, \emptyset, < \beta_d, [conclude(b = v)^{belief}; \beta_{ins}] >, Pop_concWf*, B_i, F, T, TF_i, WF_i \rangle$$

$$\frac{B'_i = B_i / b = v' \cup b = v}{b = v' \in \beta_i} \rightarrow \langle ag_i, \emptyset, \beta, Pop_concWf^*, B'_i, T_i, F, TF_i, WF_i \rangle$$

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_concWfB

$$\langle ag_i, \emptyset, \langle \beta_d, [\neg(\text{conclude}(b = v)); \beta_{ins}] \rangle, Pop_concWf^*, B_i, F, T, TF_i, WF_i \rangle \rightarrow \langle ag_i, \emptyset, \beta, fn, B'_i, T_i, F, TF_i, WF_i \rangle$$

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

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

RULE: Pop_PASend

$$\langle ag_i, \emptyset, \langle \beta_d, [Prim_Act^t; \beta_{ins}] \rangle, Pop_*, B_i, F, T, TF_i, WF_i \rangle \xrightarrow[\begin{smallmatrix} B_\xi = B_\xi \cup (T_i = T_i + t) \\ T_\xi = T_i \end{smallmatrix}]{\quad} \langle ag_i, \emptyset, \langle \beta_d, [Prim_Act^t; \beta_{ins}] \rangle, Pop_*, B_i, T_i, F, TF_i, WF_i \rangle$$

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

RULE: Pop_PA(t>0)

$$\langle ag_i, \emptyset, \langle \beta_d, [Prim_Act^t; \beta_{ins}] \rangle, Pop_*, B_i, F, T, TF_i, WF_i \rangle \xrightarrow[\begin{smallmatrix} t' = (T_\xi - T_i), T_i = T_\xi \\ T_\xi \neq T_i, (T_i + t - T_\xi) > 0 \end{smallmatrix}]{\quad} \langle ag_i, \emptyset, \langle \beta_d, [Prim_Act^{t'}; \beta_{ins}] \rangle, fn, B_i, T_i, F, TF_i, WF_i \rangle$$

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.

RULE: Pop_PA(t=0)

$$\langle ag_i, \emptyset, \langle \beta_d, [Prim_Act^t; \beta_{ins}] \rangle, Pop_*, B_i, F, T, TF_i, WF_i \rangle$$

$$\frac{T_i = T_\xi}{T_\xi! = T_i, (T_i + t - T_\xi) = 0} \rightarrow$$

$$\langle ag_i, \emptyset, < \beta_d, [\beta_{ins}] >, Pop_concWF^*, B_i, T_i, F, TF_i, WF_i \rangle$$

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 agents current location. This belief update occurs when the agent notices that the duration of the move has reached zero after the clock update. **Pop_PASend**. This is the rule the agent's use to send the duration of their next event to the scheduler.

RULE: Pop_moveSend

$$\langle ag_i, \emptyset, < \beta_d, [move(Loc = new)^t; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle$$

$$\frac{B_\xi = B_\xi \cup (T_i = T_i + t)}{T_\xi = T_i} \rightarrow$$

$$\langle ag_i, \emptyset, < \beta_d, [move(Loc = new)^t; \beta_{ins}] >, Pop_*, B_i, T_i, F, TF_i, WF_i \rangle$$

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.

RULE: Pop_move(t>0)

$$\langle ag_i, \emptyset, < \beta_d, [move(Loc = new)^t; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle$$

$$\frac{t' = (T_\xi - T_i), T_i = T_\xi}{T_\xi! = T_i, (T_i + t - T_\xi) > 0} \rightarrow$$

$$\langle ag_i, \emptyset, < \beta_d, [move(Loc = new)^{t'}; \beta_{ins}] >, fn, B_i, T_i, F, TF_i, WF_i \rangle$$

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

RULE: Pop_move(t=0)

$$\langle ag_i, \emptyset, < \beta_d, [move(Loc = new)^t; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle$$

$$\frac{T_i = T_\xi, B'_i = B_i / Loc = old \cup Loc = new, F' = F / Loc = old \cup Loc = new}{T_\xi! = T_i, (T_i + t - T_\xi) = 0} \rightarrow$$

$$\langle ag_i, \emptyset, < \beta_d, [\beta_{ins}] >, Pop_concWF^*, B'_i, T_i, F', TF_i, WF_i \rangle$$

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 scheduler time of next event when processing a communication.

RULE: Pop_commSend

$$\begin{array}{c} \langle ag_i, \emptyset, < \beta_d, [Comms(ag_j, b = v)^t; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle \\ \xrightarrow[T_\xi = T_i]{B_\xi = B_\xi \cup (T_i = T_i + t)} \\ \langle ag_i, \emptyset, < \beta_d, [Comms(ag_j, b = v)^t; \beta_{ins}] >, Pop_*, B_i, T_i, F, TF_i, WF_i \rangle \end{array}$$

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

RULE: Pop_comm(t>0)

$$\begin{array}{c} \langle ag_i, \emptyset, < \beta_d, [Comms(ag_j, b = v)^t; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle \\ \xrightarrow[T_\xi' = T_i, (T_i + t - T_\xi) > 0]{t' = (T_\xi - T_i), T_i = T_\xi} \\ \langle ag_i, \emptyset, < \beta_d, [Comms(ag_j, b = v)^t; \beta_{ins}] >, fn, B_i, T_i, F, TF_i, WF_i \rangle \end{array}$$

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

RULE: Pop_comm(t=0)

$$\begin{array}{c} \langle ag_i, \emptyset, < \beta_d, [Comms(ag_j, b = v)^t; \beta_{ins}] >, Pop_*, B_i, F, T, TF_i, WF_i \rangle \\ \xrightarrow[T_\xi' = T_i, (T_i + t - T_\xi) = 0]{T_i = T_\xi, B_j' = B_j / b = v' \cup b = v} \\ \langle ag_i, \emptyset, < \beta_d, [\beta_{ins}] >, Pop_concWF_*, B_i, T_i, F, TF_i, WF_i \rangle \end{array}$$

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

$$\begin{array}{c} \langle ag_i, < \alpha_d, [\emptyset] >, \emptyset \cup \beta, Pop_*, B_i, F, T_i, TF_i, WF_i \rangle \\ \rightarrow \\ \langle ag_i, \emptyset, \emptyset \cup \beta, Tf_*, B_i, F, T_i, TF_i, WF_i \rangle \end{array}$$

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

$$\begin{array}{c} \langle ag_i, \emptyset, < \alpha_d, [\emptyset] >, Pop_*, B_i, F, T_i, TF_i, WF_i \rangle \\ \rightarrow \\ \langle ag_i, \emptyset, \emptyset, Wf_*, B_i, F, T_i, TF_i, WF_i \rangle \end{array}$$

7 Examples of Semantics

This section details some examples of code to demonstrate the ideas behind these semantics. Each example is based on changing the colour of the moon. Each agent has the following code, the examples are additional thoughtframes/workframes to be added to this code:

```
agent X
{
    location: building_one;
    initialbeliefs:
        (moon.colour = ``blue``);
        (Campanile_Clock.time = 1);
    thoughtframes:
        ...
    workframes:
        ...
}
```

7.1 Thoughtframes/Concludes Take No Time

Here is a piece of code showing a thoughtframe which contains a single conclude statement.

```
agent one
{
    ...
    thoughtframes:
        thoughtframe tf_turnRed
        {
            when(knownval(Campanile_Clock.time <$$ 20))
            do
            {
                conclude((moon.colour = ``red``));
            }
        }
}
```

The output is represented in a table form displaying when the beliefs were introduced and what the beliefs are. This output shows how the agents believes that its initial belief that the moon is blue at time zero but because the thoughtframe starts and finishes at time zero then it changes its belief to the moon being red at time zero.

Time ▾	Belief/Fact
21600	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 7)
18000	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 6)
14400	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 5)
10800	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 4)
7200	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 3)
3600	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 2)
0	belief: (gov.nasa.arc.brahms.cleaner.one.location = gov.nasa.arc.brahms.cleaner.building_one)
0	belief: (gov.nasa.arc.brahms.cleaner.one.isMemberOf brahms.base.BaseGroup)
0	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "red")
0	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "blue")
0	belief: (gov.nasa.arc.brahms.cleaner.houseOne.location = gov.nasa.arc.brahms.cleaner.building_one)
0	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 1)

Time taken to update from blue to red is zero

7.2 Thoughtframes before Workframes

Here is a workframe and a thoughtframe, the workframe has a very high priority and the thoughtframe a very low priority. The workframe declares the moon is silver and the thoughtframe declares it to be red. The thoughtframe is repeatable and has a counter so it will loop through twenty times.

```
agent one
{
  ...
  workframes:
    workframe wf_turnSilver
    {
      priority: 1000;
      when(knownval(Campanile_Clock.time <$$ 20))
      do
      {
        changeColour();
        conclude((moon.colour = `silver`));
      }
    }
  thoughtframes:
    thoughtframe tf_turnRed
    {
      priority: 1;
      when(knownval(current.counter <$$ 20))
      do
      {
        conclude((current.counter = current.counter +1));
        conclude((moon.colour = `red`));
      }
    }
  }
```

```
}

```

If thoughtframes did not operate before workframes then the moon should be changed to silver first, since it that workframe has highest priority. However as we see in the printout the thoughtframe to change the moon red operates all twenty times before the workframe to turn it silver activates.

0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 5)
0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 6)
0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 7)
0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 8)
0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 9)
0	belief: (gov.nasa.arc.brahms.deaner.one.location = gov.nasa.arc.brahms.deaner.building_one)
0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 1)
0	belief: (gov.nasa.arc.brahms.deaner.one.isMemberOf brahms.base.BaseGroup)
0	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "red")
0	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "blue")
0	belief: (gov.nasa.arc.brahms.deaner.houseOne.location = gov.nasa.arc.brahms.deaner.building_one)
0	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 1)

7.3 Workframes Interrupted When Higher Priority Frame Becomes Available

Here is two workframes, one is started on initialisation (low priority) and other is only invoked when the moon becomes silver (with higher priority).

```
agent one
{
  ...
  workframes:
    workframe wf_turnSilverBlue
    {
      priority: 1;
      when(knownval(Campanile_Clock.time < $$ 20))
      do
      {
```

```

        changeColour();
        conclude((moon.colour == `silver`));
        changeColour();
        conclude((moon.colour == `blue`));
    }
}
bddf08    workframe wf_turnRed
{
    priority: 2;
    when(knownval(moon.colour == `silver`))
    do
    {
        changeColour()
        conclude((moon.colour == `red`));
    }
}
}

```

If there was no interruption of workframes then the moon would change colour from silver back to blue. However there is interruption and as soon as the moon becomes silver the other workframe which has higher priority becomes active and changes the moon the red. Once this higher priority workframe has finished it returns to the previous workframe and changes the moon colour to blue.

Time ▾	Belief/Fact
14400	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 5)
10800	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 4)
7200	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 3)
6000	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "blue")
4000	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "red")
3600	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 2)
2000	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "silver")
0	belief: (gov.nasa.arc.brahms.deaner.one.location = gov.nasa.arc.brahms.deaner.building_one)
0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 1)
0	belief: (gov.nasa.arc.brahms.deaner.one.isMemberOf brahms.base.BaseGroup)
0	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "blue")
0	belief: (gov.nasa.arc.brahms.deaner.houseOne.location = gov.nasa.arc.brahms.deaner.building_one)
0	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 1)

7.4 Suspended Workframe Higher Priority Than Others

To take the previous example further this example shows the same scenario but an additional workframe is added which becomes active when the moon is red. This additional workframe will have the same priority as the first workframe. Only additional workframe is displayed.

```

agent one
{
  ...
  workframes:
    workframe wf_turnSilverBlue
    {
      priority: 1;
      when(knownval(Campanile_Clock.time < $$ 20))
      do
      {
        changeColour();
        conclude((moon.colour = `silver`));
        changeColour();
        conclude((moon.colour = `blue`));
      }
    }
    workframe wf_turnRed
    {
      priority: 2;
      when(knownval(moon.colour = `silver`))
      do
      {
        changeColour()
        conclude((moon.colour = `red`));
      }
    }
    workframe wf_turnPink
    {
      priority: 2;
      when(knownval(moon.colour = `red`))
      do
      {
        changeColour()
        conclude((moon.colour = `pink`));
      }
    }
  }
}

```

Since the suspended workframes have a naturally higher priority than none-suspended then the suspended workframe is reactivated and finishes, meaning the new third workframe never becomes active producing the same result as before.

Time ▾	Belief/Fact
14400	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 5)
10800	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 4)
7200	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 3)
6000	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "blue")
4000	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "red")
3600	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 2)
2000	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "silver")
0	belief: (gov.nasa.arc.brahms.deaner.one.location = gov.nasa.arc.brahms.deaner.building_one)
0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 1)
0	belief: (gov.nasa.arc.brahms.deaner.one isMemberOf brahms.base.BaseGroup)
0	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "blue")
0	belief: (gov.nasa.arc.brahms.deaner.houseOne.location = gov.nasa.arc.brahms.deaner.building_one)
0	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 1)

7.5 Fairness

The following two workframes shows that Brahms implements fairness when multiple active workframes have equal priority. Each workframe has the same priority and are repeatable, they each change the moons colour.

```
agent one
{
  ...
  workframes:
    workframe wf_turnSilver
    {
      priority: 1;
      when(knownval(Campanile_Clock.time <$$ 20))
      do
      {
        changeColour();
        conclude((moon.colour = ``silver``));
      }
    }
    workframe wf_turnRed
    {
      priority: 2;
      when(knownval(Campanile_Clock.time <$$ 20))
      do
      {
        changeColour();
        conclude((moon.colour = ``red``));
      }
    }
  }
}
```

The results show that Brahms implements fairness because the colour of the moon alternates between the two workframes without one workframe ever becoming totally or even partially dominant.

Time ▾	Belief/Fact
30000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "silver")
28800	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 9)
28000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "red")
26000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "silver")
25200	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 8)
24000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "red")
22000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "silver")
21600	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 7)
20000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "red")
18000	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 6)
18000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "silver")
16000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "red")
14400	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 5)
14000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "silver")
12000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "red")
10800	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 4)
10000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "silver")
8000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "red")
7200	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 3)
6000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "silver")
4000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "red")
3600	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 2)
2000	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "silver")
0	belief: (gov.nasa.arc.brahms.cleaner.one.location = gov.nasa.arc.brahms.cleaner.building_one)
0	belief: (gov.nasa.arc.brahms.cleaner.one.counter = 1)
0	belief: (gov.nasa.arc.brahms.cleaner.one.isMemberOf brahms.base.BaseGroup)
0	belief: (gov.nasa.arc.brahms.cleaner.moon.colour = "blue")
0	belief: (gov.nasa.arc.brahms.cleaner.houseOne.location = gov.nasa.arc.brahms.cleaner.building_one)
0	belief: (gov.nasa.arc.brahms.cleaner.Campanile_Clock.time = 1)

7.6 Detect Fact Change and Import into Beliefs

The next example requires the use of two agents, where one agent changes a fact and another detects the change. The first agent simply changes the colour of the moon to silver. The second uses a primitive activity (long activity) to wait for the moon to change colour and then when it does it changes the moons colour itself.

```
agent one
{
  ...
  workframes:
    workframe wf_turnSilver
    {
      priority: 1;
      when(knownval(Campanile_Clock.time < $$ 20))
      do
      {
        changeColour();
        conclude((moon.colour = `silver`));
      }
    }
}
```

```

    }
  }
}

agent two
{
  ...
  workframes:
    workframe wf_turnRed
    {
      detectables:
        detectable moonChange {
          when (whenever)
            detect ((moon.colour = ``silver``))
            then continue;
        }
      priority: 1;
      when (knownval (Campanile_Clock.time < 20))
      do
      {
        wait ();
      }
    }
}

```

From the results it can be seen that agent one changes the moons colour to silver after 2000 seconds and that at the same time agent two updates its beliefs to believe this new fact.

Agent 1

Time ▾	Belief/Fact
2000	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "silver")
0	belief: (gov.nasa.arc.brahms.deaner.houseOne.location = gov.nasa.arc.brahms.deaner.building_one)
0	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "blue")
0	belief: (gov.nasa.arc.brahms.deaner.one isMemberOf brahms.base.BaseGroup)
0	belief: (gov.nasa.arc.brahms.deaner.one.counter = 1)
0	belief: (gov.nasa.arc.brahms.deaner.one.location = gov.nasa.arc.brahms.deaner.building_one)
0	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 1)

Agent 2

Time ▾	Belief/Fact
2000	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "silver")
0	belief: (gov.nasa.arc.brahms.deaner.houseFive.location = gov.nasa.arc.brahms.deaner.building_five)
0	belief: (gov.nasa.arc.brahms.deaner.moon.colour = "blue")
0	belief: (gov.nasa.arc.brahms.deaner.two isMemberOf brahms.base.BaseGroup)
0	belief: (gov.nasa.arc.brahms.deaner.two.busy = false)
0	belief: (gov.nasa.arc.brahms.deaner.two.location = gov.nasa.arc.brahms.deaner.building_five)
0	belief: (gov.nasa.arc.brahms.deaner.two.moon = "black")
0	belief: (gov.nasa.arc.brahms.deaner.Campanile_Clock.time = 1)

References

1. 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.
2. M. Sierhuis. *Multiagent Modeling and Simulation in Human-Robot Mission Operations*. (See <http://ic.arc.nasa.gov/ic/publications>), 2006.