Brahms

An Agent-Oriented Language for Work Practice Simulation and Multi-Agent Systems Development

Maarten Sierhuis, William J. Clancey, Ron J. J. van Hoof

Abstract Brahms is a multi-agent modeling language for simulating human work practice that emerges from work processes in organizations. The same Brahms language can be used to implement and execute distributed multi-agent systems, based on models of work practice that were first simulated. Brahms demonstrates how a multi-agent belief-desire-intention language, symbolic cognitive modeling, traditional business process modeling, activity- and situated cognition theories are brought together in a coherent approach for analysis and design of organizations and human-centered systems.

1 Motivation

Brahms was developed as a multiagent modeling and simulation language to visualize the social systems of work for business redesign projects [25]. In the early years (1992-1999), Brahms was purely a modeling and simulation language and tool designed to model people's work practice, i.e. the cultural,

Maarten Sierhuis
Carnegie Mellon University Silicon Valley/NASA Ames Research Center, Moffett Field, CA 94035,
Man-Machine Interaction Group, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: maarten.sierhuis-1@nasa.gov

William J. Clancey
NASA Ames Research Center, Moffett Field, CA 94035,
IHMC, Pensacola, FL
e-mail: william.j.clancey@nasa.gov

Ron J. J. van Hoof
Perot Systems/NASA Ames Research Center, Moffett Field, CA 94035,
e-mail: ronnie.j.vanhoof@nasa.gov

circumstantial, interactional influences on how work actually gets done, as opposed to an abstract top-down functional description of an organization's work process. In more recent years (2000-2003) we developed the Brahms language also as an agent-oriented language (AOL) for developing multi-agent systems (MAS). Besides running the Brahms virtual machine (BVM) as a simulation engine, by turning off the simulation clock, the BVM can execute its agents in real-time enabling the execution of a MAS. To couple human activity with external systems, there is an extensive Java application interface (JAPI) allowing the developer to integrate Brahms agents with external software applications, real-time devices, networks, etc, and develop agents completely in Java.

The Brahms language was originally conceived of as a language for modeling[1] contextual behavior of groups of people, called work practice.

> *Work Practice:* The collective performance of contextually situated activities of a group of people who coordinate, cooperate and collaborate while performing these activities synchronously or asynchronously, making use of knowledge previously gained through experiences in performing similar activities.

This created two very important ideas for the language; First, to model a group of people it is very natural to model them as software agents. Second, modeling situated behavior of a group imposes a constraint on the level of detail that is useful in modeling the dependent and independent behavior of the individuals. The right level is a representational level that falls between functional process models and individual cognitive models [6]. If we are interested in modeling a day-in-the-life of say ten or more people, modeling the individual behavior at the level of cognitive task models will be very time consuming, because these models are generally at the millisecond decision-making level. To overcome this kind of detail, the Brahms language uses a more abstract level of behavioral modeling that is derived from Activity Theory [27, 14] and Situated Action [26]. An individual's behavior is represented in terms of activities that take an amount of discrete time and can be decomposed into more detailed subactivities if necessary.

Brahms demonstrates how a multiagent belief-desire-intention (BDI) language, symbolic cognitive modeling, traditional business process modeling, activity- and situated cognition theories are brought together in a coherent approach for analysis and design of organizations and human-centered systems. The Brahms environment consists of different tools to develop, simulate or execute Brahms models and display agent and object interactions. Brahms is freely available for research purposes at the Brahms project website[2].

---

[1] We refer to Brahms programmers as modelers and Brahms programming as modeling, because we feel that Brahms is a fifth-generation multiagent model description language, rather than a third- or fourth-generation programming language.

[2] http://www.agentisolutions.com

2 Language

This section provides a detailed description of the Brahms language according to the specific criteria used to compare the different agent-oriented languages in this book. We first discuss the *specifications and syntactical aspects* of the Brahms language. This section discusses all major representational capabilities. Then, we discuss *semantics and verification*. Brahms is a practical language used in many applications at NASA. Brahms was not developed as a formal specification language for BDI and it does not have a formal semantic specification. One could be developed if desired. Brahms is a compiled language and does have a formal syntactic representation, which we will discuss. Next, we discuss *software engineering issues*. As a practical modeling and programming language, Brahms has many influences from object-oriented and rule-based languages. There is a definite influence from the Java programming language. The section ends with some more discussion of *other features of the language*, which focusses on modeling geographical environments.

The Brahms language is a pure AOL. It is *not* a set of Java libraries enabling agent-based programming in the Java language. Instead, Brahms is a full-fledged multiagent language allowing the modeler to easily and naturally represent *multiple agents*. The grammar of the Brahms language in this chapter is provided in EBNF (Extended Backus Naur Form) grammar rules. The notation used in these grammar rules is as in Table 1.

Table 1 Synopsis of the notation used in EBNF grammar rules

Construct	***Interpretation***
`::= * + {} [] | .`	Symbols part of the BNF formalism
`X ::= Y`	The syntax of X is defined by Y
`{X}`	Zero or one occurrence of X
`X*`	Zero or more occurrences of X
X+	One or more occurrences of X
`X | Y`	One of X or Y (exclusive or)
`[X]`	Grouping construct for specifying scope of operators, e.g. `[X|Y] or [X]*`
symbol	Predefined terminal symbol of the language
symbol	User-defined terminal symbol of the language
symbol	Non-terminal symbol

2.1 Specifications and Syntactical Aspects

This subsection answers the question; Does the Brahms language support various agent concepts such as, mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour?

Although Brahms was originally developed for modeling people's behavior, the Brahms language is a *domain independent* language. This means that the modeler decides what a Brahms model represents. Agents can represent whatever autonomous entity the modeler wants to represent, such as a person, an animal, or an autonomous or intelligent system. The following Brahms language features are discussed:

- Mental attributes: attributes, relations, beliefs and facts, no explicit desires, frame instantiations (intentions).
- Deliberation: concluding new beliefs, and use of thoughtframes for reasoning.
- Adaptation: changing beliefs, execution activity behavior and reasoning based on context.
- Social Abilities: groups and group inheritance, communication, and modeling the environment (objects, geography and location).
- Reactive and Cognitive-based behavior: modeling activity behavior, versus pure cognitive behavior, detectables, workframe-activity subsumption.
- Communication: communication activities, and communicative acts.

Brahms is an agent-oriented BDI-like language. It allows easy creation of groups of agents that execute activities based on local beliefs. Below is a simple taxonomy of some of the language concepts discussed in this section:

GROUPS are composed of
  AGENTS having
    BELIEFS and doing
    ACTIVITIES executed by
      WORKFRAMES defined by
        PRECONDITIONS, matching agents beliefs
        PRIMITIVE ACTIVITIES
        COMPOSITE ACTIVITIES, decomposing the activity
        DETECTABLES, including INTERUPTS, IMPASSES
        CONSEQUENCES, creating new beliefs and/or facts
    DELIBERATION implemented with
      THOUGHTFRAMES defined by
        PRECONDITIONS, matching agents beliefs
        CONSEQUENCES, creating new beliefs

2.1.1 Agents, Groups, and Attributes

Agents. A Brahms model is always about the activities of agents. An agent is therefore the most central construct in a Brahms model. Agents adhere to the standard attributes we associate with agency. They are autonomous, can be deliberative, as well as reactive and proactive, and are bounded rational.

```
agent ::=
   agent agent-name { group-membership }
   {
      { display: literal-string ; }
      { cost: number ; }
      { time_unit: number ; }
      { location: area-name ; }
      { icon : literal-string ; }
      { attributes }
      { relations }
      { initial-beliefs }
      { initial-facts }
      { activities }
      { workframes }
      { thoughtframes }
   }

group-membership ::= memberof group-name [, group-name ]*

externalagt ::= external agent agent-name ;
```

In the Brahms language, an agent is defined by the keyword *agent*, followed by the agent's name and its *group-membership*. Just as in object-oriented modeling where the concept of an object class enables one to define a type of object, Brahms includes an agent *group* concept to define a group of multiple agents with a similar make-up and behavior (see below). An agent has a number of possible facets or elements that are optional (see agent syntax above).

Group and Group Membership. The concept of a *group* in Brahms is similar to the concept of a template or class in object-oriented programming. A group represents a collection of agents that can perform similar work and have similar properties. A group defines the *attributes* and *relations*, the *initial-beliefs* and *initial-facts*, and the *activities*, *workframes* and *thoughtframes* of members in the group. The difference with classes in object-oriented programming is that the relationship between a group and its members is not an IS-A relationship, but a MEMBER-OF relationship. This is why we speak of

"*a member of a group*" instead of "an instance of a group." An agent can be a member of one or more groups. An agent will inherit attributes, relations, initial-beliefs, initial-facts, activities, workframes and thoughtframes from the group(s) it is a member of.

```
group ::=
   group group-name { group-membership }
   {
      { display: literal-string ; }
      { cost: number ; }
      { time_unit: number ; }
      { icon : literal-string ; }
      { attributes }
      { relations }
      { initial-beliefs }
      { initial-facts }
      { activities }
      { workframes }
      { thoughtframes }
   }
```

Using group membership and inheritance, we can model agent organization:

- *Functional Roles*
  Groups are similar to that of functional roles in an organization. A group in Brahms could represent a typical role in an organization and the work activities that someone performs when playing that role. For example, we could represent the role of Maintenance Technician or Flight Controller as a group.
- *Structural or Organizational Groupings*
  A group can also depict an organizational group. For example, we can define a group as "members of the Work System Design & Evaluation group at NASA Ames Research Center." We could now describe the work-activities, and initial-beliefs of members of the WSD&E group, such as when they have group meetings, etc.
- *Informal and Social Groupings*
  We can also create informal and social groups related to conceptual definitions that make sense in the modeling activity. For instance, in modeling the people at work we could create a social group "all people meeting at the water cooler." We can now describe the activities, workframes, thoughtframes and initial-beliefs that people meeting at the water cooler have in common. This might not be that interesting, but in modeling people's interactions, with for example, legacy systems we could define an informal

group of "system-xyz users." In this group we can describe how people interact with system-xyz, and what initial-beliefs the system users have (e.g. the initial-belief that the system contains specific data).

The group inheritance diagram from figure 1 is written in the Brahms language as follows:

```
jimport gov.nasa.arc.brahms.modat.kfx.KfxFile;

group Student {
   attributes:
      public string name;
      public boolean male;
      private int howHungry;
      private double preferredCashOut;
      private long perceivedtime;
      public symbol colorHair;
      public BaseGroup spouse;
      public relation(Student) studyFriend;
}

group BrahmsModeler {
   attributes:
      public map myIntIndexMap;
      public map myStringIndexMap;
      public Group myGroup;
      public java(KfxFile) javaKfxFile;
}

agent Alex_Agent memberof Student, BrahmsModeler { }
agent Kim_Agent memberof Student { }
agent Joyce_Agent { }
```

The BaseGroup definition does not have to be made, because it is part of the Base Model library that is always imported. All groups are by definition a member of BaseGroup (see Fig. 1).

Attributes. After the optional agent (and group) facets, there are a number of sections in an agent (and group). First, the *attributes section* defines the attributes for the agent. Attributes represent a property of a group or agent. Figure 1 and the Brahms code above show the attribute definitions for the two groups Student and BrahmsModeler. Attributes do not have values as in object-oriented programming. Instead, attributes have values as part of the beliefs of an agent or facts in the world. Attribute values are assigned or changed by asserting new beliefs or facts (see section 2.1.2).

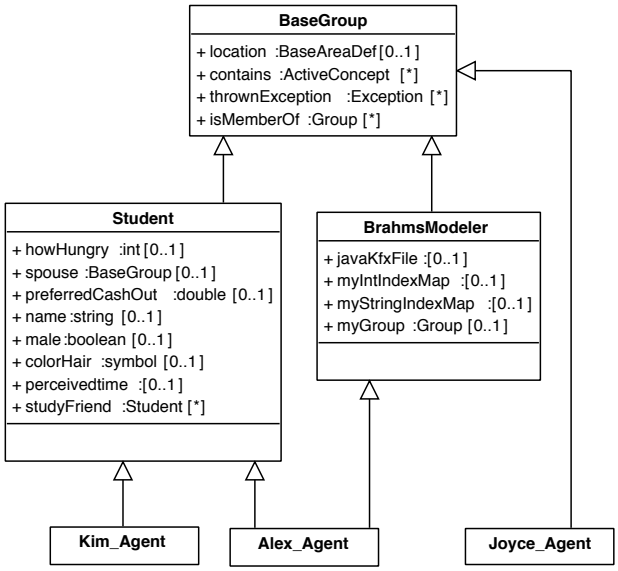

Fig. 1 Group membership and multiple-inheritance

Relations: Relations, defined in the *relations section*, represent a relationship between two objects, two agents, or an agent and an object. The scope of a relation is similar to that of an attribute. Relations are assigned or changed by asserting new beliefs or facts. Relations are unary relations between the left-hand side and the right-hand side in the relation (see fig. 2 and the source code below).

Fig. 2 Relations

```
class Song { ... }

object WhatAWonderfulWorld instanceof Song { ... }

object LaVieEnRose instanceof Song { ... }
```

```
group Artist {
   relations:
      public Song Performs;
}

agent LuisArmstrong memberof Artist {
   initial_beliefs:
      (current Performs WhatAWonderfulWorld);
      (current Performs LaVieEnRose);
}
```

2.1.2 Facts and Beliefs

In Brahms an agent acts according to its beliefs and its ability to deduce new beliefs from its current beliefs. In this section we describe the intentional notions of Brahms agents. The state of the world and that of agents in Brahms is stored in informational units called *facts* and *beliefs*.

Facts. A fact is meant to represent some physical state of the world or an attribute of some object or agent. Facts are globally true. Unlike objects, agents cannot reason with or act directly on facts, however, agents can detect facts in the world (representing noticing or sensing) turning them into beliefs for the agent (see section 2.1.7). Each BVM contains its own world fact-set containing all facts that are created during runtime by agents and objects running in that particular BVM. By representing part of the context of the agent as facts in the world, we are able to have agents react to the same facts in different ways, depending on their beliefs about these facts. Konolige defines a common fact, CF, as a fact that is known by all agents [13]. In Brahms, it is not necessary that an agent has any belief, right or wrong, about a fact. Although it is easily possible to have all agents inherit an initial-belief that corresponds to an initial-fact, or have all agents detect a particular fact at initialization, if it exists.

Beliefs. A belief represents an agent's interpretation of a fact in the world. A belief held by an agent may differ from the corresponding fact. For example, from our above example, if Alex is studying in South Hall he could believe "South Hall is 65 degrees" while the fact is that "South Hall is 80 degrees." A belief can also represent an agents conception of the world (s)he lives in. For example, our student Alex could believe "I am a student at University of California, Berkeley"—a belief about locations, or geography of the world the agents is located in, is modeled using the Brahms geography model described in section 2.4. Beliefs are local to an agent. Agents can reason about their

beliefs and create new beliefs, and agents can communicate their beliefs to other agents and objects.

Beliefs and facts can be defined as initial beliefs and facts at the group, agent, class and object level. Initial facts can also be defined for area-definitions and areas. An agent or object can also create beliefs and facts while performing an activity. A belief or fact can be thought of as an object-attribute-value triplet.

```
initial-belieforfact ::= ([value-expression | relational-expression]);

value-expression ::= obj-attr equality-operator value |
   obj-attr equality-operator sgl-object-ref

equality-operator ::= = | !=

relational-expression ::=
   tuple-object-ref relation-name sgl-object-ref { is truth-value }

value ::= literal-string | number | param-name | unknown

obj-attr ::=
   tuple-object-ref . attribute-name {( collection-index )}

tuple-object-ref ::= agent-name | object-name |
   conceptual-object-name | area-name |
   variable-name | param-name |
   current

sgl-object-ref ::= tuple-object-ref | unknown
```

Initial beliefs and facts are a way to populate agents and objects with an initial set of beliefs and create an initial set of facts in the world at agent initialization time. However, an agent can not do much if its belief-set or the world's fact-set cannot change. Agents and object can change beliefs and facts in the world by performing some behavior. How an agent or object can behave will be explained later on. Here we explain the command an agent or object can use to create new or change previously asserted beliefs and/or facts.

Facts and beliefs are created using a *conclude* statement. An agent or object can create or change either a belief for itself, or a fact in the world, or both. The syntax of the conclude statement is as follows.

```
consequence ::= conclude((resultcomparison){, belief-certainty}
   {, fact-certainty)};

resultcomparison ::= [ result-val-comp | rel-comp ]

result-val-comp ::=
      obj-attr equality-operator expression |
      obj-attr equality-operator literal-symbol |
      obj-attr equality-operator literal-string |
      obj-attr equality-operator sgl-object-ref |
      tuple-object-ref equality-operator sgl-object-ref

belief-certainty ::= bc: unsigned
fact-certainty ::= fc: unsigned
```

The conclude statement has two variables that can be specified after the result-comparison. These are the belief-certainty (bc) and fact-certainty (fc) variables. These variables specify with what certainty the agent creates the belief or fact respectively. The value of these variables can be an unsigned integer between the interval [0, 100], and specifies with what percent certainty the belief or fact is created. For example the following conclude statement states that the agent will get a belief "(current.male = true)" with 100% certainty, thus in all cases, and a fact with 0% certainty, thus never.

```
conclude((current.male = true), bc:100, fc:0);
```

2.1.3 Thoughtframes

Thoughtframes (thoughtframe! TFR) define deductions, mostly referred to as production rules. TFRs are taken to be inferences an agent makes. TFRs do not perform actions, consume no time, and cannot be interrupted. The only allowable statements in a TFR is one or more consequences, concluding new beliefs. Because TFRs represent reasoning, the agent cannot create or change facts in the world in a TFR. Therefore, no matter the value of the fact-certainty in a consequence in a TFR, a fact will never be created. A TFR consists of a variable declaration section, one or more preconditions and one or more consequences. The syntax of a TFR is given below.

```
thoughtframe ::=
   thoughtframe thoughtframe-name {
      { display : literal-string ; }
      { repeat : truth-value ; }
      { priority : unsigned ; }
      { variable-decl }
      { [ precondition-decl thoughtframe-body-decl ] |
      thoughtframe-body-decl }
   }

variable-decl ::= variables : [ variable ]*

precondition-decl ::=
   when ( { [ precondition ] [ and precondition ]* } )

thoughtframe-body-decl ::=
   do { [ thoughtframe-body-element ; ]* }

thoughtframe-body-element ::= consequence
```

As we will see later on, TFRs can be placed within composite activities, allowing for modeling problem-solving activities that take time. This is seen as: while the agent is 'in' the activity, the agent reasons using its TFRs. Conclusions of new beliefs in TFRs can execute new TFRs and/or workframes (WFR). Preconditions are similar for TFRs and WFRs.

Preconditions. When the preconditions of a TFR match the beliefs of the agent or object, its consequences are immediately executed, similar to forward-chaining production rules. An important point is that preconditions for agents only match with the beliefs of the agent. The syntax for preconditions is as follows.

```
precondition ::= [ known | unknown ] ( novalcomparison ) |
   [ knownval | not ] ( evalcomparison )

novalcomparison ::= obj-attr |
   obj-attr relation-name |
   tuple-object-ref relation-name
```

```
   evalcomparison ::= eval-val-comp | rel-comp

eval-val-comp ::=
   expression evaluation-operator expression |
   obj-attr equality-operator literal-symbol |
   obj-attr equality-operator literal-string |
   obj-attr equality-operator sgl-object-ref |

   sgl-object-ref equality-operator sgl-object-ref

rel-comp ::=
   obj-attr relation-name obj-attr { is truth-value } |
   obj-attr relation-name sgl-object-ref { is truth-value } |
   tuple-object-ref relation-name sgl-object-ref { is truth-value }
```

The agent's inference engine (part of the BVM) is implemented based on the well-known RETE algorithm [11]. However, unlike in OPS5 [3], each agent has two types of reasoning state networks (RSN); one for beliefs and one for facts. At this moment Brahms only supports conjunctions (AND) in its preconditions, but we will be adding disjunctions (OR) soon, because, even though you do not need disjunctions, it is sometimes easier for rule maintenance to be able to write rules more compactly using disjunctions.

Precondition Operators. There are four types of preconditions. The precondition types are operators that evaluate to *true* or *false*, depending on evaluating the belief-condition in each operator to match on one or more beliefs in the agent's belief-set.

knownval(evalcomparison) precondition operator evaluates to true iff:

Exists(belief b) [**Matches(b, evalcomparison)**]

This means that there exists a belief in the agent's belief-set that matches the evalcomparison. Given the below *TFR for agent Alex* and assuming the *belief-set for agent Alex*, both preconditions evaluate to *true*, firing the TFR.

```
thoughtframe tf_HowMuchMoneyToGet_HungryEQhigh_1 {
   when (knownval(current.needCash = true) and
         knownval(current.hungryness = high))
   do {
        conclude((current.preferredCashOut = 15), bc:100, fc:0);
   }
}
```

```
Alex' belief-set:
{
   (Alex.needCash = true);
   (Alex.hungryness = high);
}
```

The keyword *current* matches on the agent itself, and because the TFR is for agent Alex the two beliefs match on the preconditions. Both *knownval* preconditions return true, making the entire *when-clause* true and thus firing the TFR executing the *body-statement*. After the TFR execution, agent Alex' belief-set will now include the belief *(Alex.preferredCashOut = 15)*, because the belief-certainty is 100%. Unlike traditional production-rule systems (such as OPS5), the created belief will have a timestamp. This timestamp equals the simulation-clock time at which the TFR was fired (or system-clock time in case of real-time execution).

not(evalcomparison) precondition operator evaluates to true iff:

not(Exists(belief b) [Matches(b, evalcomparison)])

This means that none of the beliefs in the agent's belief-set match on the evalcomparison, similar to all beliefs do not match on the evalcomparison, and is the way to express that the precondition evaluates to true if the agent does not have a belief that matches the evalcomparison.

```
thoughtframe tf_HowMuchMoneyToGet_HungryEQhigh_2 {
   when (not(current.needCash = true) and
         knownval(current.hungryness = high))
   do {
         conclude((current.preferredCashOut = 10), bc:100, fc:0);
   }
}
Alex' belief-set:
{
   (Alex.needCash = false);
   (Alex.hungryness = high);
}
```

In this case all preconditions also match the beliefs of agent Alex. The *not* precondition evaluates to true, because even though the agent has a belief about the attribute needCash, the value of that belief is 'false' while the precondition tries to match on the value 'true'. Therefore, the precondition

evaluates to true. After firing of the above TFR, agent Alex will have the belief *(current.preferredCashOut = 10)*.

One might ask what would have happened if agent Alex does not have any belief about the attribute needCash for Alex, would the *not* precondition evaluate to true? The answer to this question is, yes it would, because it still would hold true that all of agent Alex' beliefs do not match the eval-comparison of the precondition.

Known precondition operator evaluates to true iff:

Exists(belief b) [Matches(b, novalcomparison)]

This means that there exists a belief in the agent's belief-set that matches the novalcomparison. A novalcomparison is a precondition expression that matches on any belief the agent has about the specified attribute or relation, without providing a necessary right-hand side value. Thus, regardless of the right-hand side value of the belief (whether for an attribute or a relation), if a belief exists the *known* precondition evaluates to *true*. Given the below TFR for agent Alex and assuming the belief-set for agent Alex, both preconditions evaluate to true, firing the TFR.

```
thoughtframe tf_HowMuchMoneyToGet_HungryEQhigh_3 {
   when (known(current.needCash) and
         knownval(current.hungryness = high))
   do {
         conclude((current.preferredCashOut = 15), bc:100, fc:0);
   }
}
Alex' belief-set:
{
   (Alex.needCash = false);
   (Alex.hungryness = high);
}
```

In this case all preconditions still also match the beliefs of agent Alex. The *known* precondition evaluates to true, because the agent has a belief about the attribute needCash, even though the value of that belief is 'false'. The precondition matches on any belief of the form *(Alex.needCash = <any-value>)*. After firing of the above TFR, agent Alex will have the belief *(current.preferredCashOut = 15)*.

unknown(novalcomparison) evaluates to true iff:

not(Exists(belief b) [Matches(b, novalcomparison))]

This means that none of the beliefs in the agent's belief-set match on the novalcomparison, similar to all beliefs do not match on the novalcomparison. Thus, regardless of the right-hand side value of the belief (whether for an attribute or a relation), if a belief exists the *unknown* precondition evaluates to *false*. This is a way to express that the precondition evaluates to true if the agent does not have a belief about a given attribute or relation. Given the below TFR for agent Alex and assuming the belief-set for agent Alex, the *unknown* preconditions evaluates to *false*, preventing the TFR from firing.

```
thoughtframe tf_HowMuchMoneyToGet_HungryEQhigh_4 {
   when (unknown(current.needCash) and
         knownval(current.hungryness = high))
   do {
         conclude((current.preferredCashOut = 5), bc:100, fc:0);
   }
}
Alex' belief-set:
{
   (Alex.needCash = true);
   (Alex.hungryness = high);
}
```

The *unknown* precondition's novalcomparison still matches on the needCash attribute belief for agent Alex. The *unknown* precondition thus evaluates to false, because the agent has a belief about the attribute needCash. The *unknown* precondition returns false for any belief of the form *(Alex.needCash = <any-value>)*.

One might ask what would happen if the right-hand side value of the belief is 'unknown', would the *unknown* precondition evaluate to true? The answer is no, in such a case the precondition would evaluate to false, because having a belief of the form *(Alex.needCash = unknown)* means that the agent knows about the attribute needCase, even though it does not know its specific boolean value.

Defining Thoughtframes. Thoughtframes can be defined at the "top-level" of a group, agent, class and/or object. Thoughtframes defined at the top-level are always active, meaning that their preconditions will be evaluated at each change in the agent's belief-set.

```
group Student {
      ...
   thoughtframes:
```

```
    thoughtframe tf_HowMuchMoneyToGet_HungryEQhigh_1 {
      when (knownval(current.needCash = true) and
            knownval(current.hungryness = high))
      do {
            conclude((current.preferredCashOut = 15), bc:100, fc:0);
      }//do
    }//tf_HowMuchMoneyToGet_HungryEQhigh_1
}//Student
```

It is also possible to define thoughtframes within a *composite activity*. In this case the thoughtframes will only be active when the agent is executing the composite activity, and is the way to model problem-solving as an activity taking time. For an explanation of composite activities see section 2.1.6.

```
group Student {
    ...
  activities:
    composite_activity SolveCashOutProblem( ) {
      ...
      thoughtframes:
        thoughtframe tf_HowMuchMoneyToGet_HungryEQhigh_1 {
          when (knownval(current.needCash = true) and
                knownval(current.hungryness = high))
          do {
                conclude((current.preferredCashOut = 15), fc:0);
          }//do
        }//tf_HowMuchMoneyToGet_HungryEQhigh_1
      ...
    }//composite_activity SolveCashOutProblem
    ...
}//Student
```

Variables. Variables can be used to write more generic rules (TFRs) or templates for activities (WFRs). Variables have quantifiers, as will be described below. Before a variable can be used it has to be declared, and the scope of the variable is bound to the frame it is declared in. There are three quantifiers for variables: *foreach*, *collectall*, and *forone*. Variables are used in preconditions to be bound to agents, objects or values. When bound in preconditions, variables can be used in consequences, detectables, and pass-by-reference parameters for activities. The syntax for defining a variable in a TFR or WFR is as follows.

```
variable ::=
   [foreach | collectall | forone ] ( type-def ) variable-name ;
```

The quantifier affects the way a variable is bound to a specific instance of the defined type of the variable.

Foreach quantifier. A foreach variable is bound to only one instance of its type definition, but for each instance that can be bound to the variable a separate frame instantiation is created. Consider, for example, the following thoughtframe with a foreach variable.

```
thoughtframe tf_CountOrders {
   variables:
      foreach(Order) order;
   when (knownval(order is-assigned-to current))
   do {
      conclude((current.numberOfOrders=current.numberOfOrders+1));
   }
}
```

If three orders are assigned to agent Alex_Agent and the agent has beliefs for all three of the orders matching the precondition, the agent's engine creates three TFR instantiations (TFI), and in each TFI the foreach variable is bound to one of the three orders.

Collectall quantifier. A collectall variable can be bound to more then one instance as a list. The variable is bound to all matching agent or object instances (depending on its type), and only one TFI is created. Consider the previous example with a collectall variable declaration:

```
variables:
   collectall(Order) order;
```

In this situation the agent's engine creates one TFI and binds the collectall variable to a list of all three orders.

Forone quantifier. A forone variable can be bound to only one instance or value, and only one TFI is created. A forone variable binds to a random belief-instance found and ignores other possible matches. In the previous example, the variable declaration would look like:

```
variables:
   forone(Order) order;
```

In this situation, similar to the collectall, only one TFI gets created, but, unlike in the collectall case, only one of the three orders gets bound to the variable.

Unbound variables. A declared variable need not be used in a precondition. In that case the variable is unbound (that is, it does not get a value) when a frame instantiation is created. An unbound variable can be bound in an activity. Note that an unbound variable may not be used in a consequence statement, which will always result in a runtime error.

Repeat. A TFR can be executed one or more times depending on the value of the *repeat* facet. In case the repeat facet is set to false, the TFR can only be performed one time for a specific binding of the variables, called a thoughtframe instantiation (TFI). The scope of the repeat facet of a TFR defined as part of a composite activity is limited to the time the activity is active, meaning that the TFR with a specific variable binding and a repeat facet set to false will not execute repeatedly while the composite activity is active.

Priority. Setting a TFR's *priority* facet allows control over the execution sequence of TFRs when more then one TFR is available at the same time. The priority facet can be set with a value greater or equal to zero. When two TFRs are available to be fired at the same time, the one with the highest priority will fire first. When both priorities are the same the sequence of execution is undefined. The default value for the priority facet, when it is not specified, is zero. Note that it is not recommended to use priorities to control the sequence of thoughtframe execution. A better modeling practice is to define preconditions controling TFR execution [5].

2.1.4 Primitive Activities

The central concept in Brahms, for the main purpose of modeling human behavior, is the concept of *activity*. An activity is an abstraction of real-life actions that help accomplish a daily task. A model of an agent's activities describes what the agent actually does over time (i.e. its behavior), based on the causal relationship between the decision to perform an activity and the past and present state of its context. In describing people's real-life activities, each activitiy in the world takes time no matter how short. A person is always within an activity taking action. Sleeping is an activity, waiting for the bus is an activity, simply doing nothing is an activity. Indeed, being alive is an activity. The following key points can be made about activities:

1. Reasoning is an activity taking time, not just an inference or deduction. Thus logical inferences happen within an activity.
2. Activities might not involve goals and tasks. For example, answering the phone is an activity that might not be part of any specific task that is being accomplished. In fact, it might be an activity that is interrupting the task being worked on.
3. Modeling activity behavior involves more than logical inferencing, namely the representation of chronological activities that agents do.
4. The activation of an activity is constrained by preconditions that are associated with an activity template it is part of. For example, activities may have preferential start times, as expressed in preconditions for the template, which may refer to the time in hours, minutes, seconds, day of the year, and/or day of the week.
5. An activity may be interrupted by a scheduled activity, such as going to lunch at noon. Time may change the priorities of activities and different people might do the same activities at different times.

In summary, activities are socially constructed engagements, situated in the real world, taking time, effort and application of knowledge, with a defined beginning and end, while not necessarily needing goals in the sense of problem-solving tasks, and being interruptable, resumable and able to be impassed. In this chapter we only describe the syntax and semantics of the Brahms activity language. For more discussion about the theory behind activities in Brahms, we refer the reader to [8, 19]. Brahms has different types of primitive activities.

- *Primitive activities*: These are the lowest level activities in an activity model. They are user-defined, take some time, but are not further specified in any detail. Parameters are time and resources. At any time during execution an agent is always executing some primitive activity. If an agent is not executing a primitive activity, one can say that the human-behavior model is underspecified.
- *Predefined activities*: These are language-level primitive activities with predefined semantics (e.g. communicate, move, get, put).
- *Java activities*: A Brahms Java activity is an user-defined primitive activity that is implemented as a Java class, using the Brahms JAPI. Java code may cause an action to happen completely outside the BVM (e.g. pop-up a dialog that says "hello world"). A Java activity can also do things within the BVM. Java code can generate output parameter values and assign them to unbound variables in a WFR, or generate new agents or objects within the Brahms model being executed. Java activities can also create new beliefs and facts, as well as interface to external systems.

activities ::= **activities :** [activity]*

```
activity ::=
   primitive-activity |
   predefined-activity |
   java-activity |
   composite-activity

predefined-activity ::=
   move-activity |
   create-agent-activity |
   create-object-activity |
   communicate-activity |
   broadcast-activity |
   get-activity |
   put-activity
```

All activities have to be declared in the activities section of either a group, agent, class, object, or composite activity. The declared activities can then be referenced in the workframes (WFRs) defined for the group, agent, class or object. It is possible to define input parameters for primitive activities. These input parameters can be used to make activities more generic. Activities can be assigned a priority. The priorities of activities in a workframe are used to define the priority of a workframe. The workframe will get the priority of the activity with the highest priority defined in the workframe. Activities and thus workframes have a duration. The duration of the activity can be defined to be a fixed amount of time or a random amount of time. For a fixed amount of time, the random facet has to be set to false and the max-duration attribute has to be set to the maximum duration in seconds. The duration of the activity can also be defined to be a random amount of time. To define a random amount of time the random facet has to be set to true, the min-duration facet has to be set to the minimum duration of the activity in seconds and the max-duration facet has to be set to the maximum duration of the activity in seconds.

```
primitive-activity ::=
primitive_activity activity-name ({param−decl [, param−decl ]*})
{
   { prim-act-facets }
}
```

```
prim-act-facets ::=
{ display : ID.literal-string ; }
{ priority : [ ID.unsigned | param-name ] ; }
{ random : [ ID.truth-value | param-name ] ; }
{ min_duration : [ ID.unsigned | param-name ] ; }
{ max_duration : [ ID.unsigned | param-name ] ; }
{ resources : [ param-name | OBJ.object-name ]
              [ , [ param-name | OBJ.object-name ]* ; }
```

Below is the definition of a Study primitive activity. The primitive activity takes a Book object as a parameter, which is used as a resource in the activity. The activity has a random duration between 30 minutes and 2 hours. The durations are given in seconds.

```
primitive_activity Study (Book course_book)
{
   display : "Study for a Cours" ;
   priority : 10 ;
   random : true ;
   min_duration : 1800 ; /* 30 mins */
   max_duration : 7200 ; /* 2 hours */
   resources : course_book;
}
```

2.1.5 Workframes

An agent cannot always apply all its available activities, given the agent's belief-state and the location it is in. Each activity is therefore associated with a conditional statement or constraint, representing a condition/activity template called *workframe* (WFR). WFRs are situation-action rules taking time. A WFR defines conditions under which an agent or object can perform an activity (or activities). WFRs are similar to TFRs, with similar syntax. They have preconditions that are matched against the belief-set of the agent or object. In short, activities describe *what* agents do, workframes describe *when* agents do what they do, thus defining when activities are executed. Workframes can be associated with groups/agents and classes/object. Having two agents with different workframes performing the same activity, can represent individual differences.

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

workframe-name ::= ID.name

variable-decl ::= variables : [ variable ]*

detectable-decl ::= detectables : [ detectable ]*

precondition-decl ::= when ({[precondition] [and precondition]*})

workframe-body-decl ::= do {[workframe-body-element]*}

workframe-body-element ::= [activity-ref | consequence |
        delete-operation]

activity-ref ::= activity-name({param-expr[, param-expr]*});

delete-operation ::= delete [ variable-name | param-name ];
```

A workframe is a more important unit than the simple precondition-activity-consequence design might suggest, because a workframe may model relationships involving location, object resources such as tools and documents, required information, other agents the agent is working and communicating with, and the state of previous or ongoing work. Active workframes may establish a context of activities for the agent and thereby model the agent's intentions, e.g., calling person X to give or get information, or going to the fax machine to look for document Y. In this way, behavior may be modeled as continuous across time, and not merely reactive.

```
group PrimitiveActivityPerformer {
   attributes:
      public boolean execute_PAC_1;

   activities:
      primitive_activity PAC_1(int pri) {
         display: "PAC 1";
         priority: pri;
         max_duration: 900;
      }


      primitive_activity PAC_2(int pri, int dur) {
         display: "PAC 2";
         priority: pri;
         max_duration: dur;
      }

   workframes:
      workframe wf_PAC_1 {
         repeat: true;
         when (knownval(current.execute_PAC_1 = true))
         do {
            PAC_1(1);
            conclude((current.execute_PAC_1 = false));
         }
      }

      workframe wf_PAC_2 {
         repeat: true;
         do {
            PAC_2(0, 1800);
            conclude((current.execute_PAC_1 = true), bc:25);
            PAC_2(0, 600);
         }
      }
}
```

Workframes can be interrupted and resumed, based on the priorities of activity references within. Priorities of activities can be dynamically assigned through parameters passed in. The above example shows two WFRs in the group PrimitiveActivityPerformer. WFR *wf_PAC_2* has no preconditions and *repeat = true*. This workframe will therefore always fire in an endless loop.

WFR *wf_PAC_1* has one precondition, which at the start of execution will always be false, since the agent will not have a belief that will match it.

On closer examination of the body of WFR *wf_PAC_2* we see that the WFR first calls activity *PAC_2* with priority zero and duration 1800 seconds and later on again with priority zero with a duration of 600 seconds. Given that the priorities of both activity-references in the WFR are zero, the WFR itself gets a priority of zero at instantiation (WFI).

However, in between the two activity-references the activity concludes a belief that will match the precondition for WFR *wf_PAC_1*. A match on this precondition would create a WFI for *wf_PAC_1* with priority one due to the fact that it calls activity *PAC_1* with priority one, and always has a duration of 900 seconds after which the conclude statement sets the belief to false, which stops the repeat=true and ends the WFI. Thus, WFR *wf_PAC_1* always only fires ones, even though the repeat is set to true.

Because the WFI for *wf_PAC_1* gets priority one its priority is higher than that of WFR *wf_PAC_2*. This *interrupts* the WFI for *wf_PAC_2* after the conclude statement, and starts a WFI for *wf_PAC_1* for 900 seconds. After the WFI for *wf_PAC_1* is completed, the WFI for *wf_PAC_2* again becomes the only WFI and thus automatically the highest priority WFI, and thus *wf_PAC_2* continues execution at the point it was interrupted, which is at the start of the second *PAC_2* activity-reference.

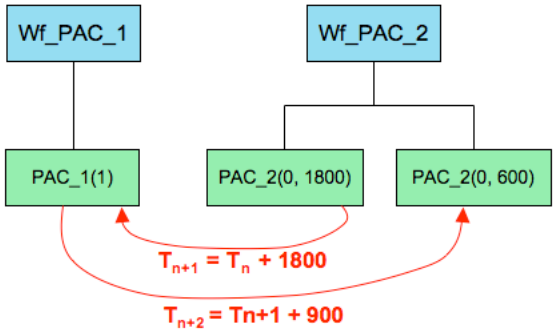

Fig. 3 Interrupted Workframe-Activity Hierarchy

As you can see in Fig. 3 WFR *wf_PAC_2* gets interrupted after the first call to activity *PAC_2*, at time $T_n + 1800$. At that time activity *PAC_1* starts executing. The next event is the end of activity *PAC_1* and thus of WFI *wf_PAC_1*. At time $T_{n+1} + 900$ WFI *wf_PAC_2* comes out of being interrupted and again becomes the current WFI, and the second activity *PAC_2* starts execution.

There is one more point to make about WFR *wf_PAC_2*; the conclude statement has a belief-certainty of 25. This means that the belief in the

conclude statement is only created in 25% of the time. This means that the interruption of *wf_PAC_2* only occurs once in four executions on average.

Primitive activities take time, which may be specified by the modeler as a definite quantity or a random quantity within a range. However, because WFIs can be interrupted and never resumed, when an activity will actually finish cannot be predicted from its start time.

2.1.6 Composite Activities

Composite activities are user-defined detailed activities that are decomposed into sub-activities. The lowest activity in a composite activity is always either a primitive-, predefined-, or a Java activity. A composite activity describes what an agent does while it is "in" the activity. A composite activity can be interrupted, when one of its lower-level activities gets interrupted, or one of its contained workframes gets interrupted.

A composite activity requires one or more workframes to execute. Since activities are called within the do-part of a workframe, each is performed at a certain time within the workframe. The body of a workframe has a top-down, left-to-right execution sequence (see Fig. 4). Preference or relative priority of workframes can be modeled by grouping them into ordered composite activities. The workframes within a composite activity, however, can be performed in any order depending on when their preconditions are satisfied. In this way, workframes can explicitly control executions of composite activities, and execution of workframes depend not on their order, but on the satisfiability of their preconditions and the priorities of their activities.

```
composite-activity ::=
composite-activity activity-name( { param-decl [ , param-decl ]* } )
{
   { display : literal-string ; }
   { priority : [ unsigned | param-name ] ; }
   { end_condition : [ detectable | nowork ] ; }
   { detectable-decl }
   { activities }
   { workframes }
   { thoughtframes }
}
```

A composite activity can terminate in the following three ways:

1. A composite activity terminates whenever the WFR in which it is executed terminates, due to a WFR detectable of type complete or abort (see section 2.1.7).

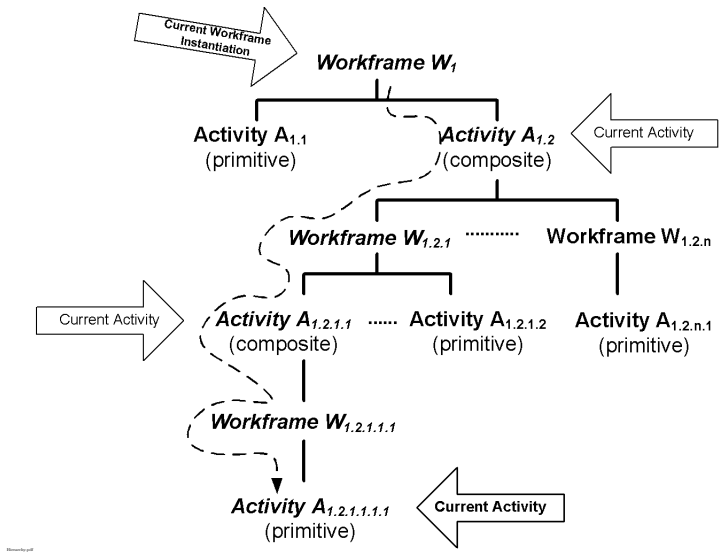

Fig. 4 Workframe-Activity Hierarchy

2. A composite activity terminates immediately whenever its *end_condition* is declared to be a *detectable*, and a detectable with an *end_activity* action is declared within the composite activity and is activated.
3. A composite activity terminates when the modeler has defined the *end_condition* to be *nowork*, and there is no workframe in the composite activity that is either available or is the current workframe being executed.

During the execution of a composite activity, the engine continuously checks whether the agent has received a belief that matches any detectables.

2.1.7 Detectables

A detectable is a language construct used within an activity or WFR by which an agent may notice facts in the world. The noticing of the fact may cause the agent to continue, impasse, stop, or to finish the activity or WFR. Detectables are used for detecting facts and reacting to beliefs that are created based on the fact detection, while the agent is executing a WFR and an activity. It allows for modeling contextual awareness of the agent, which means that the agent only detects relevant facts for the current activity. This enables modeling of reactive behavior that is constrained by the agent's activity, i.e. what it is currently doing.

```
detectable ::=
detectable detectable-name {
   { when ( [ whenever | unsigned ] ) }
      detect ( ( resultcomparison ) { , detect-certainty } )
   { then [ workframe-det | activity-det ] } ;
}

detect-certainty ::= dc : unsigned

workframe-det ::= continue | impasse | abort | complete

activity-det ::= end_activity
```

A detectable is defined in a workframe (see workframe-det) or in a composite activity (see activity-det). A detectable is active while a workframe/activity is active. It is used for noticing states of the world, and being able to act upon those.

When-condition: For each detectable it needs to be specified when the detectable is active in the workframe or activity. There are two options:

- *Whenever*: This means that, when the workframe or activity in which the detectable is defined is active, the detectable is checked every time a new fact is asserted and also every time a new belief is asserted for the agent.
- *At a specified percent completion time*: An unsigned integer specifies that, when the workframe or activity in which the detectable is defined is active, the detectable needs to be checked at the percentage completion of the workframe or activity, varying from 0% (start) to 100% (end) completion. These kind of detectables are only checked once.

The default, if the when-condition is not specified, is *whenever*.

Detect-condition: There is a two-step process for the activation of detactables:

1. Fact Detection: This step is subdivided into two parts:
   a. Notice fact: Facts are matched on only the left-hand side of the detect condition.
   b. Create belief: fact becomes a belief, regardless of the right-hand side of the detect condition.
2. Trigger action: Execute the detectable-type action, when the detect condition is *true*, based solely on the existence of the belief(s) in the agent's belief-set that match the detect-condition.

Detect-certainty: The fact detect-certainty is a number ranging from 0 to 100 and represents the probability that the fact(s) will be detected, based on the detect-condition. A detect-certainty of 0% means that the fact(s) will never be detected. A detect-certainty of 100% means that a fact will always be detected, based on the left-hand side of the detect-condition in the detectable. Any detect-certainty in between means that the fact(s) will be detected given a probability with a normal distribution between 0 and 1. The default, if the detect-certainty is not specified, is 100. Note, however, that the detect-certainty has no influence on the agent getting the fact as a belief by other means then the detectable. For instance, the agent can get the belief(s) through reasoning (TFRs), or through a communication with other agents. In that case, the detectable action can fire, regardless of the fact detect-certainty

Detectable-action: An agent never reacts directly to the facts it detects, only to the beliefs that match the detect condition. A detectable merely causes a belief to be asserted as part of its fact detection. The action portion or trigger is activated by matching against the beliefs; i.e. it is possible to trigger the action of a detectable by only asserting a belief without the same fact being present in the world state. Detection and triggers are evaluated independent of one another. There are five different detectable actions possible, the first four are only valid for detectables in WFRs and the last one is only valid for detectables in composite activities:

- *continue*: This action has no effect, and only used for having agents detect facts and turn them into beliefs (i.e. only noticing, and no action). If no detectable-action is specified, this is the default action.
- *impasse*: This action impasses the workframe until the impasse is resolved. The detect-condition is the impasse condition. What this means is that as long as the agent has a belief that matches the detect-condition in the impasse-detectable, the WFI is impassed. As soon as the agent gets a belief that makes the detect-condition false, the impasse is resolved and the WFI becomes interrupted, vying for becoming the current WFI again, based on priority.
- *abort*: This action terminates the workframe immediately.
- *complete*: This action terminates the current activity and skips all remaining activities in the workframe, but still executes all remaining consequences.
- *end_activity*: This action is only meaningful when used in a detectable for a composite activities. It causes the composite activity to be ended immediately, regardless if there are workframes or thoughtframes in it that are or can become active.

2.1.8 Classes and Objects

In Brahms, agents are intentional. However we also want to be able to describe artifacts in the real world as action-oriented systems, but unintentional at the same time. We model such an artifact as an object. An example of an object in Brahms is a fax machine. If we want to describe the behavior of a fax machine, we could argue that we could describe a fax machine as an intentional agent. However, in the real world we would never ascribe intention to the actions of a fax machine. A fax machine mainly reacts to facts in the world; such as a person pushing the start button on the fax machine that makes the fax machine start faxing a document. Since in Brahms we are interested in describing the world with its animate and inanimate entities, we want the capability to make a difference between an intentional entity (an agent), like a person, and an unintentional entity (an object) like a fax machine.

An object, in Brahms, is a construct that generally represents an artifact or data. Objects could be data objects (e.g. a database record), inanimate objects (e.g. a table) or computational objects (e.g. a computer system). The key properties of objects are facts, beliefs, thoughtframes, workframes and activities, which together represent the state and causal behaviors of objects. Some objects may have internal states, such as information in a computer, that are modeled as beliefs. Other artifact states such as the fact that a phone is off the hook are facts about the artifact in the world.

Classes: Classes in Brahms represent an abstraction of one or more object instances. The concept of a class in Brahms is similar to the concept of a template or class in object-oriented programming (Rumbaugh et al. 1998). It defines the thoughtframes, activities and workframes, initial-facts and initial-beliefs for instances of that class (objects). Brahms does not allow multiple inheritance for objects.

Objects: Objects in Brahms have all of the elements that an agent has, plus two additional elements; conceptual-object membership and resource. Furthermore, instead of having group-membership (MEMBER-OF) relationships, an object can have class-inheritance (IS-A) relationships with classes.

Objects can have a belief-set. Beliefs in an object can model data encoded within the object. Beliefs can be seen as the information that an object carries, such as the text written on a piece of paper.

Objects can act on Facts or on Beliefs: By default, WFR preconditions in objects match on facts, not on the beliefs inside the object. TFR preconditions in objects match on beliefs inside the object, similar to agents. The problem becomes how to do data processing within an object? In other words, how to conclude a belief in a TFR that triggers an activity in a WFR? Concluding a fact in a TFR is not allowed and since a WFR in an object

can only react to facts this seems impossible. To solve this problem, WFRs in objects can specify a workframe-type:

- Factframe: Preconditions match on facts. This is the default type in objects.
- Dataframe: Preconditions match on beliefs

class ::=
class class-name { **extends** class-name [**,** class-name]* }
{
  { class-facets }
}

object ::=
object object-name **instanceof** class-name
  {**partof** conceptual-object-name [**,** conceptual-object-name]*}
{
  { class-facets }
}

class-facets ::=
{ **display :** literal-string **;** }
{ **cost :** number **;** }
{ **time_unit :** number **;** }
{ **resource :** truth-value **;** }
{ **icon :** literal-string **;** }
{ attributes }
{ relations }
{ initial-beliefs }
{ initial-facts }
{ activities }
{ workframes }
{ thoughtframes}

2.1.9 Communications

In Brahms communication between agents and objects is done by communicating beliefs. The communication of beliefs is done with a *communication* activity that transfers beliefs to/from one agent to one or several other agents, or to/from an (information carrier) object. A communication activity is used to model different types of communications that can be observed in the world.

Examples are: face to face conversations, reading or writing a document, or data entered into computers.

An agent or object has to have the belief before it can communicate (i.e. tell) the belief to another agent or object. The recipient agent or object will have its original beliefs overwritten with the communicated beliefs. The syntax of a communication activity is as follows:

```
communicate-activity ::=
communicate activity-name ( { param-decl [ , param-decl ]* } )
{
   { display : literal-string ; }
   { priority : [ unsigned | param-name ] ; }
   { random : [ truth-value | param-name ] ; }
   { min_duration : [ unsigned | param-name ] ; }
   { max_duration : [ unsigned | param-name ] ; }
   { resources }
   { type : [ phone | fax | email | face2face | terminal |
            pager | none | param-name ] ; }
   with : [ [ agent-name | object-name | param-name ]
            [ , [ agent-name |object-name | param-name ] ]* ;
   about : transfer-definition [ , transfer-definition ]* ;
   { when : [ start | end | param-name ] ; }
}

transfer-definition ::=
transfer-action ( communicative-act | resultcomparison )

transfer-action ::= send | receive
communicative-act ::= object-name | param-name
```

The direction of communication is defined by the transfer-definition:

- send: The agent communicated with will *always* receive the belief.
- receive: The agent from which the beliefs are received does not know it is being communicated with. Also, that agent needs to have the belief.
- Transfer of beliefs happens either at the start or at the end of the activity.

Another way of communication is using the *Communicator library*. The Communicator library can be used to send and receive FIPA (Foundation of Intelligent Physical Agents)Foundation of Intelligent Physical Agents communicative acts[3]. The Communicator library implements external activities for agents to communicate with other agents through communicative acts. These activities can be used to create, read, manipulate, retract, and send communicative-act objects.

[3] http://www.fipa.org/specs/fipa00037/index.html

The library defines the class *CommunicativeAct*. CommunicativeAct objects are serializable objects that are communicated between agents. A CommunicativeAct object needs an *envelope* with the address information (from, to, date, ...) and transport hints, and a *payload* for the message content and content properties according to the FIPA definition:

```
class CommunicativeAct extends SerializableObject {
   attributes:
      public map envelope;
      public map payload;
      public Exception raisedException;
}//CommunicativeAct
```

Serializable objects are objects that can be efficiently communicated between agents either running toghether in one BVM, or distributed over the network between multiple BVMs. The BVM uses an efficient dedicated protocol for communicating serializable objects between agents distributed over a network.

2.2 Semantics and Verification

In this section we briefly discuss the preciseness, expressiveness, and the verifiability of the Brahms language.

Brahms is a strongly typed compiled BDI agent language. As the reader has seen in the previous sections, Brahms has a well-defined grammar specified in EBNF and a clear, but not formally defined, semantics.

The Brahms compiler's lexer and parser are generated using JavaCC[4]. JavaCC generates a top-down (recursive descent) parser. The parser and lexer work together to parse the source files and to identify the appropriate tokens for use by the parser. The parser generates a parse tree in the first pass which is also where syntactic analysis takes place. In the second pass the compiler performs the semantic analysis which includes type checking. In the last pass the compiler generates an internal object model with the compiled code and uses that model to then generate compiled code for the BVM. Brahms' compiled code is generated as XML-based Brahms compiled concept files that have the extension '.bcc'. At the moment, the compiler does not have an optimizer.

Brahms has a clear and precise semantics that is part of the second pass of the compiler. Currently, there is work in progress to be able to make formal verification of Brahms models possible. The approach taken is to re-generate

[4] https://javacc.dev.java.net/doc/features.html

a Brahms model into a Jason program [2]. Since *Jason* is a pure BDI language with a clear defined semantics a Brahms model will be verifiable as a Jason program [10].

2.3 Software Engineering Issues

In this section we discuss some of the software engineering and programming language principles, such as abstraction, inheritance, modularity, overloading, overriding, information hiding, error handling, generic programming, etc., that have been considered or adopted within design of the Brahms language.

2.3.1 Agent-oriented versus object-oriented.

With belief-based agent-oriented languages, such as Brahms, people often confuse groups and agents with classes and objects in an object-oriented language (OOP), such as Java. Object-oriented programming includes features such as encapsulation, polymorphism, and inheritance, enabling the notion of information-hiding.

In OOP encapsulation is created by definition of member attributes in a class. The purpose of encapsulation of information is to hide the physical implementation of data, so that if it is changed, the change is restricted to the class definition. Encapsulation takes a different form in belief-based agents. In belief-based agent-oriented programming, the issue is not about hiding the physical data storage definition, but rather hiding the internal belief-state of an agent, so that the agent can use it to act upon and change it independent from other agents. Agents are autonomous behavioral entities, whereas objects are simply encapsulated data and function containers that can be accessed by others, through well-defined interfaces. Agents can only interact through the use of communication protocols.

Polymorphism and inheritance in OOP is the abstraction of similar types and functionalities of objects into an inheritance hierarchy of abstract types to more specific types. It is about inheritance of similar properties and functions, as well as about function or method overloading, where we redefine a function or method at a more specific level if needed, so that others can interface with objects of similar types in the hierarchy in similar ways. Although these capabilities are useful in software engineering in general, and are thus also useful also in agent-oriented programming, they are not the key differences between agent-oriented and object-oriented. The Brahms language also has inheritance of properties, etc, in groups. Brahms also has polymorphism in the form of activity overloading (see below). However, the notion of groups and inheritance in Brahms is not about abstract class types, but about group membership. In other words, the behavior of agents is defined by the different

groups an agent belongs to. These so-called *communities of practice* define what the agent knows and when and how it will behave as a member of that group, independently from others. This is because the agent is a member of that group and not because the agent is of a particular type. These differences are subtle, but very important in distinguishing belief-based agent-oriented programming from object-oriented programming.

2.3.2 Activity overriding and overloading

The Brahms language allows polymorphism by providing both activitiy overriding and overloading (ad-hoc polymorphism). This makes it possible to write a workframe in a high-level group that is inherited by subgroups that override/overload an activity referenced in the workframe. In the example below the subgroups *LunarRobot* and *LunarAstronaut* each override the activity *LunarActivity* as a composite activity. Both groups also inherit the WFR *wf_PerformLunarActivity* from the group *LunarExplorer*. This WFR will call the overridden activity for each subgroup, and thus lunar robot and lunar astronaut agents execute their lunar activity appropriately.

```
group LunarExplorer {
   activities:
      primitive_activity LunarActivity(map input) {
         max_duration: 10;
      }

   workframes:
      workframe wf_PerformLunarActivity {
         when (...)
         do {
            LunarActivity(map input);
         }
      }
}

group LunarRobot memberof LunarExplorer {
   activities:
      composite_activity LunarActivity(map input) {
         ...
      }
}
```

```
   group LunarAstronaut memberof LunarExplorer {
   activities:
      composite_activity LunarActivity(map input) {
         ...
      }
}
```

2.3.3 Java integration

The ultimate objective is to completely integrate Brahms with the Java language, which would allow the Brahms modeler/programmer to write pure Java code as part of a Brahms model/program. However, at the moment this is not possible yet. Brahms currently has two ways of interfacing with Java using the Brahms JAPI:

- **Java activities** are primitive activities written in Java. To write a Java activity you will need to define the Java activity in the Brahms model, and implement the activity by writing the activity using the Brahms JAPI. To do this you need to create a Java class that extends from the *AbstractExternalActivity* abstract class in the JAPI. The AbstractExternalActivity is an interface for external activities implemented in Java, called by Brahms Java activities. The external activity can perform any Java action. This abstract class provides access to parameters passed to Brahms Java activities, and allows for adding bindings to unbound variables passed to Brahms java activities through parameters. Most importantly, you need to define the *doActivity* method to execute the Java activity.
- **External agents** are Brahms agents written in the Java programming language. To write an external agent you will need to define the agent as an external agent in the Brahms model and then write the external agent in Java using the Brahms JAPI. To do this you need to create a Java class that extends from the *AbstractExternalAgent* abstract class in the JAPI. The AbstractExternalAgent is an interface for external agents implemented in Java, loaded into the virtual machine to participate in a Brahms simulation or real-time agent execution. The external agent can perform any Java action. This abstract implementation provides access to the concepts loaded in the virtual machine and the world state to allow for communications with these concepts and to allow for world state changes to be triggered by this agent.
- **Java objects** can be referenced using Java class types as Brahms attribute types. This allows referencing Java objects from within the Brahms language.

2.4 Other features of the language: geography model

In Brahms agents and objects can be situated in a model of the physical world. The world is represented independent of the capability of agents. An *areadefinition* is used for defining a class of *area* instances used for representing geographical locations. Areadefinitions are similar to classes in their use. Examples of areadefinitions are "Building", and "City". An example of an area is "Berkeley". Areas can be decomposed into sub-areas. For example, a building can be decomposed into one or more floors. A floor can be decomposed into offices. The decomposition can be modeled using the PART-OF relationship. A *path* connects two areas and represents a route that can be taken by an agent or object to travel from one area to another. The modeler may specify distance as the time it takes to move from area1 to area2 via the path. The BVM automatically generates location facts and beliefs for agents and objects moving from one area to another.

Agents and objects can be located in an initial location (i.e. areas). Agents and objects can move to and from areas. When agents and/or objects come into a location, the BVM automatically creates a location fact *(agent.location = <current-area>)*. Agents always know where they are and they notice other agents and objects. When agents come into a location, the BVM automatically gives the agent a belief about its new location (same as the location fact), and also gives the agent a location belief for all other agents and objects currently in that location. When an agent or object leaves a location, the location fact and beliefs are retracted from all agents that are in that location the moment the agent or object leaves. Agents and objects can carry (through the *containment* relation) other agents and objects. Contained agents and objects are not "noticed" until they are put into the area by the containing agent or object.

areadef ::=
areadef areadef-name { areadef-inheritance }
{
  { **display :** literal-string **;** }
  { **icon :** literal-string **;** }
  { attributes }
  { relations }
  { initial-facts }
}
areadef-inheritance ::= **extends** areadef-name[**,** areadef-name]*

```
area ::=
area area-name instanceof areadef-name { partof area-name }
{
   { display : literal-string ; }
   { icon : literal-string ; }
   { attributes }
   { relations }
   { initial-facts }
}
```

The geography model is a conceptual model, meaning that it does not represent the geography as a graphical three-dimentional model. Areas can have attributes and relations, and define initial facts. Facts about areas can represent the state of a location, e.g. the temperature in an area. The BVM automatically generates facts about the 'partof' relationships in the geography. Agents can detect these facts and thus learn (i.e. get beliefs) about the areas in their environment.

The example geography model below, defines a simple geography for the University of Berkeley in Berkeley, CA. This model defines the university buildings SouthHall and SpraulHall, where two students (Kim and Alex) are initially located. Furthermore, the model defines two bank branches and two restaurants in the city of Berkeley.

```
// Area defintions
areadef University extends BaseAreaDef { }
areadef UniversityHall extends Building { }
areadef BankBranch extends Building { }
areadef Restaurant extends Building { }

// ATM World
area AtmGeography instanceof World { }

// Berkeley
area Berkeley instanceof City partof AtmGeography { }

// inside Berkeley
area UCB instanceof University partof Berkeley { }
area SouthHall instanceof UniversityHall partof UCB { }
area SpraulHall instanceof UniversityHall partof UCB { }
area Telegraph_Av_113 instanceof BankBranch partof Berkeley { }
area Bancroft_Av_77 instanceof BankBranch partof Berkeley { }
area Telegraph_Av_2405 instanceof Restaurant partof Berkeley { }
```

```
area Telegraph_Av_2134 instanceof Restaurant partof Berkeley { }

// initial location
agent Kim_Agent memberof Student {
   location: SouthHall;

agent Alex_Agent memberof Student {
   location: SouthHall;
}
```

Agents and objects can move within the geography model. To have an agent move from one location to another you can do the following:

- Use a *move(to_location)* activity in a WFR.
- Specify the duration of the move in clock-ticks. By default the duration is zero, unless,
- There is a define a *Path* object between two areas. A path defines a duration to move from area1 to area2. A path object defines a bi-directional path.
- The BVM creates and retracts location facts and beliefs automatically.
- Agents in an area will detect arrivals and departures of other agents and objects (by the creation and retraction of location beliefs for agents located in the area).
- In the move activity you can specify (sub-)area arrival and departure detection for the agents.
- The BVM calculates the shortest path between areas, given a geography model.
- Contained objects and agents move with the agent or object that is moving. However, they will not get noticed by other agents, until they are placed in the destination area by the agent (using a put activity).

3 Platform

The Brahms Agent Environment (BAE) is a collection of tools for developing complex models of agents, objects and areas for the purpose of simulating work practice, or for developing MAS solutions to support the people that are part of a work system. The BAE also supports the development of distributed agent-based solutions in support of an organization's workflow. In this section we describe the BAE tools, available documentation, standards compliance, and interoperability and portability features of Brahms.

3.1 Available tools and documentation

The Brahms tools and documentation that are included in the BAE are:

- **The Brahms Compiler** (BC). The BC is a compiler for the Brahms language. The compiler compiles .b source files into .bcc byte-code files.
- **The Brahms Virtual Machine** (BVM. The BVM is both a simulation engine for simulating Brahms models, and a MAS execution environment for real-time agents.
- **The Composer**. The Composer is a dedicated integrated development environment (IDE) for Brahms. It provides a project editor, model editors, a source code editor, and several post-execution displays. The modeler can both compile and run a model from within the Composer. The Composer is a useful tool for those who use Brahms for modeling and simulating work practice.
- **The Brahms Eclipse Plugin**. The Brahms Eclipse Plugin is a plugin for the Eclipse development environment. The plugin is useful for those who use Brahms as a MAS development tool, and also develop and integrate with Java code.
- **The AgentViewer**. The AgentViewer is a post-execution event timeline viewer for agent and object beliefs, workframes, activities and thoughtframes, as well as inter-agent and -object communications. The AgentViewer is a kind of debugging tool, although it is used after executing the model and is not an interactive debugging environment. During the execution of the model (either in simulation mode or in real-time mode), all events are stored by the event-logger of the BVM in an ascii-formated history file. Using the AgentViewer application, this history file can be parsed into a MySQL database that the AgentViewer uses to generate a TimeLine view.
- **The Communication Display**. The Communication Displaycommunication!display provides a spring diagram of the agent and object communications. It shows the to and from communications, as well as the number of beliefs and/or CommunicativeActs communicated. The Communication Display is integrated with the AgentViewer application and uses the same MySQL database to retrieve its data.
- **Documentation**. Brahms documentation is provided on the Brahms website[5]. The documentation available via this website can be easily accessed through Quick Links on the home page.

[5] http://www.agentisolutions.com

3.2 Standards compliance, interoperability and portability

All the tools that are part of the BAE are written in the Java language and require the Java Runtime Environment version 6. Currently, the BAE is supported on Windows 2000/XP, Linux, OS X, and Solaris. The AgentViewer and Communication Display require MySQL (either version 4.1, 5.0.51 or later) to be installed.

In its simulation mode, agents cannot be distributed over multiple BVMs, and the BAE uses a Brahms native communication protocol. In simulation as well as in real-time execution mode Brahms agents can be integrated with other agent systems through the use of external agents (see section 2.1.1).

In its real-time execution mode, the BAE uses a custom architecture for its communication, naming/directory service and agent-life cycle management. This custom architecure is called the Collaborative Infrastructure (CI). It is a Java-based communication framework for agent-based communications loosely based on FIPA. The CI is an open agent communication framework and has a Java and C++ API, allowing Brahms agents to be integrated with Java, C++ and C programs that also use the CI as their agent communications architecture. The CI includes a directory service for registering and finding CI agents. The CI uses a Brahms native protocol using sockets as its transport layer for communication between distributed agents.

3.3 Other features of the platfrom

The BAE has been extensively tested over the years in both simulation mode and in real-time execution mode. We do not have any specific performance metrics available, but BAE version 1.2.7 is currently running the OCAMS application 24x7 in NASA's Mission Control (see section 4).

4 Applications supported by the language and/or the platform

In this section we describe a number of the most prominent applications of Brahms. We categorize Brahms applications into those that primarily use Brahms as a simulation environment, and those that use Brahms as a real-time (distributed) MAS execution environment.

Brahms has been used in many simulation projects at the NYNEX and Bell Atlantic phone companies [6], NASA (see below), the Universities of Twente [4], Amsterdam [15] and Utrecht [16], and by several research orga-

nizations throughout the world. At NASA, Brahms has been used to develop a distributed multi-agent human-robot exploration system (see below), simulation of collaborative traffic flow management for future concepts of the US National Airspace [28, 29], and most recently to simulate and implement an intelligent workflow application for NASA's Mission Control (see below). Following is a description of a number of NASA simulation and MAS applications developed with Brahms.

Apollo EVA Simulations
From 1998 until 2001, Sierhuis developed Brahms simulations of the Apollo 12, 14, 15 & 16 Apollo Lunar Surface Exploration Package offload and deployment extra-vehicular activities (EVA) on the Moon [18].

Day-in-the-life Simulation onboard the ISS
The International Space Station (ISS) is one the most complex projects ever, with numerous interdependent constraints affecting productivity and crew safety. This requires planning years before crew expeditions, and the use of sophisticated scheduling tools. We presented an agent-based model and simulation of the activities and work practices of astronauts onboard the ISS based on an agent-oriented approach. Between 2001 and 2003 we developed a Brahms simulation model of a day-in-the-life onboard of the ISS Alpha crew [1, 20].

MER Mission Operations Simulation
Mission operations systems for space missions are comprised of a complex network of human organizations, information and deep-space network systems and spacecraft hardware. Similar to the operations within traditional organizations, one of the problems in mission operations is the management of the mission information systems related to the human work processes. Brahms was used to model and simulate NASA's Mars Exploration Rover (MER) mission operation work process [24, 21, 17].

Shuttle Mission Operations Simulation
In this project we used the Brahms environment to model and simulate JSC's Mission Operations Directorate (MOD) organization, and the work performed during the Shuttle pre-launch through docking phases with the International Space Station [23]. The output of the simulation is a detailed time line of the flight controllers activities and communication and metrics of different work activity and workload.

Mobile Agents MAS
We have developed and tested an advanced EVA communications and computing system to increase astronaut self-reliance and safety, reducing dependence on continuous monitoring and advising from mission control on Earth. This system, called the Mobile Agents Architecture (MAA), is voice con-

trolled and provides information verbally to the astronauts through programs called personal agents. The system partly automates the role of CapCom in Apollo including monitoring and managing EVA navigation, scheduling, equipment deployment, telemetry, health tracking, and scientific data collection [7, 12].

OCA ISS Flight Control Simulation and Intelligent Workflow MAS The OCA Mirroring System (OCAMS) is a practical engineering application of multi-agent systems technology, involving redesign of the tools and practices in a complex, distributed system. OCAMS is designed to assist flight controllers in managing interactions with the file system onboard the ISS. The simulation-to-implementation engineering methodology combines ethnography, participatory design, multiagent simulation, and agent-based systems integration [9]. The OCAMS system is currently deployed in the ISS Mission Control at NASA Johnson Space Center (JSC) in Houston. OCAMS supports the ISS OCA officer 24x7 in their uplinking, downlinking, mirroring, archiving and distributing of files to and from the ISS.

5 Final Remarks

In this chapter we described the Brahms language and environment. Brahms has been in development as an agent simulation language since 1992, and has matured to a full-fledged AOL. Brahms is well-tested and stable. This is proven by the fact that Brahms is used in NASA Mission Control for the development of operational multi-agent systems.

Brahms contributes to the multi-agent languages community in at least two ways: 1) Brahms is both an agent-based simulation language and a MAS development environment. This allowed us to develop a *from simulation to implementation* agent-oriented software engineer methodology that has been applied at NASA [22], 2) the Brahms language was the first AOL language that integrated a BDI architecture with a reactive activity-based subsumption architecture, all the way back to the early nineties [19].

In the near future, we are working on integrating the Brahms and the Java language more. The next release of Brahms will have a seamless integration of Java objects. Preconditions in workframes and thoughtframes will allow matching on Java object members, without them becoming beliefs. This will optimize the use of Java objects from within the Brahms language. We are also in the process of adding the use of lists in preconditions. Our ultimate objective is to completely combine the Brahms and Java languages, allowing the Brahms programmer to write Java code within their Brahms program, without the need for using a Java API. Brahms will support both Java objects, methods and agent activities, and the ability to call Java ob-

ject methods directly from a workframe. This will combine the benefit of both object-oriented and agent-oriented programming in one language.

The BAE has a stable and free release available for research purposes from the Brahms website at http://www.agentisolutions.com. Brahms does not (as of now) provide an open-source distribution, but is available for free download under the Brahms Research license agreement.

Acknowledgements Brahms development started in 1992 as a collaboration between the former R&D center of the then NYNEX corporation (NYNEX Science and Technology) and the former Institute for Research on Learning (IRL), a spinoff of Xerox PARC. Since 1998, Brahms has been developed and used by the Work Systems Design and Evaluation group in NASA Ames' Intelligent Systems division. We thank all our NYNEX, IRL and NASA funders over the past sixteen years. In particular, we like to thank Jim Euchner (NYNEX) and Mike Shafto (NASA) for their continued support of Brahms and our Brahms research team.

Appendix (Language Summary)

1(a) Brahms agents include mental attitudes, deliberation, adaptation, social abilities, and reactive as well as cognitive-based behaviour.

1(b) Brahms provides two types of communication capabilities: 1) a built-in belief communication activity, 2) a FIPA-based Communication Library for sending/receiving Communicative Acts.

1(c) Brahms, currently, does not support any mobility service.

1(d) Brahms is easy to learn. Brahms users include not only computer scientists, but also cognitive scientists, psychologists, economists and even an architect.

1(e) Brahms has a precise syntax and semantics. The syntax is specified in EBNF. The semantics is currently not formalized, but is descibed as part of the Brahms language document.

1(f) Brahms is suitable for the development of agent-based work practice, organizational, work flow and cognitive simulations, as well as the implementation of a variety of agent-oriented programs and applications.

1(g) Brahms allows for extention and definition of new language components through the definition of Java activities using the JAPI.

1(h) Although the Brahms semantics is currently not formalized, Brahms does allow for a clear path for the (formal) verification of programs (also called models).

1(i) Software Engineering and Programming Language principles, such as abstraction, inheritance, modularity, overloading, information hiding, error handling, generic programming, have been adopted within the design of the Brahms language.

1(j).i Brahms can be integrated with the Java programming language, using the JAPI, in both simulation and real-time execution mode.

1(j).ii In real-time execution mode, Brahms agents can communicate with other general Java or C++ agents, using the agent Collaborative Infrastructure (CI).

2(a).i The Brahms website (http://wwww.agentisolutions.com) provides Java docs of the JAPI, a detailed Brahms language specification, both EBNF syntacs and semantics, a Brahms tutorial that includes excercises and documentation, and a web-based discussion forum. The BAE installation is done with an easy to use installation wizard. The Brahms website includes a readme file with some additional information on configuring MySQL.

2(a).ii Brahms requires the Java Runtime Environment (version 6), and is currently supported on Windows 2000/XP, Linux, OS X, Solaris. The AgentViewer tool requires MySQL 4.1, 5.1.51 or later to be installed.

2(b) In real-time execution mode, Brahms uses a custom agent collaborative infrastructure (CI). Both Brahms and the CI use Communicative Acts, loosely based on FIPA. The CI provides a custom naming/directory service and custom agent life-cycle management for managing the starting and stopping of distributed agents running in one or more Brahms Virtual Machines (BVMs).

2(c) Brahms is not Open Source, but does allow for being extended with additional functionality. Using the JAPI, it is possible to add new services, external agents to interact with, external systems, and java activities to add additional activity behaviors.

2(d).i Brahms logs history events that are used post-execution in the AgentViewer tool, for the display of all agent events (new beliefs, workframe/activity and thoughtframe execution, movement in the geography model, and communication).

2(d).ii Brahms is installed as the Brahms Agent Environment using an easy install wizard. The Brahms web-site provides a web-based discussion forum through which the Brahms developers can be contacted http://www.agentisolutions.com/cgi-bin/Ultimate.cgi). There is no specific maintenance provided to external users, however, the Brahms team is regularly updating the BAE with new releases for download. Any bugs in the BAE that are reported will be resolved in the next release, or provided as updates on the website.

2(d).iii The BAE does not include specific tools for management or real-time monitoring. However, there are two separate IDEs provided: 1) the Composer is an IDE through which Brahms models can be designed, implemented, compiled, and executed, 2) there is also a Brahms Eclipse Plugin. The Composer includes the Agentviewer, which is can be used as a post-execution debugger. Both the compiler and the BVM has configuration files that can be set outside the Composer in a text editor, or within the Composer using property editors.

2(e) Existing tools and applications integrated are JacORB CORBA, E-mail Client, FTP Client, IM Client (Jabber), GPS, Biosensors, digital

cameras, MS Excel (J-integra), MS Word (J-integra), RIALIST speech dialoque system, LEGACI astronaut metabolic calculation algortithms, Compendium.

2(f) Brahms requires Java Runtime Environment (version 6) and MySQL version 4.1, 5.1.51 or later

2(g).i Currently, there are no specific performance metrics available. However, depending on the complexity of the agents, one BVM can easily simulate 150 Brahms agents and objects. In distributed real-time execution mode, the number of BVMs is unlimited, and depending on the complexity of the agents, each BVM can easily run 10 to 20 agents.

2(g).ii The BAE is a thoroughly tested and stable agent environment. It is used to execute a MAS application 24x7 in NASA's International Space Station Mission Control. A free release is available for research purposes only. Brahms is not Open Source.

2(h).i The new version of Brahms will support open multi-agent systems and heterogeneous agents through the use of the Collaborative Infrastructure (CI). Other ways are to develop proxy agents using the external agent JAPI.

2(h).ii The BAE, through the CI, provides distributed control. The Brahms language provides hierarchical structure of agents. However, the directory service for distributed agents does not.

2(h).iii The BAE provides a Communicative Acts library and templates for programming multi-agent systems (both in Brahms and Java).

3(a) Brahms has been used to develop research, real-world and industrial applications both for simulation and for MAS development. The most prominent application is the OCA Mirroring System (OCAMS) in NASA's International Space Station Mission Control and Mobile Agents, a planetary exploration MAS workflow framework for robots and astronauts.

3(b) Brahms is a domain-independent simulation and MAS language. It can be used for agent-based simulation, as well as for MAS development and execution. Brahms is not geared towards any specific domain, but has mostly been used in the space mission operations and exploration domain. It is particularly useful for simulating work practice and organizations, and developing intelligent agent-based workflow services and applications.

References

1. Acquisti, A., Sierhuis, M., Clancey, W.J., Bradshaw, J.M.: Agent based modeling of collaboration and work practices onboard the international space station. In: 11th Computer-Generated Forces and Behavior Representation Conference, pp. p.181–188. Orlando, Fl. (2002)
2. Bordini, R.H., Hbner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Series in Agent Technology. Wiley (2007)

3. Brownston, L., Farrell, R., Kant, E., Martin, N.: Programming Expert Systems in OPS5. Addison-Wesley (1985)
4. Bruinsma, G., de Hoog, R.: Exploring protocols for multidisciplinary disaster response using adaptive workflow simulation. In: B.V.d. Walle, M. M. Turoff (eds.) International Conference on Information System for Crisis Response and Management (ISCRAM). Newark, New Jersey (2006)
5. Clancey, W.: Heuristic classification. Artificial Intelligence **27**(3), 289–350 (1985)
6. Clancey, W., Sachs, P., Sierhuis, M., Hoof, R.v.: Brahms: Simulating practice for work systems design. International Journal on Human-Computer Studies **49**, 831–865 (1998)
7. Clancey, W., Sierhuis, M., Alena, R., Berrios, D., Dowding, J., Graham, J., Tyree, K., Hirsh, R., Garry, W., Semple, A., Buckingham Shum, S., Shadbolt, N., Rupert, S.: Automating capcom using mobile agents and robotic assistants (2007)
8. Clancey, W.J.: Simulating activities: Relating motives, deliberation, and attentive coordination. Cognitive Systems Research **3**(3), 471–499 (2002)
9. Clancey, W.J., Sierhuis, M., Seah, C., Reynolds, F., Hall, T., Scott, M.: Multi-agent simulation to implementation: A practical engineering methodology for designing space flight operations. In: A. Artikis, G. O'Hare, K. Stathis, G. Vouros (eds.) The Eighth Annual International Workshop "Engineering Societies in the Agents World" (ESAW 07), vol. LNAI. Springer, London (2008)
10. Fisher, M., Pearce, E., Wooldridge, M., Sierhuis, M., Visser, W., Bordini, R.H.: Towards the verifications of human-robot teams. In: IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation. Loyola College Graduate Center, Columbia, MD (2005)
11. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**, 17–37 (1982)
12. Hirsh, R., Graham, J., Tyree, K., Sierhuis, M., Clancey, W.J.: Intelligence for human-robotic planetary surface robots. In: A.M. Howard, E.W. Tunstel (eds.) Intelligence for Space Robotics. TSI Press, Albuquerque (2006)
13. Konolige, K.: A Deduction Model of Belief. Morgan Kaufmann, San Mateo, CA (1986)
14. Leont'ev, A.N.: Activity, Consciousness and Personality. Prentice-Hall, Englewood Cliffs, NJ (1978)
15. Netten, N., Bruinsma, G., van Someren, M., de Hoog, R.: Task-adaptive information distribution for synamic collaborative emergency response. International Journal of Intelligent Control and Systems **11**(4), 238–247 (2007)
16. van Putten, B.J., Dignum, V., Sierhuis, M., Wolfe, S.R.: Opera and brahms: a symphony? In: Agent-Oriented Software Engineering (AOSE) 2008 at The Sixth International Joint Conference on Autonomous Agents & Multi-Agent Systems (AAMAS 2008), vol. Forthcoming LNCS Proceedings. Springer, Estoril, Portugal (2008)
17. Seah, C., M.Sierhuis, Clancey, W.: Multi-agent modeling and simulation approach for design and analysis of mer mission operations. In: 2005 International Conference on Human-Computer Interface Advances for Modeling and Simulation (SIMCHI'05). 2005 Western Simulation Multiconference (WMC'05), New Orleans, Louisiana (2005)
18. Sierhuis, M.: Modeling and simulating work practice; brahms: A multiagent modeling and simulation language for work system analysis and design. Ph.d. thesis, University of Amsterdam, SIKS Dissertation Series No. 2001-10 (2001)
19. Sierhuis, M.: "it's not just goals all the way down"– "it's activities all the way down". In: G.M.P. O'Hare, A. Ricci, M.J. O'Grady, O. Dikenelli (eds.) Engineering Societies in the Agents World VII, 7th International, Workshop, ESAW 2006, Dublin, Ireland, September 6-8, 2006, Revised Selected and Invited Papers, *Lecture Notes in Computer Science*, vol. LNCS 4457/2007, pp. 1–24. Springer, Dublin, Ireland (2007)
20. Sierhuis, M., Acquisti, A., Clancey, W.: Multiagent plan execution and work practice: Modeling plans and practices onboard the iss. In: 3rd International NASA Workshop on Planning and Scheduling for Space. Houston, TX (2002)

21. Sierhuis, M., Clancey, W., Seah, C., Trimble, J., Sims, M.H.: Modeling and simulation for mission operations work system design. Journal of Management Information Systems **Vol. 19**(No. 4), 85–129 (2003)
22. Sierhuis, M., Clancey, W.J., Seah, C.H.: Organization and work systems design and engineering; from simulation to implementation of multi-agent systems. In: Agent Directed Simulation, chap. 13. Wiley (To Appear)
23. Sierhuis, M., Diegelman, T.E., Seah, C., Shalin, V., Clancey, W.J., Selvin, A.M.: Agent-based simulation of shuttle mission operations. In: Agent-Directed Simulation 2007 part of the 2007 Spring Simulation Multiconference, pp. 53–60. The Society for Modeling and Simulation International, ACM/SIGSIM, Norfolk, VA (2007)
24. Sierhuis, M., Sims, M., Clancey, W., Lee, P.: Applying multiagent simulation to planetary surface operations. In: L. Chaudron (ed.) COOP'2000 workshop on Modelling Human Activity, pp. 19–28. Sophia Antipolis, France (2000)
25. Suchman, L.: Representations of work. Communications of the ACM/Special Issue 38(9) (1995)
26. Suchman, L.A.: Plans and Situated Action: The Problem of Human Machine Communication. Cambridge University Press, Cambridge, MA (1987)
27. Vygotsky, L.S.: Mind in Society: The Development of Higher Psychological Processes. Harvard University Press, Cambridge, MA (1978)
28. Wolfe, S.R., Jarvis, P.A., Enomoto, F.Y., Sierhuis, M.: Comparing route selection strategies in collaborative traffic flow management. In: Intelligent Agent Technology (IAT 2007). IEEE press, Fremont, CA, USA (2007)
29. Wolfe, S.R., Sierhuis, M., Jarvis, P.A.: To bdi, or not to bdi: Design choices in an agent-based traffic flow management simulation. In: Agent Directed Simulation 2008 held at the SpringSim Multi-Conference 2008. ACM, Ottawa, Canada (2008)

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.