BRAHMS

Language Specification

TM99-0008

Version 3.0 Final

2 December 2009

Technical Memorandum
TM99-0008

BRAHMS LANGUAGE SPECIFICATION

VERSION 3.0 – FINAL

CONTACT

Brahms Contact

Maarten Sierhuis – Project Manager (650) 604-4917

ABSTRACT

This document describes the Brahms language. It specifies the language in the Backus-Naur Form. This document specifies all language constructs in the Brahms language syntactically and semantically. Models written using this version of the language specification can only be run in the Brahms Virtual Machine.

DATE: 2 December 2009

KEYWORDS: Brahms, Language, Backus-Naur Form, Syntax, Semantics, Element Description

This document has not been reviewed by the Intellectual Property Organization.

CONTRIBUTORS

William J. Clancey

Ron van Hoof

Maarten Sierhuis

Robert Nado

APPROVED

Maarten Sierhuis Date

Project Manager

REVISION HISTORY

Version	Contact	Action	
Version 0.1 Draft 01/29/1997	Ron van Hoof 203/531-4741	New	Initial Version
Version 0.2 Draft 02/10/1997	Ron van Hoof 203/531-4741	Add	Added precondition, consequence, detectable, transfer-definition, module and made language more consistent.
Version 0.3 Draft 02/11/1997	Ron van Hoof 203/531-4741	Add, Change	Changed 'end' statements in syntax. Added descriptions and semantics for model, version information, group, and agent.
Version 0.4 Draft 02/12/1997	Ron van Hoof 203/531-4741	Add	Completed descriptions for all model elements.
Version 0.5 Draft 02/20/1997	Ron van Hoof 203/531-4741	Change	Changed syntax definitions based on comments from Maarten Sierhuis and Bill Clancey.
Version 0.6 For Review 02/27/1997	Ron van Hoof 203/531-4741	Add	Added semantics for all concepts.
Version 1.0 For Review 01/29/1997	Ron van Hoof 203/531-4741	Add, Change	Changes after review. Added geographical definitions, area-def, area and path. Added location attribute to agent and object. Made parameter lists for activities more general, removed specific parameters. Corrected punctuation where necessary. Made separation between string and symbol type. Added current limitations sections in semantics where appropriate. Added list types for parameters.
Version 1.0 Final 03/20/1997	Ron van Hoof 203/531-4741	Add, Change	Added 'type' attribute to communicate activity. Simplified activity references not to include 'move', 'create-object', 'communicate', or 'broadcast', but just the name.
Version 1.1 Draft 04/17/1997	Ron van Hoof 203/531-4741	Change	Modifications after external review. Changes opening and closing symbols from ':' and 'end' to '{' and '}'. Modified attributes and relations to be more flexible for future extensions. Changed float into double. Removed keywords goal-location, destination-location and destination-conceptual-object. Changed into location and conceptual-object.
Version 1.2 Draft 04/30/1997	Ron van Hoof 203/531-4741	Change	Added meta class keywords 'Class', 'Group', 'ConceptualClass', 'AreaDef'. Changed classtypedef to include type casting for the ID and added the new meta classes as possible class types. Added keyword 'workframe' and 'thoughtframe' to their definitions for consistency.
Version 1.3 Draft 07/29/1997	Ron van Hoof 203/531-4741	Change	Removed the type casting requirement for ID's, ID's of all objects have to be unique. Added 'not' modifier to precondition.

Version 1.4 Draft 08/05/1997	Ron van Hoof 203/531-4741	Change	Changed keywords, eliminated the use of '-'. Removed model-construct allowing for concepts to be stored in separate files. Changed import statement to make use of the ability to store concepts in separate files and to allow for better use of libraries. Removed the merging requirement, meaning that concept names have to be unique and will not automatically be merged. Removed the version information construct.
Version 1.5 Draft 09/04/1997	Ron van Hoof 203/531-4741	Add	Added ability to define an icon for the concepts.
Version 1.6 Draft 11/18/1997	Ron van Hoof 203/531-4741	Add	Added ability to define the name of the destination object in create-object activity.
Version 1.6 Final 11/18/1997	Ron van Hoof 203/531-4741	Change	Changed status to final after approval.
Version 1.7 Draft 04/12/1999	Ron van Hoof 203/531-4741	Add Change	Added package-declaration. Changed Model to Compilation unit. Created new section for model for future use. Created separate section for import declaration.
Version 1.7 For Review 04/12/1999	Ron van Hoof 203/531-4741	Change	Completed draft, changed status to For Review.
Version 1.7 Final 04/28/1999	Ron van Hoof 203/531-4741	Change	Version 1.7 has been approved without changes.
Version 1.8 Draft 11/11/1999	Ron van Hoof 203/531-4741	Change	Added ability to specify object and attribute tuple using dot notation (obj.att). Removed ability for consequences, detectables and transfer definitions to have expressions on the left hand side. Made correction in valid value comparisons. OArO and OrO specified that r could be an evaluation operator, this is incorrect and should be an equality operator.
Version 1.8 For Review 11/11/1999	Ron van Hoof 203/531-4741	Change	Completed draft, changed status to For Review.
Version 1.8 Final 11/17/1999	Ron van Hoof 203/531-4741	Change	Removed ability to specify the relational operators '>', '<', '>=', '<=' for initial beliefs/facts, consequences, detectables, and transfer definitions.
Version 2.0 Draft 12/13/1999	Ron van Hoof 203/531-4741	Add Change	Added support for class hierarchies for conceptual classes and area definitions. Added meta types Object, Area, ConceptualObject, Concept, ActiveConcept, ActiveClass, ActiveInstance, GeographyConcept, ConceptualConcept. Allowing for 'location' as the name for an attribute. Updated base model.
Version 2.0 For Review 12/13/1999	Ron van Hoof 203/531-4741	Change	Completed draft, changed status to For Review.

Version 2.0 Final 12/15/1999	Ron van Hoof 203/531-4741	Change	Changed status to final after review with no comments.
Version 2.1 Draft 03/09/2000	Ron van Hoof 203/531-4741	Change	Changed the syntax of preconditions to prevent the use of a right hand side in comparisons in case of a known or unknown modifier. Updated the semantics for the preconditions to reflect the syntax changes and to add additional notes regarding the allowable expressions in a condition that cannot be checked by the compiler but will be detected at runtime in the virtual machine.
Version 2.1 For Review 03/09/2000	Ron van Hoof 203/531-4741	Change	Completed draft, changed status to For Review.
Version 2.1 Final 03/16/2000	Ron van Hoof 203/531-4741	Change	Added an additional constraint for the preconditions stating that no nested expressions are allowed. Removed all restrictions that were added for running models in the G2 based simulation engine.
Version 2.2 Final 09/09/2000	Ron van Hoof 203/531-4741	Change	Added Java Activity and updated semantics of move, and create object activity to describe in more detail the behavior of the activity in the virtual machine.
Version 2.3 Final 05/16/2001	Ron van Hoof 203/531-4741	Change	Added External Agent.
Version 2.4 For Review 05/29/2001	Ron van Hoof 203/531-4741	Change	Added Put and Get activities to manage containment.
Version 2.4 Final 07/09/2001	Ron van Hoof 203/531-4741	Change	Marked document as final after review by the Brahms team without comments.
Version 2.5 For Review 07/09/2001	Ron van Hoof 203/531-4741	Change	Added create_agent activity to allow for dynamic creation of agents in a model.
Version 2.5 Final 07/30/2001	Ron van Hoof 203/531-4741	Change	No comments after review, marked document as final.
Version 2.6 For Review 08/06/2001	Ron van Hoof 203/531-4741	Change	Enabled the use of ‘unknown’ as a value in (initial) beliefs, facts, and all conditions as well as the use of ‘unknown’ for the truth-value of a relationship.
Version 2.6 Final 08/14/2001	Ron van Hoof 203/531-4741	Change	Added support for priorities in thoughtframes. Added section on ‘unknown values’.
Version 2.7 Final 09/04/2001	Ron van Hoof 203/531-4741	Change	Based on user feedback added the ability for transferring contained items from/to a location, object or agent using the put and get activities. Added ‘destination’ attribute to the put activity and added the ‘source’ attribute to the get activity.
Version 2.8 Final 09/19/2001	Ron van Hoof 203/531-4741	Change	Added a ‘type’ property to the definition of a workframe for a class/object. The type attribute can have as value ‘factframe’ or ‘dataframe’ and specifies whether its preconditions are to be matched against facts or beliefs.

Version 2.9 Final 12/04/2001	Ron van Hoof 203/531-4741	Change	Expanded the control for model builders in specifying to what areas a broadcast travels to and in specifying what areas can detect the arrival and departure of a moving agent. The appropriate properties have been added to the broadcast activity and move activity. The semantics of the areas has been modified to include that the virtual machine will generate facts about the area hierarchy.
Version 2.10 Final 4/09/2002	Ron van Hoof 203/531-4741	Change	Added Gesture Activity to allow agent gestures to be visualized in a three-dimensional view of a simulation.
Version 2.11 Final 6/20/2002	Ron van Hoof 203/531-4741	Change	Added Create Area Activity to allow agents to dynamically create new areas.
Version 2.12 Final 11/6/2003	Ron van Hoof 732/632-9459	Change	Now allowing detectable conditions to use the relational operators >, >=, <, <= in addition to = and !=.
Version 2.13 Final 10/26/2006	Ron van Hoof 732/632-9459	Add Change	Added support for two new attribute types, long and map. The map type is a collection type for which indices/keys are used to retrieve the attribute values. The initial statements and conditions have been modified to support these collection indices. Added a section on the use of the map collection type.
Version 2.14 Final 11/1/2006	Ron van Hoof 732/632-9459	Change	Added support for activity overloading. Names of activities no longer need to be unique within the declaration of an activities section, however their signatures do need to be unique (activity name plus the types of the argument list in the order the arguments are declared).
Version 2.15 Final 11/6/2006	Ron van Hoof 732/632-9459	Change	Added support for a <class type> variable on the left hand side of detectables allowing detectables to detect any fact with a concept on the left hand side that is type compatible with the class type concept declared in the detectable condition as: <concept class>.attribute = ?. The trigger uses this same variable to trigger the action but matched against beliefs. Added support for a new way to declare relations as part of the attributes section. Added support for allowing attributes, variables and parameters to be of a Java type, with support for Java specific import statements required to resolve Java types.
Version 2.16 Final 6/29/2007	Ron van Hoof 732/632-9459	Change	Added support for using a CommunicativeAct from the communications library as the content to be communicated to a recipient and if a CommunicativeAct is used to only allow for the reading of the contents of that CA but not the 'reading' of a CA from another agent.
Version 2.17 Final 7/16/2007	Ron van Hoof 732/632-9459	Change	Added support for the 'delete' operation.

Version 2.18 Final 5/19/2009	Robert Nado 650/604-0413	Change	Unified syntax for transfer definition conditions and detectable conditions, allowing use of class variables in transfer definition conditions. Added support for using a '?' wildcard on the right-hand side of these conditions or omitting the right-hand side entirely. The modifier of a frame precondition may now be omitted, defaulting to 'knownval' or 'known' depending on whether the precondition has a right-hand side.
Version 3.0 Final 12/2/2009	Robert Nado 650/604-0413	Add Change	Added support for : Java objects to be used as the object in an object/attribute statement. new Brahms primitive types: byte, char, short, and float local variables in the body of a workframe generalized conclude statement assignments, Java method invocations, Java constructor invocations, Java array creation and access in the body of a workframe
Actions Taken are: **New** = new document, **Add/Delete/Change** = a section or topic has been added, or deleted, or changed.			

TABLE OF CONTENTS

TABLES

REFERENCES

1. *The Java Language Specification*, Third Edition, James Gosling and Bill Joy and Guy Steele and Gilad Bracha, Sun Microsystems, Inc., http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

1. INTRODUCTION

1.1 PURPOSE

This document contains the language definition for the Brahms system. This document serves two purposes:

1. Defining the modeling capabilities of Brahms. This document gives descriptions of the modeling concepts, the formal syntax for the modeling concepts and the semantics related to the modeling concepts. This document describes the language to be used for building Brahms models.

2. A requirements specification for the parser for Brahms models. The parser will check the model for errors and will create a 'compiled' version of the model that serves as the input for the simulation engine.

1.2 USAGE OF THIS DOCUMENT

This document defines the modeling language for Brahms. This document is to be used by model builders to create Brahms models. Brahms model builders will have to comply to the language as defined in this document. This document is also used as a requirements specification for the parser that needs to be build to parse these models.

The language elements are defined in BNF (Backus-Naur Form) grammar rules. The notation used in these grammar rules is given in table 1.

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

Table 1: Synopsis of the notation used in BNF grammar rules

1.3 INTENDED AUDIENCE

This document is intended for anyone who wants to build models in Brahms. They will have to comply to the syntax as defined in this document. This document is also intended for the design team designing the parser for the Brahms models.

1.4 SUMMARY

Chapter 2 describes the modeling constructs, the syntax for the modeling constructs and the semantics for the modeling constructs. The semantics describe additional constraints for the syntax that cannot be defined using the BNF grammar rules (static semantics) and describe the meaning of some of the constructs where additional explanation is required (dynamic semantics).

Chapter 3 gives an overview of all the keywords defined for Brahms models.

Chapter 4 defines the concepts that will serve as the basis for every new concept (BaseGroup, BaseClass, BaseAreaDef, BaseConceptualClass).

2. LANGUAGE DEFINITION

2.1 IDENTIFIERS (ID)

name	**::=**	[letter][letter | digit | '-']*
qualified-name	**::=**	**name ['.' name]***
letter	**::=**	'a' | 'b' |…| 'z' | 'A' | 'B' |…| 'Z' | '_'
digit	**::=**	'0' | '1' |…| '9'
blank-character	**::=**	' ' | '\t' | '\n' | '\f' | '\r'
number	**::=**	[integer | long | double | float]
integer	**::=**	{ **+** | **-** } unsigned
long	**::=**	{ **+** | **-** } unsigned { **l** | **L** }
unsigned	**::=**	[digit]+
double	**::=**	[integer**.**unsigned]
float	**::=**	[integer**.**unsigned]{ **f** | **F** }
truth-value	**::=**	**true** | **false** | **unknown**
literal-string	**::=**	**"** [letter | digit | '-' | ':' | ';' | '.']* **"**
literal-character	**::=**	**'** [letter | digit | '-' | ':' | ';' | '.'] **'**
literal-symbol	**::=**	name
literal	**::=**	number | truth-value | literal-string | literal-character | literal-symbol

It is possible to add comments to models. One line comments need to start with '//'. Multi-line comments have to start with '/*' and end with '*/'.

2.2 COMPILATION UNIT (CUN)

2.2.1 Description

A compilation unit is the goal symbol for the syntactic grammar of Brahms models. A compilation unit consists of three parts, each of which is optional.

- A package declaration, giving the fully qualified name of the package to which the compilation unit belongs
- Brahms import declarations that allow Brahms types from other packages to be referred to using their simple names

Java import declarations that allow Java types to be referred to using their simple names

- Type declarations of group, agent, class, object, conceptual object class, conceptual object, area definition, area and path types.

2.2.2 Syntax

compilation-unit **::=** [PCK.package-declaration]
[IMP.import-declaration]*
[GRP.group |
AGT.agent |
CLS.object-class |
OBJ.object |
COC.conceptual-object-class |
COB.conceptual-object |
ADF.area-def |
ARE.area |
PAT.path]*

2.2.3 Semantics

A compilation unit is a file in a file system with the extension '.b'. The compiler loads a '.b' file when it is references in an import declaration.

2.3 PACKAGE DECLARATION (PCK)

2.3.1 Description

A package declaration appears within a compilation unit to indicate the package to which the compilation unit belongs. A package can also be referred to as a library. A compilation unit that has no package declaration is part of an unnamed package.

2.3.2 Syntax

package-declaration ::= **package** package-name **;**

package-name ::= ID.qualified-name

2.3.3 Semantics

The package name mentioned in a package declaration must be the fully qualified name of the package.

If a type named *T* is declared in a compilation unit of a package whose fully qualified name is *P*, then the fully qualified name of the type is *P*.*T*; thus in the example:

package nasa.phonemodel;

group PhoneUsers { }

the fully qualified name of group PhoneUsers is nasa.phonemodel.PhoneUsers.

The package declaration is used to find Brahms concepts in the file system. A package is to be mapped to a directory in the file system. The package declaration represents a hierarchical directory structure. The package nasa.phonemodel maps to a directory nasa\phonemodel in the file system. If the group PhoneUsers were defined in a file named PhoneUsers.b then this file would be located in the nasa\phonemodel directory. The compiler and Brahms virtual machine use the library path to find concepts in a specific package relative to the library path.

Compilation units that do not have a package statement are part of an unnamed package. The compiler and Brahms virtual machine use the library path to find concepts in an unnamed package by trying to locate them in the directory specified by the library path. It is the responsibility of the model builder to prevent naming conflicts in concepts that are part of an unnamed package. It is highly recommended to use packages for all Brahms concepts.

2.4 IMPORT DECLARATION (IMP)

2.4.1 Description

The Brahms language support two types of import declaration, a Brahms import declaration and Java import declaration.

The Brahms import declaration allows a Brahms type declared in another package to be referred to by a simple name that consists of a single identifier. When a Brahms model is compiled in compatibility mode the Brahms import declaration makes concepts defined in other compilation units available as one model, i.e. the import declaration is global, a concept imported in one compilation unit can be referenced from another compilation unit without requiring it to be imported into that compilation unit. However when the compiler is set to be strict (the preferred setting going forward) then each compilation unit must declare imports for every Brahms type used within the compilation unit, i.e. the imports are local to the compilation unit and an imported Brahms type would not be available to another compilation unit unless it is imported in that compilation unit. The change to the strict setting allows for multiple concepts with the same simple name but different qualified names to be used within a model.

The Java import declaration allows for a Java type to be referred to by its simple name. The Brahms compiler uses the Java import declarations to resolve and locate the Java Class file for the Java type to ensure validity of the Java type.

2.4.2 Syntax

Import-declaration	**::=**	[brahms-import-declaration | java-import-declaration]
brahms-import-declaration	**::=**	[**import** brahms-single-type-import **;** | **import** brahms-multi-type-import **;**]
brahms-single-type-import	**::=**	concept-name | package-name **.** concept-name
concept-name	**::=**	ID.name
brahms-multi-type-import	**::=**	***** | package-name **. ***
java-import-declaration	**::=**	[**jimport** java-single-type-import **;** | **jimport** java-type-import-on-demand **;**]
java-single-type-import	**::=**	ID.name | package-name **.** ID.name

java-type-import-on-demand **::=** package-name **. ***

2.4.3 Semantics

The Brahms import declaration allows for the import of specific concepts or for the import of a library of concepts defined in a package. The import of a specific concept is realized by referencing its name. The name of the concept must be the same as the name of the file in which it is stored. The extension of the file must always be '.b'.

To reference a specific concept in a library the package name can be used. The package name reflects the directory in which the concept is stored with a 'library-path' as its base path. So for example if the library-path is

library-path = C:\brahms

and I have an import statement like

import nasa.phonemodel.PhoneUsers;

then the concept PhoneUsers is expected to be defined as

```
package nasa.phonemodel;
group PhoneUsers { }
```

and is expected to be found in the file

C:\brahms\nasa\phonemodel\PhoneUsers.b

It is also possible to reference all concepts in a specific library. The wildcard '*' can be used in place of a specific concept-name. The following import statement will import all concepts in the phonemodel library:

import nasa.phonemodel.*;

This statement will import all concepts defined in the directory C:\brahms\nasa\ phonemodel defined in the files with the extension '.b' assuming the library-path is set to 'C:\brahms'.

The import statement:

import *;

will import all concepts defined in the files with extension '.b' that are in the same directory as the file in which the import statement is defined.

By default every model imports the 'brahms.base.*' library (referred to as the 'BaseModel') containing base constructs for groups and classes and containing standard available classes and relations. The import of this library does not have to be defined explicitly.

A java single type import declaration imports a single name type, by mentioning its canonical name.

A java type-import-on-demand declaration imports all the accessible types of a named type or package as needed. The import of java.lang.* is not required, the compiler always includes that package in its search path.

A java import declaration makes Java types available by their simple name only within the compilation unit that actually contains the import declaration.

2.5 MODEL (MOD)

2.5.1 Description

A Brahms model may be thought of, or expressed, as statements in a formal language developed for describing work practice. The language is domain-general, in the sense that it refers to no specific kind of social situation, workplace, or work practice; however it does embody assumptions about how to describe social situations, workplaces and work practice.

2.5.2 Syntax

model **::=** [PCK.package-declaration]
[IMP.import-declaration]*

2.5.3 Semantics

A model is defined by defining a compilation unit that can contain a package declaration and one or more import statements. The import statements are used to import those Brahms agents and objects that make up the model. This special compilation unit is compiled by the Brahms compiler into a model file with references to those imported agents and objects. When the Brahms virtual machine is told to load this model file it will load, initialize and start the agents and objects defined in this model file.

2.6 GROUP (GRP)

2.6.1 Description

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 beliefs. A group defines the work activities (activity frames and thought frames), the initial-beliefs of members in the group and the initial-facts in the world. 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."

Functional Roles

In terms of organizations the concept is similar to that of functional roles in an organization. A group in Brahms could represent a typical role in an organization; The work activities that someone performs when he or she plays that role. For example, we could represent the role of Maintenance Technician or Central Office Engineer as a "group".

Structural Groupings

A group in Brahms can also depict an organizational group. For example, we can define a group as "Members of the Work System Design group at S&T." We could now describe the work-activities, and initial-beliefs of members of the WSD group at S&T.

Conceptual Groupings

We can also create informal groups related to conceptual definitions that make sense in the modeling activity. For instance, in modeling the people at S&T we could create a group "People at 400 Westchester Avenue." We can now describe the activities and initial-beliefs that people at 400 Westchester Avenue have in common. This might not be that interesting, but in modeling people's interactions with legacy systems in NYNEX we could define a conceptual group of "LMOS users." In this group we could now describe how people interact with LMOS, and what initial-beliefs LMOS users have (for instance, the initial-belief that LMOS has data about today's trouble tickets).

2.6.2 Syntax

group **::=** **group** simple-group-name { group-membership }
{
{ **display :** ID.literal-string **;** }
{ **cost :** ID.number **;** }

{ **time_unit :** ID.number **;** }
{ icon **:** ID.literal-string **;** }
{ attributes }
{ relations }
{ initial-beliefs }
{ initial-facts }
{ activities }
{ workframes }
{ thoughtframes }
}

simple-group-name **::=** ID.name

group-name **::=** ID.qualified-name

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

attributes **::=** **attributes :** [ATT.attribute]*

relations **::=** **relations :** [REL.relation]*

initial-beliefs **::=** **initial_beliefs :** [BEL.initial-belief]*

initial-facts **::=** **initial_facts :** [FCT.initial-fact]*

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

activity **::=** [CAC.composite-activity |
PAC.primitive-activity |
MOV.move-activity |
CAA.create-agent-activity |
COA.create-object-activity |
CRA.create-area-activity |
COM.communicate-activity |
BCT.broadcast-activity |
JAC.java-activity |
GET.get-activity |
PUT.put-activity |
GAC.gesture-activity]

workframes **::=** **workframes :** [WFR.workframe]*

thoughtframes **::=** **thoughtframes :** [TFR.thoughtframe]*

2.6.3 Semantics

Group membership

In a model a hierarchy of groups can be built by defining the group-membership. A group can be a member of more than one group. When a group is a member of a group the member-group will 'inherit' the attributes, relations, initial-beliefs, initial-facts, activities, workframes and thoughtframes from its parent groups. Private attributes and relations are not inherited, only public and protected attributes and relations are inherited. In case the same constructs are encountered in the inheritance path always the most specific construct will be used, meaning that a workframe defined for a group lowest in the hierarchy tree has precedence over a workframe with the same name higher in the hierarchy.

Cost and Time-Unit

The cost and time-unit are used for statistical purposes and define the cost/time-unit (in seconds) for work done by members of the group. The members of the group can override the cost and time-unit figures.

Defaults

Every group in a model is by definition a member of 'BaseGroup' defined in the 'BaseModel' library which is imported by definition for every model. The 'BaseGroup' defines built-in attributes, relations, initial-beliefs, initial-facts, workframes and thoughtframes as defaults for groups. The 'BaseGroup' membership does not have to be defined explicitly. Other defaults are:

display	=	<group-name>
cost	=	0
time_unit	=	0

Constraints

1. The name of a group must be unique amongst all concepts defined in the same package.

2. The time_unit defines the time in seconds.

2.7 AGENT (AGT)

2.7.1 Description

An agent in Brahms is the most central construct in a Brahms model. An agent represents an interactive system, a subject with behavior interacting with the world. An agent can for example represent a person in an organization, but could also represent an animal in a forest. A Brahms model is always about the activities of agents in a work process. In Brahms it is also possible to implement the behavior of an agent in Java instead of in the Brahms language. This is done by defining an external agent. An external agent can interact with the Brahms agents and Brahms agents can interact with external agents. External agents do not have any reasoning capabilities unless these capabilities are built-in by the implementer of the external agent.

To develop external agents the JAPI is required. The JAPI is part of the Brahms Virtual Machine. The API documentation provides all the information needed to develop external agents. The documentation specifies what API functions are available (http://www.agentisolutions.com/documentations/vmapi/index.html).

2.7.2 Syntax

agent **::=** **agent** simple-agent-name { GRP.group-membership }
{
{ **display :** ID.literal-string **;** }
{ **cost :** ID.number **;** }
{ **time_unit :** ID.number **;** }
{ **icon :** ID.literal-string **;** }
{ **location :** ARE.area-name **;** }
{ GRP.attributes }
{ GRP.relations }
{ GRP.initial-beliefs }
{ GRP.initial-facts }
{ GRP.activities }
{ GRP.workframes }
{ GRP.thoughtframes }
}

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

simple-agent-name **::=** ID.name

agent-name ::= ID.qualified-name

2.7.3 Semantics

Group membership

An agent can be a member of one or more groups. When an agent is a member of a group the agent will 'inherit' attributes, relations, initial-beliefs, initial-facts, activities, workframes and thoughtframes from the group(s) it is a member of. All attributes and relations are inherited including private ones (an agent can be seen as an instance of a group in terms of object oriented practices). In case the same constructs are encountered in the inheritance path always the most specific construct will be used, meaning that a workframe defined for the agent has precedence over a workframe with the same name defined in one of the groups of which the agent is a member.

Defaults

Every agent in a model is by definition a member of 'BaseGroup' defined in the 'BaseModel' library which is imported by definition for every model. The 'BaseGroup' defines built-in attributes, relations, initial-beliefs, initial-facts, activities, workframes and thoughtframes as defaults for agents and groups. The 'BaseGroup' membership does not have to be defined explicitly. Other defaults are:

display	=	<simple-agent-name>
cost	=	0
time_unit	=	0
location	=	none

Constraints

1. The name of an agent must be unique amongst all concepts defined in the same package.

2. The time_unit defines the time in seconds.

3. The Java class name of an external agent must be identical to the agent's name as defined in the Brahms model. The Java class must also be in the same package as the package the external agent is defined in.

2.8 OBJECT CLASS (CLS)

2.8.1 Description

The concept of a 'class' in Brahms is similar to the concept of a template or class in object-oriented programming. It defines the activities (workframes), initial-facts and initial-beliefs for instances of that class (i.e. 'objects'). Classes are used to define inanimate artifacts, such as phones, faxes, computer systems, pieces of paper, etc.

2.8.2 Syntax

object-class **::=** **class** simple-class-name { class-inheritance }

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

simple-class-name **::=** ID.name

object-class-name **::=** **ID.qualified-name**

class-inheritance **::=** **extends** object-class-name [**,** object-class-name]*

2.8.3 Semantics

Class inheritance

In a model a hierarchy of classes can be built by defining the class inheritance. A class can inherit from more then one class, so multiple inheritance is supported. When a class is a subclass of a class the subclass will 'inherit' the attributes, relations, initial-beliefs, initial-facts, activities, workframes and thoughtframes from its parent classes. Private attributes and relations are not inherited, only public and protected attributes and relations are inherited. In case the same constructs are encountered in the inheritance

path always the most specific construct will be used, meaning that for example a workframe defined for a class lowest in the hierarchy tree has precedence over a workframe with the same name higher in the hierarchy.

Cost and Time-Unit

The cost and time-unit are used for statistical purposes and define the cost/time-unit (in seconds) for work done by instances of the class. The instances of the class can override the cost and time-unit figures.

Resource

The resource attribute defines whether or not instances of the class are considered to be a resource when used in an activity (resource attribute is set to true) or whether the instances of the class are considered something that is worked on (resource attribute is set to false). The resource attribute is used in relation with the touched-objects definition for activities (see the semantical description of touched-objects in the definition of the primitive-activity).

Defaults

Every class in a model is by definition a member of 'BaseClass' defined in the 'BaseModel' library which is imported by definition for every model. The 'BaseClass' defines built-in attributes, relations, initial-beliefs, initial-facts, workframes and thoughtframes as defaults for classes. The 'BaseClass' membership does not have to be defined explicitly. Other defaults are:

display	=	<simple-class-name>
cost	=	0
time_unit	=	0
resource	=	false

Constraints

1. The name of a class must be unique amongst all concepts defined in the same package.

2. The time_unit defines the time in seconds.

2.9 OBJECT (OBJ)

2.9.1 Description

An 'object' in Brahms is the second most central element in a Brahms model. An object represents a specific artifact in the world. It is possible to model the activities of an artifact in an organization. For example the data processing activities of a computer system can be modeled. The activities can be defined in the object's class (which will be inherited by the object) and/or can be defined for the object itself.

2.9.2 Syntax

object ::= **object** simple-object-name
instanceof object-class-name
{ COB.conceptual-object-membership }
{
{ **display :** ID.literal-string **;** }
{ **cost :** ID.number **;** }
{ **time_unit :** ID.number **;** }
{ **icon :** ID.literal-string **;** }
{ **resource :** ID.truth-value **;** }
{ **location :** ARE.area-name **;** }
{ GRP.attributes }
{ GRP.relations }
{ GRP.initial-beliefs }
{ GRP.initial-facts }
{ GRP.activities }
{ GRP.workframes }
{ GRP.thoughtframes }
}

simple-object-name ::= ID.name

object-name ::= **ID.qualified-name**

2.9.3 Semantics

Conceptual object membership

An object can be part of one or more conceptual objects by defining the conceptual-object-membership for the object. This allows for later grouping of statistical results for the object with other objects in one conceptual object.

Resource

The resource attribute defines whether or not the object is considered to be a resource when used in an activity (resource attribute is set to true) or whether the object is considered something that is worked on (resource attribute is set to false). The resource attribute is used in relation with the resources definition for activities (see the semantical description of resources in the definition of the primitive-activity).

Defaults

display	=	<simple-object-name>
cost	=	0
time_unit	=	0
resource	=	<the resource attribute value of object-class-name>
location	=	none

Constraints

1. The name of an object must be unique amongst all concepts defined in the same package.

2. The time-unit defines the time in seconds.

2.10 CONCEPTUAL OBJECT CLASS (COC)

2.10.1 Description

A conceptual object class defines a type of conceptual objects used in a model. For the definition of conceptual objects see the section on conceptual objects.

2.10.2 Syntax

conceptual-object-class **::=** **conceptual_class** simple-class-name
{ conceptual-class-inheritance }
{
{ **display :** ID.literal-string **;** }
{ **icon :** ID.literal-string **;** }
{ GRP.attributes }
{ GRP.relations }
}

simple-class-name **::=** ID.name

conceptual-class-name **::=** ID.qualified-name

conceptual-class-inheritance **::=** **extends** conceptual-class-name
[, conceptual-class-name]*

2.10.3 Semantics

Conceptual class inheritance

In a model a hierarchy of conceptual classes can be built by defining the conceptual class inheritance. A conceptual class can inherit from more then one conceptual class, so multiple inheritance is supported. When a conceptual class is a subclass of a conceptual class the subclass will 'inherit' the attributes and relations from its parent conceptual classes. Private attributes and relations are not inherited, only public and protected attributes and relations are inherited. In case the same constructs are encountered in the inheritance path always the most specific construct will be used, meaning that for example an attribute defined for a class lowest in the hierarchy tree has precedence over an attribute with the same name higher in the hierarchy.

Defaults

display = <simple-class-name>

Constraints

1. The name of a conceptual-object-class must be unique amongst the concepts defined in the same package.

2.11 CONCEPTUAL OBJECT (COB)

2.11.1 Description

A conceptual object is used to allow for a user to track things that exist as concepts in people's minds, like the concept of an order. The concepts do not exist as such but do have incarnations in the form of real artifacts, such as a fax, a form, or a database record. Through conceptual objects statistics can be generated such as touch time and cycle time and object flows can be generated through a work process.

2.11.2 Syntax

conceptual-object **::=** **conceptual_object** simple-object-name

instanceof conceptual-class-name
{ conceptual-object-membership }
{
{ **display :** ID.literal-string **;** }
{ **icon :** ID.literal-string **;** }
{ GRP.attributes }
{ GRP.relations }
}

simple-object-name **::=** ID.name

conceptual-object-name **::=** ID.qualified-name

conceptual-object-membership **::=**
partof conceptual-object-name
[**,** conceptual-object-name]*

2.11.3 Semantics

Conceptual-object-membership

A conceptual-object can in itself be a member of other conceptual object forming a hierarchy of concepts for grouping statistical results.

Defaults

display = <simple-object-name>

Constraints

1. The name of a conceptual-object must be unique amongst all concepts defined in the same package.

2.12 AREA DEFINITION (ADF)

2.12.1 Description

An area definition is used for defining area constructs used for representing geographical information in a model. Area definitions are similar to classes in their use. Examples of area definitions are ‘World’, ‘Building’, and ‘Floor’.

2.12.2 Syntax

area-definition **::=** **areadef** simple-area-def-name { area-def-inheritance }
{
{ **display :** ID.literal-string **;** }
{ **icon :** ID.literal-string **;** }
{ GRP.attributes }
{ GRP.relations }
{ GRP.initial-facts }
}

simple-area-def-name **::=** ID.name

area-def-name **::=** ID.qualified-name

area-def-inheritance **::=** **extends** area-def-name [**,** area-def-name]*

2.12.3 Semantics

Area definition inheritance

In a model a hierarchy of area definitions can be built by defining the area definitions inheritance. An area definition can inherit from more then one area definition, so multiple inheritance is supported. When an area definition is a subclass of another area definition the subclass will 'inherit' the attributes, relations, and initial-facts from its parent area definitions. Private attributes and relations are not inherited, only public and protected attributes and relations are inherited. In case the same constructs are encountered in the inheritance path always the most specific construct will be used, meaning that for example an attribute defined for an area definition lowest in the hierarchy tree has precedence over an attribute with the same name higher in the hierarchy.

Defaults

display = <simple-area-def-name>

Constraints

1. The name of an area definition must be unique amongst the concepts defined in the same package.

2.13 AREA (ARE)

2.13.1 Description

An area represents a geographical location and is used to create a geographical representation for use in the model. Examples are 'NewYorkCity', 'SandTBuilding', etc. Area's are instances of area definitions.

2.13.2 Syntax

area **::=** **area** simple-area-name
instanceof area-def-name
{ **partof** area-name }
{
{ **display :** ID.literal-string **;** }
{ icon **:** ID.literal-string **;** }
{ GRP.attributes }
{ GRP.relations }
{ GRP.initial-facts }
}

simple-area-name **::=** ID.name

area-name **::=** ID.qualified-name

2.13.3 Semantics

Area Decomposition

Areas can be decomposed into sub-areas. For example a building can consist of one or more floors. The decomposition can be modeled using the part-of relationship. Model builders have indicated that they frequently want to reason about the area decomposition in their models. The virtual machine therefore generates a set of initial facts about the decomposition of the areas in a model. The virtual machine generates initial facts using the built-in relations 'isSubAreaOf' and 'hasSubArea' defined in the BaseAreaDef area definition. For each area an initial fact is generated about its aggregate area using the 'isSubAreaOf' relation, only one initial fact can be generated since an area can only be part of one other area. For each area we also generate an initial fact for every sub area of the area using the 'hasSubArea' relation. Agents can detect any of these facts as needed by these agents. Note that the virtual machine only generates the initial facts for the direct relation between areas and sub-areas and not the indirect relations. If area A1 has a sub-area B1 and B1 has a sub-area C1 then we only generate the initial facts relating A1 to B1 and B1 to C1 but we do not generate the initial facts relating A1 to C1. This relationship would have to be deduced by the model builder if it is required for a model.

Defaults

display = <simple-area-name>

Constraints

1. The name of an area must be unique amongst the concepts defined in the same package.

2.14 PATH (PAT)

2.14.1 Description

A path connects two areas together and represents a route that can be taken by an agent or object to travel from one area to another. For the path is specified how long it takes to travel from one area to the other.

2.14.2 Syntax

path-def **::=** **path** simple-path-name
{
{ **display :** ID.literal-string **;** }
area1 : ARE.area-name **;**
area2 : ARE.area-name **;**
{ **distance :** ID.unsigned **;** }
}

simple-path-name **::=** ID.name

path-name **::=** ID.qualified-name

2.14.3 Semantics

Distance

The distance represents the time it takes to move from area1 to area2 and vice versa. In future versions of the language the distance will represent the actual distance and based on the transportation used to travel over the path the duration will be calculated.

Defaults

display = <simple-path-name>
distance = 0

Constraints

1. The name of a path must be unique amongst the concepts defined in the same package.

2. The distance represents the travel duration in seconds.

2.15 ATTRIBUTE (ATT)

2.15.1 Description

Attributes represent a property of a group, agent, object class or object. Attributes may also represent properties of Java objects. Attributes have values. Attributes of a class or value type are single-valued attributes, attributes of a collection type are multi-valued. The value of an attribute is defined through facts and/or beliefs. For more information about collection types see section 2.41.

2.15.2 Syntax

attribute **::=** { **private** | **protected** | **public** }
attribute-type-def
attribute-name
{ attrib-body }
;

attribute-name **::=** **location** | ID.name

attribute-type-def **::=** [type-def
| collection-type-def
| relation-type-def]

type-def **::=** [class-type-def
| value-type-def
| java-type-def]

class-type-def **::=** [**Agent** |
Group |
Class |
Object |
ActiveClass |
ActiveInstance |
ActiveConcept |
ConceptualClass |
ConceptualObject |
ConceptualConcept |
AreaDef |
Area |
GeographyConcept |
Concept |
GRP.group-name |
CLS.object-class-name |
COC.conceptual-class-name
ADF.area-def-name]

value-type-def **::=** [**int** | **long** | **double** | **symbol** | **string** | **boolean** |
char | byte | short | float]

collection-type-def **::=** [**map**]

relation-type-def **::=** **relation (** class-type-def **)**

java-type-def **::=** **java (** java-ref-type-def **)**

java-ref-type-def **::=** java-class-or-interface-type-def [**[ ]**]*

java-class-or-interface-type-def **::=**
java-type-decl-specifier { java-type-arguments }

java-type-decl-specifier **::=**
[ID.name [**.** ID.name]*
| java-class-or-interface-type-def **.** ID.name]

java-type-arguments ::= **<** java-type-argument [**,** java-type-argument]* **>**

java-type-argument ::= [java-ref-type-def
| **?** { java-wildcard-bounds }]

java-wildcard-bounds ::=
[**extends** java-ref-type-def
| **super** java-ref-type-def]

attrib-body ::= **{**
{ **display :** ID.literal-string **;** }
}

2.15.3 Semantics

Attribute scope

Attributes are always defined within a group, agent, conceptual-class, conceptual-object, class or object definition and cannot be defined outside any of these concepts or inside of any other concepts. Attributes can have different scopes within the specified concepts defined by one of the keywords private, protected or public.

Private attributes:

Private attributes are scoped down to only the concept for which it is defined. The private attribute is not inherited by sub groups or sub classes (agents /objects that are members/instances of the group/class will inherit the attribute) and the private attribute can only be referenced by initial beliefs, initial facts, workframes and thoughtframes for that specific concept.

Protected attributes:

Protected attributes are inherited by sub groups and sub classes. Protected attributes can only be referenced by initial beliefs, initial facts, workframes and thoughtframes of the concept for which the attribute is specified or any of the sub groups/sub classes and of agents/objects that are members/instances of the sub group(s)/class(es).

Public attributes:

Public attributes are similar to protected attributes, the only difference is that they can be referenced by initial beliefs, initial facts, workframes and thoughtframes in any group, agent, class or object.

Value assignment

Value assignment of attributes differs from value assignments in third and fourth generation computer languages (which usually use an assignment operator like '=' or ':='. Assignment of a value for an attribute is done through beliefs and facts.

Meta types

The meta types allow for binding of concepts that are considered to be a subtype of the meta types. The following concepts can be bound to the specified meta types:

Group	any group
Agent	any agent regardless of the group it is a member of
Class	any class
Object	any object regardless of the class it is an instance of
ActiveClass	any group and class
ActiveInstance	any agent and object
ActiveConcept	any active class and active instance
ConceptualClass	any conceptual class
ConceptualObject	any conceptual object
ConceptualConcept	any conceptual class and conceptual object
AreaDef	any area definition
Area	any area
GeographyConcept	any area definition and area
Concept	any active concept, conceptual concept and geograpy concept

Collection types

The collection types allow for attributes to have multiple values assigned to them. The following collection types are currently supported:

map	collection for which values are accessible via a unique index or key being either an integer or string.

Relation types

The relation type is a new way to declare relations in the language in the attributes section as an alternative to declaring the relations in the relations section. This is in preparation for eliminating all section headers and would still allow for the notion of relations since they require a different notation and different handling in conditions compared to regular attributes. For more on relations see the section on relations 2.16.

Java types

Attributes, variables and parameters can now also be declared to be of any Java reference type allowing for the modeler to directly reference Java objects. Note that the Brahms language assumes at a minimum Java 5.0, the Brahms language supports the generics notations. For more information on Java reference types see the Java Language Specification [1].

Attributes of Java Objects

Attributes may be used to represent the values of properties of Java objects in Brahms facts and beliefs. An attribute of a Java object is not explicitly declared in the Brahms language; rather, it is resolved to a Java property whenever there is an occurrence of *object*.*attribute* (see BEL.obj-attr) where *object* is a variable or parameter that has been declared to be of a Java type. Based on the context of use, the Java property may be required to be readable or writable.

In determining the property of a Java object that is referenced by an attribute name, Brahms first looks for public methods defined on the Java object's class following the Java Beans naming convention. For an attribute "foo", it will look for zero argument methods with a declared return type named "getFoo" or "isFoo" and single argument void methods named "setFoo". If Brahms fails to find a method using these names, it will look for a public method with the same name as the attribute, in this case "foo", that is either a zero argument method with a declared return type or a single-argument method with a void return type. Finally, if no such method can be found, Brahms will look for a field "foo" declared in the class with any scope (public, private, package, or protected).

Currently, Brahms represents properties of Java objects as single-valued attributes. A Java object cannot have a collection type attribute or participate in Brahms relations.

Defaults

The default scope of an attribute is 'public'.

display = <attribute-name>

Constraints

1. The name of an attribute must be unique within the definition of a group, agent, class or object. In case of a name conflict in multiple inheritance (two different concepts from which is inherited define an attribute with the same name) the following conflict resolution strategy is chosen. If both attributes are of the same type just one definition will remain with the same name and same type. If the types of the attributes differ an error will be generated.

2. If the name of the attribute is location, its type must be the name of an area definition or the meta-type Area.

3. If a Java type is used as the type of the attribute the Java type must reference a Java type that is either referenced by its simple name or by its fully qualified name(s). If the simple name is used the simple name must be resolvable to a fully qualified name using the Java import statements (jimport). The compiler must be able to load the Java Class for the type, the compiler uses this method to ensure that the Java type is valid. The Java Classes for the types used in the Brahms language must therefore be in the Java classpath.

2.16 RELATION (REL)

2.16.1 Description

Relations represent a relation between two concepts. The first (left hand side) concept is always the concept for which the relation is defined, the second concept (right hand side) can be any concept.

NOTE: The syntax for the declaration of the relation will be replaced by the preferred syntax of declaring an attribute to be of a relation type. See the previous section for details.

2.16.2 Syntax

relation **::=** { **private** | **protected** | **public** }
ATT.class-type-def
relation-name
{ attrib-body }
;

relation-name::= ID.name

2.16.3 Semantics

Relation scope

Relations are always defined within a group, agent, conceptual- class, conceptual-object, class or object definition and cannot be defined outside any of these concepts or inside of any other concepts. Relations can have different scopes within the specified concepts defined by one of the keywords private, protected or public.

Private relations:

Private relations are scoped down to only the concept for which it is defined. The private relation is not inherited by sub groups or sub classes (agents /objects that are members/instances of the group/class will inherit the relation) and the private relation can only be referenced by initial beliefs, initial facts, workframes and thoughtframes for that specific concept.

Protected relations:

Protected relations are inherited by sub groups and sub classes. Protected relations can only be referenced by initial beliefs, initial facts, workframes and thoughtframes of the concept for which the relation is specified or any of the sub groups / sub classes and of agents/objects that are members/instances of the sub group(s)/class(es).

Public relations:

Public relations are similar to protected relations, the only difference is that they can be referenced by initial beliefs, initial facts, workframes and thoughtframes in any group, agent, class or object.

Defaults

The default scope of a relation is 'public'.

display = <relation-name>

Constraints

1. The name of a relation must be unique within the definition of a group, agent, class or object. In case of a name conflict in multiple inheritance (two different concepts from which is inherited define a relation with the same name) the following conflict resolution strategy is chosen. If both relations are of the same type just one definition will remain with the same name and same type. If the types of the relations differ an error will be generated.

2.17 VARIABLE (VAR)

2.17.1 Description

Variables can be used in a workframe or thoughtframe to write more generic work- and thoughtframes. Before a variable can be used it has to be declared. The scope of the variable is bound to the frame it is declared in. A variable that is not declared within the workframe it is used in, must be declared higher up in the activity-tree the workframe is part of. (The activity tree is created through composite activities.)

Variables can be declared either in the variables section of a workframe or thoughtframe definition or in the body of a workframe definition. Frame variables declared in the variables section have scope throughout the frame definition, including preconditions and detectable conditions. Local variables are declared in the body of a workframe and have a scope that is limited to subsequent body elements.

2.17.2 Syntax

variable **::=** [**collectall** | **foreach** | **forone**]
([ATT.type-def] **)**
variable-name
{ variable-body }
;

variable-name **::=** ID.name

variable-body **::=** **{**
{ **display :** ID.literal-string **;** }
}

local-variable **::=** ATT.type-def variable-name
{ '=' JAV.initializer-expression }
;

2.17.3 Semantics

Quantification

Frame variables are of one of three quantification types: collect-all, for-each and for-one. The difference between the three quantification types is the way variables are bound to a specific context of a defined type (agent, object, or other value). The difference in binding is as follows:

for-each variable:

A for-each variable is bound to only one context. For each context that can be bound to the variable a separate instance is created for the workframe in which the variable is bound.

For example in the following frame:

```
workframe DoSomething {
      variables:
            foreach(Order) order;

      when (
            knownval(order is-assigned-to current) )
      do {
            workOnOrder();
      }
}
```

There are three Order instances in the model (order1, order2, and order3) satisfying the precondition. For this workframe three instances will be created in which the for-each variable is bound to one of the orders in each frame instantiation. This means that the agent for which the workframe is defined can only work on one order at a time and will work on them in consecutive order if no interruptions take place.

collect-all variable:

A collect-all variable can be bound to more than one context. The variable will be bound to all contexts satisfying the condition in which it is defined. Only one frame instantiation will be created as a result of the binding with the collect-all variable. If we consider the same example as for for-each variables changing the quantification of the variable to collect-all.

```
workframe DoSomething {
      variables:
            collectall(Order) order;

      when (
            knownval(order is-assigned-to current) )
      do {
            workOnOrder();
      }
}
```

Also assume that again three orders match with the precondition based on the beliefs of the agent, then all three orders are bound to the variable and one frame instantiation will be created for the agent to work on. This means that the agent for which this workframe is defined will work on all orders at the same time.

for-one variable:

A for-one variable can be bound to only one context. Only one frame-instantiation will be created as a result of the binding with the for-one variable. A for-one variable binds to the first context satisfying the condition in which it is defined. If we consider the same example as for for-each variables changing the quantification of the variable to for-one.

```
workframe DoSomething {
      variables:
            forone(Order) order;

      when (
            knownval(order is-assigned-to current) )
      do {
            workOnOrder();
      }
}
```

Also assume that again three orders match with the precondition based on the beliefs of the agent, then one of the orders will be bound to the variable and one frame instantiation will be created for the agent to work on. This means that the agent only works on one order and it doesn't matter on which order. The other two orders will not be worked on.

local variable:

Once a frame variable has been bound in a particular frame instantiation it may not be bound to a new value or values by a frame body element. Local variables, on the other hand, may be freely rebound to new values by body elements that follow their declarations.

A local variable is not declared to have one of the three quantification types. A local variable is most similar to a for-one frame variable in that it may only be bound to a single value at a time for any frame instantiation.

A local variable declaration may have an optional initializer expression that is evaluated to produce an initial binding for the variable, The initializer expression may be a Brahms expression, a Java expression or (for a variable of type Java array) an array initializer. The syntax and semantics of an initializer expression are presented in the Java Expression section (JAV). If the initializer expression evaluates to multiple values due to the presence of collect-all variables or parameters bound to collect-all variables, only the first value will be bound to the local variable by default. However, as explained in the Java Expression section, if the local variable has a suitable Java List or Collection type then multiple values will be accumulated into a list and bound to the local variable.

Defaults

A variable is by default an assigned variable unless otherwise specified.

display = <variable-name>

Constraints

1. The name of the variable must be unique within the definition of a workframe or thoughtframe.

2.18 INITIAL-BELIEF (BEL)

2.18.1 Description

A belief is a first-order predicate statement about the world. Beliefs are always *local* to an agent or object, i.e. only the agent/object can access its beliefs, no other agent/object can. This allows us to represent how a specific agent 'views' the state of the world. For objects beliefs represent information stored in/on the object. Agents act based on their beliefs. Beliefs are the 'triggers' of agent's actions.

Initial beliefs define the initial state for an agent and define the initial information for objects. Initial beliefs are turned into actual beliefs for the agent when the model is initialized for a simulation run. Beliefs can also be created by consequences in work- and thoughtframes, by detectables as well as through communications.

2.18.2 Syntax

initial-belief **::=** **(** [value-expression | relational-expression] **) ;**

value-expression **::=** obj-attr equality-operator value |

obj-attr equality-operator sgl-object-ref

equality-operator **::=** **=** | **!=**

evaluation-operator **::=** BEL.equality-operator | **>** | **>=** | **<** | **<=**

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

tuple-object-ref **::=** AGT.agent-name |

OBJ.object-name |

ARE.area-name |

VAR.variable-name |

PAC.param-name |

current

collection-index **::=** ID.literal-string |

ID.unsigned |

VAR.variable-name |

PAC.parameter-name

sgl-object-ref **::=** AGT.agent-name |

OBJ.object-name |

ARE.area-name |

VAR.variable-name |

PAC.param-name |

unknown |

current

value **::=** ID.literal-string | ID.number | PAC.param-name | **unknown**

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

2.18.3 Semantics

Constraints

1. Variables and parameters are not allowed in the definition of an initial belief.
2. The attribute type and the right hand side value-type of a value-expression must be the same, except in the case the attribute type is a collection type.
3. The left hand side and right hand side types in a relational expression must match the types as defined for the relation used in the relational expression.

2.19 INITIAL-FACT (FCT)

2.19.1 Description

Facts represent the state of the world. A fact is a first-order predicate statement about the world. Facts are in contrast to beliefs, *global*. Any agent can detect a fact in the world and turn it into a belief and act on it. Objects on the other hand, react to facts (in workframes).

Initial facts define the initial state of the world. Initial facts are turned into actual facts in the world when the model is initialized for a simulation run. Facts can also be created by consequences in workframes (not in thoughtframes).

2.19.2 Syntax

initial-fact **::=** **(** [BEL.value-expression | BEL.relational-expression] **) ;**

2.19.3 Semantics

Constraints

1. Variables and parameters are not allowed in the definition of an initial fact.

2. The attribute type and the right hand side value-type of a value-expression must be the same, except in the case the attribute type is a collection type.

3. The left hand side and right hand side types in a relational expression must match the types as defined for the relation used in the relational expression.

2.20 WORKFRAME (WFR)

2.20.1 Description

A workframe is an action rule for an agent or object. It is a declarative description of under what condition (in case of an agent, beliefs that an agent has or in case of an object, the facts or beliefs in the world depending on the workframe type) the agent/object will perform the activities specified in the body of the rule. Workframes are treated like data-driven (forward chaining) production rules. However, workframes are different from production rules, in that they specify activities that agents and objects can perform (are engaged in) - production rules specify what conclusions can be drawn based on the conditions that are met.

In Brahms we separate facts in the world from beliefs that agents have. For example, in Brahms we can have a fact '*the color of John's Car is red*'. Agent John might have the belief '*the color of John's Car is red*', but agent Caroline might have the belief '*the color of John's Car is green*'. Agent workframes get 'worked on' (in production rules systems we call this 'get fired') based on the beliefs that agents have. This means that, in the example above, if John and Caroline have the same workframe using the belief of John's Car is red as a condition for the activation of the workframe; John will start working on the workframe, whereas Caroline will not start working on the workframe. Using this separation of beliefs and facts in the world allows Brahms to model agent's activities, based on changes in the world (facts) detected through detectables, and the agent-specific beliefs that are created. For objects beliefs are the information that an object carries. By default workframes for objects are only triggered by facts in the world, the type of the workframe is by default 'factframe'. If the type of the workframe is set to 'dataframe' then the workframe will be triggered by the beliefs of the object. The workframe is in that case an information processing frame. It is not possible to specify the type of the workframes for agents. Agent's workframes are only activated by beliefs matching the preconditions.

2.20.2 Syntax

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

workframe-name ::= ID.name

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

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

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

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

workframe-body-element ::= [PAC.activity-ref |
CON.consequence |
DEL.delete-action |
VAR.local-variable |
JAV.assignment **|**
JAV.method-invocation-statement]

2.20.3 Semantics

Type

The type attribute can only be set for workframes defined for classes and objects. The value for the type attribute can be one of ‘factframe’ or ‘dataframe’. The default value is ‘factframe’ If the value is ‘factframe’ then the workframe’s preconditions are matched against the facts in the world. If the value is ‘dataframe’ then the workframe’s preconditions are matched against the beliefs of the object for which the workframe is specified. A dataframe is a workframe processing the data/information maintained by an object allowing the processed data/information to result in actions that change the state in the world.

Repeat

A workframe can be performed one or more times depending on the value of the 'repeat' attribute. A workframe can only be performed once if the repeat attribute is set to false. A workframe can be performed repeatedly if the repeat attribute is set to true. In case the repeat attribute is set to false, the workframe can only be performed once for the specific binding of the variables at run-time. The scope of the repeat attribute of a workframe defined as part of a composite activity is limited to the time the activity is active, meaning that the workframe with a specific binding and a repeat set to false will not execute repeatedly while the composite activity is active. As soon as the composite activity is ended the states are reset and in the next execution of the activity it is possible for the workframe with the same binding to be executed. So only for top-level workframes the state will be stored permanently during a simulation run.

Priority

The workframe priority can be set in one of two ways. The priority can be set by setting the value for the priority attribute or the priority can be deduced based on the priorities of the activities defined within the workframe, the workframe will get the priority of the activity with the highest priority. If no priority is specified the priority will be deduced from the activities, otherwise the specified priority is used.

Defaults

display	=	<workframe-name>
type	=	factframe
repeat	=	false
priority	=	0

Constraints

1. The workframe name has to be unique within the definition of a group, agent, object-class or object.

2.21 THOUGHTFRAME (TFR)

2.21.1 Description

A thoughtframe is the Brahms equivalent of a production rule for an agent or object. A thoughtframe allows an agent or object to deduce new beliefs from existing beliefs. The difference between a thoughtframe and a workframe is that a thoughtframe can only have consequences in its body. A thoughtframe consists of preconditions and consequences.

2.21.2 Syntax

thoughtframe **::=** **thoughtframe** thoughtframe-name
{
{ **display :** ID.literal-string **;** }
{ **repeat :** ID.truth-value **;** }
{ **priority :** ID.unsigned **;** }
{ WFR.variable-decl }
{ [WFR.precondition-decl thoughtframe-body-decl] }
}

thoughtframe-name **::=** ID.name

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

thoughtframe-body-element **::=** CON.consequence

2.21.3 Semantics

Repeat

A thoughtframe can be performed one or more times depending on the value of the 'repeat' attribute. A thoughtframe can only be performed once if the repeat attribute is set to false. A thoughtframe can be performed repeatedly if the repeat attribute is set to true. In case the repeat attribute is set to false, the thoughtframe can only be performed once for the specific binding of the variables at run-time. The scope of the repeat attribute of a thoughtframe defined as part of a composite activity is limited to the time the activity is active, meaning that the thoughtframe with a specific binding and a repeat set to false will not execute repeatedly while the composite activity is active. As soon as the composite activity is ended the states are reset and in the next execution of the activity it is possible for the thoughtframe with the same binding to be executed. So only for top-level thoughtframes the state will be stored permanently during a simulation run.

Priority

Setting the thoughtframe priority allows the model builder to control the execution sequence of thoughtframes if more then one thoughtframe is available at the same time. The priority can be set by setting it to a value greater than 0. Note that it is not recommended to use priorities to control the sequence of thoughtframe execution. A better modeling practice is to define better preconditions for the thoughtframes.

Defaults

display	=	<thoughtframe-name>
repeat	=	false
priority	=	0

Constraints

1. The thoughtframe name has to be unique within the definition of a group, agent, object-class or object.

2. It is not possible to use unassigned variables in thoughtframes, therefor the definition of these variables is not allowed.

3. The consequences in thoughtframes can only conclude beliefs.

2.22 PRIMITIVE ACTIVITY (PAC)

2.22.1 Description

A primitive activity is the lowest level of activity an agent or object works on for a specified amount of time. A primitive activity has no side-effects.

2.22.2 Syntax

primitive-activity **::=** **primitive_activity** activity-name **(**
{ param-decl [**,** param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | param-name] **;** }
{ **random :** [ID.truth-value | param-name] **;** }
{ **min_duration :** [ID.unsigned | param-name] **;** }
{ **max_duration :** [ID.unsigned | param-name] **;** }
{ resources }
}

activity-name **::=** ID.name

param-decl **::=** param-type param-name

param-type **::=** ATT.type-def

param-name **::=** ID.name

resources **::=** **resources :** [param-name | OBJ.object-name]
[, [param-name | OBJ.object-name]*;

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

param-expr **::=** GRP.group-name |
AGT.agent-name |
CLS.object-class-name |
OBJ.object-name |
COC.conceptual-class-name |
COB.conceptual-object-name |
ARE.area-name |
VAR.variable-name |
ID.literal |
unknown

2.22.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for primitive activities. These input parameters can be used to make activities more generic. In the reference the values for the input parameters have to be passed.

Priority

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.

Duration

Activities in general have duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

Resources

Artifacts (objects) can be defined as being a resource or not by setting the resource attribute to either true or false. In general artifacts that are worked on by agents are not considered to be a resource in an activity (a form, a fax). Artifacts that are used by an agent in an activity are considered to be resources (a fax machine, a telephone).

It is possible to associate artifacts with activities for statistical purposes and for the purpose of generating object flows by defining them in the list of resources for an activity. Artifacts that are defined as resources are also called resource objects. Resource objects associated with activities will only collect statistics and will not be used for the object flow generation. Artifacts which are defined not to be a resource and which are associated with an activity are also called touch objects. Touch objects should be associated with one or more conceptual object(s) for object flow purposes and statistical purposes.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name.

5. The minimum duration of the activity defines the minimum duration in seconds.

6. The maximum duration of the activity defines the maximum duration in seconds.

2.23 MOVE ACTIVITY (MOV)

2.23.1 Description

A move activity is a primitive activity but is used to move an agent from its current location to the location as specified in the activity. The move activity first checks whether the duration is specified for the move activity. If the specified duration is larger than 0 it will use that duration for the move. If no duration is specified or the specified duration is 0 then the simulation engine tries to determine its duration by calculating the time it takes to move from the current location to the goal location using the shortest path specified between the two location. If no paths are specified between the locations then the duration of the move activity will be 0.

When the move activity is started the agent or object being moved will be location-less. All of the facts and beliefs about the location of the agent will be retracted as well as the location facts and beliefs of the agents or objects contained by the agent or object. By default all agents in the same location as the start location of the move will detect that the agent/object is leaving and will retract the beliefs about the location of those agents and objects. It is possible to specify explicitly in which areas agents will detect the moving agent leaving the location by specifying a list of areas in the detectDepartureIn property for the move activity. By default all agents in the specified areas and their sub areas will detect the agent leaving and will therefore retract the location beliefs about that agent and the agents/objects being carried. If the detectDepartureInSubAreas is false then only agents in the specified areas will detect the departure, not the agents in the sub areas of the specified areas.

When the activity is interrupted or impassed the agent or object will still remain location-less. It is possible that while the activity is interrupted the agent or object was moved to another location. If the agent or object moves while the activity was interrupted the duration of the interrupted activity will be recalculated as if the move starts all over again. If the agent or object did not move while interrupted then the activity will resume where it left of, the duration will not be recalculated, the activity will be active for the remaining duration.

On arrival in the goal location a fact and belief will be asserted for the traveling agent or object about its new location. For all contained agents/objects a fact is asserted about their new location, the contained agents will also get a belief about their new location. By default the agents in the goal location will detect the traveling agent as well as any contained agents. The traveling agent and contained agent(s) will detect all the agents and objects in the goal location. For any of the contained objects the traveling agent will get a belief of the object if the agent already knew about the location of the contained object before it started the move. The model builder can control who detects the arrival by specifying a list of areas in the detectArrivalIn property for the activity. Only the agents located in the specified list of areas and their sub-areas will detect the arrival of the agent and its contained agents. If the detectArrivalInSubAreas property is set to

false then the agents located in the sub areas of the specified areas will not detect the arrival of the agent and its contained agents.

2.23.2 Syntax

move-activity ::= **move** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
location : [ARE.area-name | PAC.param-name] **;**
{ **detectDepartureIn :** [ARE.area-name |
PAC.param-name]
[**,** [ARE.area-name | PAC.param-name]* **;** }
{ **detectDepartureInSubAreas :** [ID.truth-value |
PAC.param-name] **;** }
{ **detectArrivalIn :** [ARE.area-name | PAC.param-name]
[, [ARE.area-name | PAC.param-name]* **;** }
{ **detectArrivalInSubAreas :** [ID.truth-value |
PAC.param-name] **;** }
}

2.23.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for move activities. These input parameters can be used to make activities more generic. In the reference the values for the input parameters have to be passed. It is recommended to make the first parameter the goal-location of the move activity.

Priority

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.

Duration

Activities in general have duration. In case of the move activity it is not necessary to define a duration of the activity. The duration of the activity is calculated by the simulation engine using the path definitions defining the distance between two locations. The simulation will determine the shortest path of travel from the current location to the goal location and calculate the travel time based on the distance. The duration of the move activity will in that case be the same as the calculated travel time. It is possible however to still define the duration of the activity. If a duration is specified the travel time will not be calculated and ignored, the specified duration will be used by the simulation engine. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

DetectDepartureIn

The detectDepartureIn-property of the move activity specifies the areas for which the agents located in those areas will detect the moving agent leaving its current location. Those agents will retract the belief about the location of the moving agent if they have a belief about the location of the moving agent.

DetectDepartureInSubAreas

The detectDepartureInSubAreas property has as a value either true or false and specifies whether the departure of the agent is noticed by only the agents in the areas specified by the detectDepartureIn property or also the sub areas of the specified areas. If the value is true then not only will the agents located in the specified areas detect the agent's departure, also the agents in the sub areas of the specified areas will detect the departure. If the value is false the agents in the sub areas of the specified areas will not detect the departure. The default is true.

DetectArrivalIn

The detectArrivalIn-property of the move activity specifies the areas for which the agents located in those areas will detect the moving agent arriving in its destination location. Those agents will assert the belief about the new location of the moving agent.

DetectArrivalInSubAreas

The detectArrivalInSubAreas property has as a value of either true or false and specifies

whether the arrival of the agent is noticed by only the agents in the areas specified by the detectArrivalIn property or also the agents in the sub areas of the specified areas. If the value is true then not only will the agents located in the specified areas detect the agent's arrival, also the agents in the sub areas of the specified areas will detect the arrival. If the value is false the agents in the sub areas of the specified areas will not detect the arrival. The default is true.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
detectDepartureIn	=	<the start location of the moving agent>
detectDepartureInSubAreas	=	true
detectArrivalIn	=	<the destination location of the moving agent>
detectArrivalInSubAreas	=	true

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The location parameter must be of type <area-def-name>.

6. The parameter types for the 'detectDepartureIn' and 'detectArrivalIn' areas must be of type <area-def-name>, the values for the parameters assigned to the 'detectDepartureIn' and 'detectArrivalIn' property must be VAR.variable-name, PAR.parameter-name or ARE.area-name.

7. The detectDepartureInSubAreas and detectArrivalInSubAreas parameters must be of type boolean and its input values must be one of true or false.

8. The minimum duration of the activity defines the minimum duration in seconds.

9. The maximum duration of the activity defines the maximum duration in seconds.

2.24 CREATE AGENT ACTIVITY (CAA)

2.24.1 Description

A create agent activity is a primitive activity allowing to dynamically create one or more new agents. By default the activity creates only one agent. The user can also specify a quantity defining how many agents have to be created in the activity, this is useful if more than one agent has to be created in the activity without having to repeat the activity. When creating agent(s) the agents can be made a member of one or more groups. The groups have to be specified in the memberof property of the activity. The created agents will inherit all the behavior specified for the groups.

The user can specify when the actual creation has to take place by setting the 'when' value to either start or end. The created agent(s) are bound to the unbound variable specified in the destination property of the activity. If no destination variable is specified the binding does not take place.

After the agent is created the activity will determine whether it needs to place the object in a specific location. The activity checks the location property. If a location is specified the instance will be placed in that location. If no location is specified the agent remains location-less. When the agent is placed in the location a fact will be created about the location of the new agent and all agents in that location will detect the new agent's location (they will get a belief about the location of the new instance) and the newly created agent will detect all agents and objects in that location.

The newly created agent(s) will be initialized, initial beliefs will be asserted to the belief set of the agent and initial facts about the new agent will be asserted to the fact set of the world. The agent's work will be initialized.

The last thing the activity does is bind the unbound variable specified in the destination property with the new agent(s). If more than one agent is created the variable must be a collect-all variable. If the variable is for-one or for-each only the first agent will be bound to the variable.

2.24.2 Syntax

create-agent-activity ::= **create_agent** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{

{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
{ **memberof :** [GRP.group-name | PAC.param-name]
[**,** [GRP.group-name | PAC.param-name]*
] **;** }
{ **quantity :** [ID.unsigned | PAC.param-name] **;** }
{ **destination :** [PAC.param-name] **;** }
{ **destination_name :** ID.literal-symbol **;** }
{ **location :** [ARE.area-name |
PAC.param-name] **;** }
{ **when :** [*start* | *end* | PAC.param-name] **;** }
}

2.24.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for create-agent activities. These input parameters can be used to make activities more generic. In the reference the values for the input parameters have to be passed.

Priority

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.

Duration

Activities in general have duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set

to the maximum duration of the activity in seconds.

When

The when attribute defines when the actual action has to take place, at the 'start' of the activity or at the 'end' of the activity.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
memberof	=	BaseGroup
quantity	=	1
destination	=	none
destination_name	=	NoNameAgent
location	=	none
when	=	end

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The memberof parameter type has to be of type GRP.group-name and its input value must be one of VAR.variable-name or GRP.group-name.

6. The quantity parameter has to be of integer type and its input value must be one of VAR.variable-name or an integer value. The quantity must be 1 or more.

7. The destination parameter type has to be of type Grp.group-name and its input value must be an unbound variable (VAR.variable-name).

8. The location parameter type has to be of type ADF.area-def and its input value must be one of VAR.variable-name or ARE.area-name.

9. The when parameter must be of type *symbol* and its input values must be one of *start* or *end*.

10. The minimum duration of the activity defines the minimum duration in seconds.

11. The maximum duration of the activity defines the maximum duration in seconds.

2.25 CREATE OBJECT ACTIVITY (COA)

2.25.1 Description

A create object activity is a primitive activity allowing to dynamically create new (conceptual) objects or make copies of objects. The user can specify when the actual creation/copy has to take place by setting the 'when' value to either start or end. In either case, new or copy, a new (conceptual) instance is created and assigned to the unbound variable specified in the destination property of the activity.

The (conceptual) class from which an instance is created is determined differently between the new and copy action. The behavior of the two will therefor be discussed separately.

In case of a new action the activity will determine the class of the new instance from the source property. If the source specifies a class or conceptual class the activity will create an instance of that class. If the source specifies a (conceptual) object it will create an instance of the class of that (conceptual) object.

In case if a copy action the activity will determine the class of the new instance from the destination property. The source will determine the (conceptual) class from the unbound variable. The activity then creates an instance of that class.

After the instance is created and if the instance is an object and not a conceptual object the activity will determine whether it needs to place the object in a specific location. The activity first checks the location property. If a location is specified the instance will be placed in that location. If the location property does not specify a location the activity will check the source property. If the source property is an object it will check if that object has a location. If so the object will be placed in the same location as the source object. When the instance is placed in the location a fact will be created about the location of the new instance and all agents in that location will detect the new instance's location (they will get a belief about the location of the new instance).

The newly created instance will be initialized, initial beliefs will be asserted to the belief

set of the object and initial facts about the new instance will be asserted to the fact set of the world. If the instance is an object with behavior its work will be initialized.

The activity will next determine whether the new (conceptual) instance needs to be made a part of conceptual objects. The activity first checks whether the conceptual-object property specifies any conceptual objects. If so it will make the new instance a part of those conceptual object(s). If no value is specified for the conceptual-object property the activity will check with the source. If the source property specifies an object or conceptual object, it will make the new instance a part of the same conceptual objects the source object is a part of. The appropriate aggregate relations will be generated as facts and beliefs.

If the new instance is created as a copy the activity will copy all beliefs of the source object to the new instance. This is only performed if both the source and new instance are objects, not when they are conceptual instances.

The last thing the activity does is bind the unbound variable specified in the destination property with the new instance.

2.25.2 Syntax

create-object-activity ::= **create_object** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
action : [*new* | *copy* | PAC.param-name] **;**
source : [CLS.object-class-name |
OBJ.object-name |
COC.conceptual-class-name |
COB.conceptual-object-name |
PAC.param-name] **;**
destination : [PAC.param-name] **;**
{ **destination_name :**ID.literal-symbol **;** }
{ **location :** [ARE.area-name |
PAC.param-name] **;** }
{ **conceptual_object :**
[COB.conceptual-object-name |
PAC.param-name]
[**,** [COB.conceptual-object-name |
PAC.param-name]] **;** }

{ **when :** [*start* | *end* | PAC.param-name] **;** }
}

2.25.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for create-object activities. These input parameters can be used to make activities more generic. In the reference the values for the input parameters have to be passed. It is recommended to make the first three parameters in a create-object activity:

1. action
2. source object
3. destination object

Priority

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.

Duration

Activities in general have duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

When

The when attribute defines when the actual action has to take place, at the ‘start’ of the activity or at the ‘end’ of the activity.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
name	=	no-name
location	=	<location of source object>
conceptual-object	=	<conceptual-object of source object>
when	=	end

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The action parameter must be of type *symbol* and its input values must be one of *new* or *copy*.

6. The source parameter type has to be of type CLS.object-class-name or COC.conceptual-class-name and its input value must be one of VAR.variable-name, CLS.object-class-name, OBJ.object-name, COC.conceptual-class-name, or COB.conceptual-object-name.

7. The destination parameter type has to be of type CLS.object-class-name or COC.conceptual-class-name and its input value must be an unbound variable (VAR.variable-name).

8. The location parameter type has to be of type ADF.area-def and its input value must be one of VAR.variable-name or ARE.area-name.

9. The conceptual-object parameter type has to be of type COC.conceptual-class-name or list-of COC.conceptual-class-name and its input value(s) must be one of VAR.variable-name, COB.conceptual-object-name or a list of any one of these elements.

10. The when parameter must be of type *symbol* and its input values must be one of *start* or *end*.

11. The minimum duration of the activity defines the minimum duration in seconds.

12. The maximum duration of the activity defines the maximum duration in seconds.

13. If the source of the create-object action is a class-name only the *new* action is allowed.

2.26 CREATE AREA ACTIVITY (CRA)

2.26.1 Description

A create area activity is a primitive activity allowing to dynamically create a new area. When creating a new area the area is by default made an instance of the area definition BaseAreaDef, the user can specify his/her own area definition as well. The area definition has to be specified in the 'instanceof' property. The area will inherit all the behavior specified for the area definition. The new area can be turned into a sub area for another area by assigning the parent area for the new area to the 'partof' property.

The new area can be inhabited by an initial set of agents or objects by specifying the agents and objects that are to be the inhabitants of the new area using the 'inhabitants' property.

The user can specify when the actual creation has to take place by setting the 'when' value to either start or end. The created area is bound to the unbound variable specified in the destination property of the activity. If no destination variable is specified the binding does not take place.

The newly created area will next be initialized, initial facts about the new area will be asserted to the fact set of the world.

The last thing the activity does is bind the unbound variable specified in the destination property with the new area.

2.26.2 Syntax

create-area-activity ::= **create_area** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }

{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
{ **instanceof :** [ADF.areadef-name | PAC.param-name] **;** }
{ **partof :** [ARE.area-name | PAC.param-name] **;** }
{ **inhabitants :** [AGT.agent-name | OBJ.object-name |
PAC.param-name]
[**,** [AGT.agent-name | OBJ.object-name |
PAC.param-name]*] **;** }
{ **destination :** [PAC.param-name] **;** }
{ **destination_name :** ID.literal-symbol **;** }
{ **when :** [*start* | *end* | PAC.param-name] **;** }
}

2.26.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for create-area activities. These input parameters can be used to make activities more generic. In the reference the values for the input parameters have to be passed.

Priority

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.

Duration

Activities in general have duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

Instance Of

The instanceof property specifies the area definition the new area is to be an instance of. It's the new area's parent 'class'.

Part Of

The partof property specifies the area the new area is to be made a part of. This makes the new area a sub area of the specified 'part of' area.

Inhabitants

The inhabitants property specifies the agents and objects that are to be located in the new area. If any of the inhabitants is located somewhere else then they will be relocated to the new area without any time consumption.

When

The when attribute defines when the actual action has to take place, at the 'start' of the activity or at the 'end' of the activity.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
instanceof	=	BaseAreaDef
partof	=	none
inhabitants	=	none
destination	=	none
destination_name	=	NewArea
when	=	end

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The instanceof parameter type has to be of meta-type AreaDef and its input value must be one of VAR.variable-name or ADF.areadef-name.

6. The partof parameter has to be of type ADF.areadef-name and its input value must be one of VAR.variable-name or ARE.area-name.

7. The inhabitant parameter has to be of type GRP.group-name or CLS.class-name and its input value must be one of VAR.variable-name, AGT.agent-name or OBJ.object-name.

8. The destination parameter type has to be of type ADF.areadef-name and its input value must be an unbound variable (VAR.variable-name).

9. The when parameter must be of type *symbol* and its input values must be one of *start* or *end*.

10. The minimum duration of the activity defines the minimum duration in seconds.

11. The maximum duration of the activity defines the maximum duration in seconds.

2.27 COMMUNICATE ACTIVITY (COM)

2.27.1 Description

The communicate activity is a primitive activity but allows for the communication of beliefs between the initiating agent or object and another agent or object. With the introduction of the communications library the communication can be more formalized by specifying a message object being an instance of the CommunicativeAct class for the transfer content. The communication activity in that case communicates the beliefs of the CommunicativeAct's belief set. The user can specify when the actual transfer has to take place by setting the 'when' value to either start or end.

2.27.2 Syntax

communicate-activity ::= **communicate** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{

{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
{ **type :** [*phone* | *fax* | *email* | *face2face* |
pager | *none* | PAC.param-name] **;** }
with : [[AGT.agent-name |
OBJ.object-name |
PAC.param-name]
[**,** [AGT.agent-name |
OBJ.object-name |
PAC.param-name]*] **;**
about : TDF.transfer-definition
[**,** TDF.transfer-definition]* **;**
{ **when :** [*start* | *end* | PAC.param-name] **;** }
}

2.27.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for communicate activities. These input parameters can be used to make activities more generic. In the reference the values for the input parameters have to be passed. It is recommended to make the first parameter(s) in a communicate activity the concepts with which is communicated.

Priority

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.

Duration

Activities in general have duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

Type

The type attribute defines what type of communication is used. The type can be one of phone, fax, email, face2face, pager, or none (meaning not specified).

About

The about attribute specifies what beliefs are to be communicated. The transfer definition specifies either the beliefs that are to be transferred or a CommunicativeAct defining the message to be transferred. If the transfer definition is a CommunicativeAct and the transfer action is 'send' then any destination agent or object can be specified. However if the transfer definition is a CommunicativeAct and the transfer action is 'receive' then the value of the 'with' attribute must be the CommunicativeAct declared in the transfer definition to indicate that the agent or object wishes to read the contents of the CommunicativeAct. An agent is not permitted to directly receive a CommunicativeAct from another agent. The agent in such a case is required to request the agent for the message and the receiving agent can then choose to send the requesting agent the desired CommunicativeAct.

When

The when attribute defines when the actual communication has to take place, at the 'start' of the activity or at the 'end' of the activity.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
type	=	none
when	=	end

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The type-parameter type has to be a literal-symbol and its values must be one of phone, fax, e-mail, face2face, pager, or none.

6. The with-parameter type has to be one of 'Agent' , CLS.object-class-name, list-of Agent or list-of CLS.object-class-name and its input value must be one of AGT.agent-name, OBJ.object-name, VAR.variable-name or a list of any of these elements. The with-parameter must be of type CommunicativeAct if the about-parameter specifies a transfer definition of type 'receive(<CommunicativeAct>)' and must be the exact same CommunicativeAct value as specified in the transfer definition.

7. The when parameter must be of type *symbol* and its input values must be one of *start* or *end*.

8. The minimum duration of the activity defines the minimum duration in seconds.

9. The maximum duration of the activity defines the maximum duration in seconds.

2.28 BROADCAST ACTIVITY (BCT)

2.28.1 Description

The broadcast activity is a primitive activity but allows for the initiator to broadcast information into a location. By default every agent in the same location as the initiator will receive the broadcasted belief. The model builder can instead also specify to what areas the beliefs are to be broadcast indicating whether their sub areas should be included or not. If the model builder specifies a list of areas then the broadcast will communicate all beliefs to be broadcast to all agents located in those specified areas. The model builder can specify when the actual transfer has to take place by setting the trigger value to either start or end. The user can in this case not specify with whom or what should be communicated.

2.28.2 Syntax

broadcast-activity ::= **broadcast** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
{ **type :** [*phone* | *fax* | *email* | *face2face* |
pager | *none* | PAC.param-name] **;** }
{ **to :** [ARE.area-name | PAC.param-name]
[**,** ARE.area-name | PAC.param-name]* **;** }
{ **toSubAreas :** [ID.truth-value | PAC.param-name] **;** }
about : TDF.transfer-definition
[**,** TDF.transfer-definition]* **;**
{ **when :** [*start* | *end* | PAC.param-name] **;** }
}

2.28.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for broadcast activities. These input parameters can be used to make activities more generic. In the reference the values for the input parameters have to be passed.

Priority

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.

Duration

Activities in general have a duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

To

The to-property of the broadcast activity specifies the areas in which a broadcast can be heard. This means that all agents located in the specified areas will receive the broadcast beliefs.

ToSubAreas

The toSubAreas property has as a value either true or false and specifies whether a broadcast is also heard in the sub areas of the areas specified in the to-property. If the value is true then not only will the broadcast go to all agents located in the specified areas, it will also go to all the sub areas of the areas specified in the to-property. The default is true.

When

The when attribute defines when the actual broadcast has to take place, at the 'start' of the activity or at the 'end' of the activity.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
to	=	<performing agent's location>
toSubAreas	=	true
when	=	end

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The parameter types for the 'to' areas must be of type <area-def-name>, the values for the parameters assigned to the 'to' property must be either VAR.variable-name, PAR.parameter-name or ARE.area-name.

6. The toSubAreas parameter must be of type boolean and its input values must be one of true or false.

7. The when parameter must be of type *symbol* and its input values must be one of *start* or *end*.

8. The minimum duration of the activity defines the minimum duration in seconds.

9. The maximum duration of the activity defines the maximum duration in seconds.

2.29 JAVA ACTIVITY (JAC)

2.29.1 Description

A java activity is a primitive activity but its actual behavior is specified in Java code. The java activity specifies the fully qualified name of the class that implements the IExternalActivity interface or extends the AbstractExternalActivity class. When the java activity is to be executed an instance of the class is created and the code for the activity executed. If the class extends the AbstractExternalActivity class then the java code will have access to the parameters passed to the activity, belief set of the agent or object and the fact set of the world and will be able to conclude new beliefs and facts.

2.29.2 Syntax

java-activity **::=** **java** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }

{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
class : [ID.literal-string | PAC.param-name] **;**
{ **when :** [*start* | *end* | PAC.param-name] **;** }
}

2.29.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for java activities. These input parameters can be used to make activities more generic. In the activity reference the values for the input parameters have to be passed. It is possible to pass unbound variables as parameters to the java activity that can then be bound to a value in the Java code called when the activity is executed. This is a good method to get results back from a Java activity.

Priority

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.

Duration

Activities in general have a duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

When

The when attribute defines when the actual Java code has to be executed, at the ‘start’ of the activity or at the ‘end’ of the activity.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
when	=	end

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The minimum duration of the activity defines the minimum duration in seconds.

6. The maximum duration of the activity defines the maximum duration in seconds.

7. The when parameter must be of type symbol and its input values must be one of start or end.

8. The class parameter must be of type string. The value for the class must specify the fully qualified name of the Java class. The class must be in the Java classpath. The class must implement the interface gov.nasa.arc.brahms.vm.interfaces. IExternalActivity or extend the abstract class gov.nasa.arc.brahms.vm.interfaces. AbstractExternalActivity.

2.30 GET ACTIVITY (GET)

2.30.1 Description

A get activity is a primitive activity that allows an agent or object to pick up or transfer one or more objects and/or agents, referred to as items, form an area, agent or object, to carry it with it while performing activities. The picked up agents and/or objects are said to be contained by the agent or object that picked up the agents/objects. An agent or object can put the picked up agents/objects down or transfer them to another agent or object by using the put activity (PUT).

When the agents and/or objects are picked up the get activity will generate the appropriate containment beliefs and facts. For each picked up agent/object the activity will generate a fact in the world and a belief for the agent/object performing the activity of the form:

<agent|object> contains <item>

If the item(s) are being transferred from another agent or object then the containment relation for that item for the agent or object the item is transferred from will be negated.

<agent|object> contains <item> is false

Whenever the agent or object moves to another location it will take the contained items with it to that new location. The contained items will no longer have a location until they are put down in a location, i.e. no location facts will be generated for contained items. If the moving agent/object knows of the location of the contained item before the move it will also know of the location of the contained item after the move, i.e. a location belief will be generated for the carried agent/object for the carrying agent/object.

2.30.2 Syntax

get-activity **::=** **get** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
items
{ source : [OBJ.object-name | AGT.agent-name |
ARE.area-name | PAC.param-name] **;** }

{ **when :** [*start* | *end* | PAC.param-name] **;** }
}

items ::= **items :** [param-name | OBJ.object-name | AGT.agent-name]
[**,** [param-name | OBJ.object-name | AGT.agent-name]***;**

2.30.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for get activities. These input parameters can be used to make activities more generic. In the activity reference the values for the input parameters have to be passed.

Priority

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.

Duration

Activities in general have a duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

Items

The items are the agents or objects that need to be picked up or transferred and are to be contained by the agent or object performing the get activity.

Source

The source specifies where the item(s) are being picked up or transferred from. The source can be either an area, another agent or another object. If no source is specified, then the agent's or object's current location is assumed.

When

The when attribute defines when the actual pick-up or transfer has to take place, at the 'start' of the activity or at the 'end' of the activity.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
source	=	<agent's of object's current location>
when	=	end

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The minimum duration of the activity defines the minimum duration in seconds.

6. The maximum duration of the activity defines the maximum duration in seconds.

7. The when parameter must be of type symbol and its input values must be one of start or end.

8. The parameter types for items must be of type <group-name> or list-of <group-name> or <object-class-name> or list-of <object-class-name>, the parameters assigned to the items must be either VAR.variable-name or AGT.agent-name or OBJ.object-name or a list of any of these three.

9. The parameter type for the source must be of type <areadef-name>, <class-name> or <group-name>. The value assigned to the parameter must be one of

VAR.variable-name, AGT-agent-name, OBJ-object-name, or ARE-area-name.

2.31 PUT ACTIVITY (PUT)

2.31.1 Description

A put activity is a primitive activity that allows an agent or object to put down (drop) or transfer one or more objects and/or agents, referred to as items, carried by the agent or object performing the activity. The items are either dropped in the current or specified location or transferred to another agent or object. The carrying agent/object will no longer carry the item while performing future activities. An agent or object must have picked up, retrieved or received the items earlier by using the get activity (GET) or put activity (in case of transfer to the agent). If the agent or object is not carrying an item being put down by the carrying agent/object, then the put activity will do nothing except take time if a duration is specified.

When the agents and/or objects are dropped or transferred, the put activity will generate the appropriate negation of the containment beliefs and facts. For each dropped agent/object the activity will generate a fact in the world and a belief for the agent/object performing the activity of the form:

<agent|object> contains <item> is false

If the items are being transferred to another agent or object then for those items a new containment relation is generated both in the form of facts and beliefs:

<agent|object> contains <item>

Whenever the agent or object moves to another location it will no longer take the dropped items with it to that new location. For each item dropped in a location the activity will generate a location fact and a location belief for the dropped item for each agent that is located in the same location. The location of the dropped item will be the same location as the carrying agent or object if no destination is specified and the specified location if a destination location is specified. If the carrying agent or object does not have a location when the item(s) are dropped then the dropped items will not get a location either and no location facts and beliefs are generated for the dropped items.

2.31.2 Syntax

put-activity **::=** **put** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{

{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
items
{ destination : [OBJ.object-name | AGT.agent-name |
ARE.area-name | PAC.param-name] **;** }
{ **when :** [*start* | *end* | PAC.param-name] **;** }
}

items ::= **items :** [param-name | OBJ.object-name | AGT.agent-name]
[**,** [param-name | OBJ.object-name | AGT.agent-name]***;**

2.31.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for get activities. These input parameters can be used to make activities more generic. In the activity reference the values for the input parameters have to be passed.

Priority

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.

Duration

Activities in general have a duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

Items

The items are the agents or objects that need to be dropped or transferred and are to be contained by the agent or object receiving the items or are to receive a location when dropped in a location. When dropped in a location they are no longer contained by anything.

Destination

The destination specifies where the item(s) are to be dropped or to what the items are to be transferred to. The destination can be either an area, another agent or another object. If no destination is specified, then the agent's or object's current location is assumed.

When

The when attribute defines when the actual drop has to take place, at the 'start' of the activity or at the 'end' of the activity.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0
max_duration	=	0
resources	=	none
destination	=	<agent's of object's current location>
when	=	end

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The minimum duration of the activity defines the minimum duration in seconds.

6. The maximum duration of the activity defines the maximum duration in seconds.

7. The when parameter must be of type symbol and its input values must be one of start or end.

8. The parameter types for items must be of type <group-name> or list-of <group-name> or <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or AGT-agent-name or OBJ.object-name or a list of any of these three.

9. The parameter type for the destination must be of type <areadef-name>, <class-name> or <group-name>. The value assigned to the parameter must be one of VAR.variable-name, AGT-agent-name, OBJ-object-name, or ARE-area-name.

2.32 GESTURE ACTIVITY (GAC)

2.32.1 Description

A gesture activity is a primitive activity used for indicating gesture changes made by an agent or object. This activity is primarily to be used in combination with a virtual reality environment such as OWorld/Adobe Atmosphere. The gestures as indicated by the gesture activity are visualized in the virtual reality environment provided that environment supports the specified gestures.

2.32.2 Syntax

gesture-activity **::=** **gesture** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **random :** [ID.truth-value | PAC.param-name] **;** }
{ **min_duration :** [ID.unsigned | PAC.param-name] **;** }
{ **max_duration :** [ID.unsigned | PAC.param-name] **;** }
{ PAC.resources }
gesture : [ID.literal-symbol | PAC.param-name] **;**
}

2.32.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for gesture activities. These input parameters can be used to make activities more generic. In the activity reference the values for the input parameters have to be passed.

Priority

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.

Duration

Activities in general have a duration. The duration of the activity can be defined to be a fixed amount of time. The random attribute 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 attribute has to be set to true, the min-duration attribute has to be set to the minimum duration of the activity in seconds and the max-duration attribute has to be set to the maximum duration of the activity in seconds.

Gesture

The gesture specifies the gesture made by the agent or object performing the activity. This is just a symbolic value and does not affect an agent or object's behavior in a Brahms simulation or agent-system. The virtual reality environment that interfaces with the Brahms environment interprets the symbolic value and visualizes the gesture if that environment supports the gesture. It is up to the model builder to come to an agreement with the developers of the virtual environment on what gestures will be supported and how they should be named.

Defaults

display	=	<activity-name>
priority	=	0
random	=	false
min_duration	=	0

max_duration = 0
resources = none

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

4. The parameter types for resources must be of type <object-class-name> or list-of <object-class-name>, the parameters assigned to the resources must be either VAR.variable-name or OBJ.object-name or a list of any of these two.

5. The minimum duration of the activity defines the minimum duration in seconds.

6. The maximum duration of the activity defines the maximum duration in seconds.

7. The parameter type for gesture must be of type symbol.

2.33 COMPOSITE ACTIVITY (CAC)

2.33.1 Description

A composite activity is an activity that has to be decomposed into more specific workframes. Unlike primitive activities no duration is specified for this activity. The duration of this type of activity depends on the workframes that will be worked on as part of the activity. Composite activities allow us to build a hierarchy of workframes.

2.33.2 Syntax

composite-activity ::= **composite_activity** PAC.activity-name **(**
{ PAC.param-decl [**,** PAC.param-decl]* } **)**
{
{ **display :** ID.literal-string **;** }
{ **priority :** [ID.unsigned | PAC.param-name] **;** }
{ **end_condition :** [**detectable** | **nowork**] **;** }
{ WFR.detectable-decl }

```
    { GRP.activities }
    { GRP.workframes }
    { GRP.thoughtframes }
}
```

2.33.3 Semantics

Declaration and reference

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 defined for the group, agent, class or object.

Parameters

It is possible to define input parameters for composite activities. These input parameters are in the first place used to pass on variables defined in a workframe for use in the workframes defined for the composite activity and second they can be used to make activities more generic. In the reference the values for the input parameters have to be passed.

Priority

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.

Duration

Composite activities themselves do not have a duration. Composite activities are decomposed into other workframes that a concept can work on as part of the activity which eventually result in a primitive activity to be executed having a specific duration.

End-condition

The end-condition of a composite activity defines how a composite activity can be ended. There are three possibilities:

1. Only end it on the basis of an end-activity detectable. The end-condition has to be set to ‘detectable’. When a detectable having an action type of ‘end-activity’ is detected the composite activity will be ended.

2. End the activity when there’s no more work to work on. The end-condition has to be set to ‘no-work’. If none of the workframes as defined in the composite-activities can be worked on the activity will be ended.

3. End the activity when there's o more work to work on, or when an end-activity detectable is detected. The end-condition has to be set to 'no-work' and an end-activity detectable needs to be defined for the composite activity. This case combines the first two cases.

Defaults

display = <activity-name>
priority = 0
end_condition = nowork

Constraints

1. The signature of an activity must be unique within the definition of a group, agent, class, object, or composite-activity. The signature consists of the name of the activity and the types of the argument list in the order the arguments are listed.

2. The input parameter type of a parameter defined in the declaration of an activity must be the same as the input value type or variable type in the reference of the activity.

3. The parameters assigned to any of the attributes must be of the correct type.

2.34 PRECONDITION (PRE)

2.34.1 Description

Preconditions control the activation of a workframe or thoughtframe. For a frame to become active the preconditions defined for the frame have to be satisfied. Preconditions are satisfied by either matching beliefs of an agent (if the workframes are defined for an agent or the frame is a thoughtframe) or by matching facts in the world (if the workframes are defined for an object). Preconditions can include variables as part of their matching of specific beliefs/facts.

2.34.2 Syntax

precondition **::=** { [**known** | **unknown**] } **(** novalcomparison **)** |
{ [**knownval** | **not**] } **(** evalcomparison **)**

novalcomparison **::=** BEL.obj-attr |
BEL.obj-attr REL.relation-name |
BEL.tuple-object-ref REL.relation-name

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

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

rel-comp **::=** BEL.obj-attr REL.relation-name BEL.obj-attr
{ **is** ID.truth-value } |
BEL.obj-attr REL.relation-name BEL.sgl-object-ref
{ **is** ID.truth-value } |
BEL.tuple-object-ref REL.relation-name BEL.sgl-object-ref
{ **is** ID.truth-value }

expression **::=** term | expression [**+** | **-**] term

term **::=** factor | term [***** | **/** | **div** | **mod**] factor

factor **::=** primary | factor **^** primary

primary **::=** **-** primary | element

element **::=** ID.number |
BEL.obj-attr |
VAR.variable-name |
unknown

2.34.3 Semantics

Precondition modifiers

A precondition may be defined with one of four modifiers: known, knownval, unknown, or not. The modifier for a precondition may be omitted; in this case, the modifier defaults to ‘known’ for a novalcomparison (no right-hand-side value) and to ‘knownval’ for an evalcomparison (a right-hand-side value is present). The modifiers have the following meaning.

known:

The modifier ‘known’ represents the possibility for an agent/object to have a belief/fact, but be unspecific as to whether the agent/objects knows the actual value.

For example, to evaluate the following precondition:

```
known (car1.color)
```

The simulation engine would simply check with the belief set of an agent to see whether the agent has a belief of the form:

```
car1.color = ?
```

If the engine finds a belief of this form, as it would when the following belief is present:

```
car1.color = red
```

then the engine would evaluate the precondition as true. A simple relational precondition like:

```
known (John is-the-son-of)
```

will evaluate to true when the engine finds any of the following beliefs (the right hand side and truth-value are completely ignored):

```
John is-the-son-of Bill is true
John is-the-son-of Bill is false
John is-the-son-of Jack is true
John is-the-son-of Jack is false
```

A more complex precondition like:

```
known (Cimap-order1.service-tech is-the-son-of)
```

will evaluate to true if the following beliefs are present:

```
Cimap-order1.service-tech = <agent1>
<agent1> is-the-son-of ?
```

where <agent1> is either an agent or object.

knownval:

The modifier ‘knownval’ (known value) means that the simulation engine must find a precise match for the precondition. The precondition is only true if matching beliefs/facts can be found for both the left hand side and the right hand side and if the relation between them is found as well. For an example of a complex precondition such as:

```
knownval(Cimap-order1.service-tech is-the-son-of Cimap-
order2.service-tech)
```

the following beliefs must be present:

```
Cimap-order1.service-tech = <agent1>
Cimap-order2.service-tech = <agent2>
<agent1> is-the-son-of <agent2>
```

When using variables, the engine will find as many matches as there are valid instantiations for the variables.

unknown (aka no-knowledge-of):

When the modifier 'unknown' is used, the simulation engine looks at the beliefs of the agent or facts in the world for objects for possible matches of the precondition. If there are any matches, the precondition evaluates to false, if no matches are found the precondition evaluates to true. The 'unknown' modifier can be interpreted as 'The agent/object has no beliefs/facts for <precondition>'. However, there are intricacies that need to be explained further.

When matching a precondition of the form: O_1A_1, the simulation engine looks for a belief of the form O_1A_1= ?. When a belief of the form O_1A_1= ? is found, the simulation engine interprets this to mean that the agent 'knows' about this object and attribute and thus the precondition is false.

When the precondition is of the form O_1 rel however, no matter what the right hand side or the truth of the relation is, the simulation engine will simply look up whether the agent/object possesses the belief/fact O_1 rel ?, and if so will evaluate the precondition to be false.

All other preconditions, require at least two steps for the simulation engine to determine the truth or falsehood of the precondition.

The form O_1A_1 rel requires the simulation engine to evaluate first the O_1A_1 then the result of the O_1A_1 (say O_2) with the relation. When a belief/fact for either the OA or for O_2 rel is not found, the precondition will be evaluated to true, if both are found the precondition will evaluate to false. For example given the following beliefs:

```
John.car = car1
car1 is-driven-by Jack
```

and the precondition:

```
unknown (John.car is-driven-by)
```

The simulation engine will evaluate the precondition to false, because it finds a belief for "John.car = ?" with the value car1 and it finds a belief for car1 is-driven-by. If either of the beliefs were not available the precondition would evaluate to true.

not (aka no-matching-beliefs):

Not works similar to unknown in that when there is no belief for the precondition specified with the not modifier the precondition will evaluate to true. If a belief does exist for the condition in the precondition then the not modifier works similar to the modifier knownval, but negates the resulting truth-value. The simulation engine will first try the

knownval for the precondition. If the precondition with the knownval modifier evaluates to true then the precondition with the not modifier evaluates to false and vice versa.

Precondition Evaluation Order

When variables are used in one or more precondition(s) the order in which the preconditions are specified is important. Depending on the order different outcomes are possible. The reason that precondition order is important is that the simulation engine is not a standard pattern matcher, but actually evaluates the preconditions causing potential assignments of values to variables. For example:

```
knownval(John.car = <car>)
```

The simulation engine tries to find a belief of the form 'John.car='. If it finds one stating 'John.car=car1' then it will assign the value car1 to the variable <car>.

If you were to write the following two preconditions in the following order the outcome might be unexpexted:

```
not(John.car = <car>)
knownval(<car> belongs-to <company>)
```

Suppose we have the following beliefs:

```
John.car = car1
car2 belongs-to nynex
```

The simulation engine will evaluate the first precondition first and first treat the precondition as a knownval therefor assigning the value 'car1' to the variable <car> because it matches the precondition with the belief 'John.car = car1'. Since this precondition is a not this precondition will always evaluate to false. The simulation engine would not continue but if it would then the simulation engine would verify the second precondition. It found a binding for the <car> variable and will substitute its value. It will then try to find a belief of the form 'car1 belongs-to <company>. It cannot find such a belief and therefor will fail the evaluation causing the frame not to be made available. However if you turn the preconditions around the outcome is different.

```
knownval(<car> belongs-to <company>)
not(John.car = <car>)
```

In this case <car> will be bound to car2, the first precondition evaluates to true. The second precondition will be evaluated and the simulation engine tries to find the belief 'John.car=car2', it cannot find such a belief but due to the 'not' modifier the precondition will evaluate to true causing the frame to be made available.

The precondition ordering will also be important when taking into account the use of variables. The next section discusses the distinction made by the simulation engine in the types variables and how that can affect the importance of the precondition order.

Pre-, Post- and Unassigned Variables

The simulation engine makes a distinction in how variables are bound in a frame. The three types of value assignments are pre-assigned, post-assigned and unassigned.

Unassigned variables are variables not used in any preconditions but that get their binding in an activity.

Pre-assigned variables are variables that get their values assigned in preconditions and get a pre-binding before the preconditions are evaluated. Pre-assigned variables are variables used in an object/attribute tuple (OA) or that are used in an object/relation tuple (OR) or object/relation/object triplet (ORO) where the object is a variable. In case of the ORO it could be one of the objects that is a variable or both. The simulation engine makes sure that for each OA, OR (with an (un)known modifier) and ORO there is at least one matching belief/fact before fully evaluating the preconditions. The variables used in these condition elements will get a pre-binding by matching the variables with the object values in the beliefs/facts. A final binding will be determined when the preconditions are evaluated.

Post-assigned variables are variables that get their values assigned in preconditions as well, but they will get a binding during the evaluation of the preconditions. These variables have no pre-binding like pre-assigned variables do. Post-assigned variables are the variables not used in any OA, OR, ORO condition elements but are usually ‘assignment’ variables specified on the left hand side or right hand side of a value condition, for example:

```
<myagent>.car = <mycar>
```

<myagent> is part of an OA pair and is therefor a post-assigned variable. <mycar> is not specified in any OA, OR, ORO condition element and is therefor a post-assigned variable. The simulation engine will have found potential matched for the OA and will have pre-bound the <myagent> variable. During the evaluation of the precondition the simulation engine will then for each value of <myagent> get the belief/fact that caused that value binding for <myagent> and retrieve its right hand side. Assume that the belief was:

```
John.car = car1
```

<myagent> is John and the right hand side is ‘car1’. The simulation engine will now assigne the value ‘car1’ to the variable <mycar> during the evaluation of the precondition.

Due to the distinction between pre- and post-assigned variables ordering of preconditions is also important if no conflicts are to occur with the constraints listed below. Assume a frame with the following preconditions:

```
knownval(<totalOrders> = <numVMOrders>+Builder.numOrders)
knownval(VM.numOrders = <numVMOrders>)
```

In this case the first precondition has two post-assigned variables <totalOrders> and <numVMOrders>. The simulation engine can resolve Builder.numOrders to a value but cannot resolve the values for the post-assigned variables. This would be an endless list of possible values. The simulation engine would report an error and fail the evaluation of the precondition. If the preconditions would now be reversed

```
knownval(VM.numOrders = <numVMOrders>)
knownval(<totalOrders> = <numVMOrders>+Builder.numOrders)
```

then the simulation engine resolves the <numVMOrders> post-assigned variable first, it will bind a value to it by finding a belief of the form VM.numOrders = ? and assigning the right hand side value to the variable. Then during the evaluation of the second precondition the <numVMOrders> variable will have a value bound to it that can be used together with the right hand side value of the belief Builder.numOrders = ? to assign a value to <totalOrders>. The evaluation of all preconditions will succeed and the frame can be made available.

Constraints

1. The left hand side attribute type and the right hand side value-type or right hand side attribute type of a value-expression must be the same, except in the case of an attribute being of a collection type and an index having been specific for the attribute, in that case any value can be assigned, if no index is specified however only unknown is valid or an expression resulting in the same collection type. If an object-attribute-index is used on the left or right hand side to resolve to a value then the type compatibility constraint is relaxed, the compiler will assume that a value of the correct type is returned to compare to the left or right hand side. The virtual machine will at run-time ensure that type compatibility holds, if not it will evaluate the precondition to false and generate a warning message in the log.

2. The left hand side and right hand side types in a relational expression must match the types as defined for the relation used in the relational expression. Only the right hand side can be defined to use an object-attribute-index. The compiler will allow this without type checking, the compiler will assume that a value of the correct type is returned for the relation. The virtual machine will at run-time ensure that type compatibility holds, if not it will evaluate the precondition to false and generate a warning message in the log. The use of object-attribute-index on the left hand side would resolve to an unknown type not allowing the compiler to verify whether the relation is declared for that type and is therefore not permitted.

3. Expressions must evaluate to a value of type int, long or double.

4. No nested expressions are allowed. The first release of the virtual machine will not support them yet. As soon as the virtual machine is able to support nested expressions, this constraint will be lifted.

5. Only one unbound post-assigned variable can be used in a precondition.

6. No unbound post-assigned variables can be used in a precondition using one of the relational operators '<', '>' or '!='.

7. No unbound post-assigned variables can be used in an expression.

Note that constraints 5 to 7 cannot be detected by the compiler but only by the simulation engine. The simulation engine will report errors in the error log and will fail the evaluation of the precondition in which these constraints are violated.

2.35 CONSEQUENCE (CON)

2.35.1 Description

A consequence is a logical statement for concluding/asserting new beliefs for an agent or object and/or facts in the world. When new facts are concluded about a Java object, the corresponding properties of the Java object are updated accordingly. In addition to explicitly specifying the new beliefs or facts to be concluded, a consequence may be used to query the beliefs of a Brahms object or the property values of a Java object. The results of the query will be concluded as new beliefs for the agent or object and/or facts in the world. For such a query on a Java object, the Brahms engine may attempt to register the Brahms agent or object as a property change listener with the Java object so that subsequent updates to the object's attributes from within Java code will be mirrored in Brahms beliefs and/or facts.

2.35.2 Syntax

consequence **::=** **conclude (**resultcomparison { con-config } **) ;**

resultcomparison **::=** old-style-comp | new-style-comp

old-style-comp **::=** **(** [result-val-comp | PRE.rel-comp] **)**

new-style-comp **::=** result-val-comp |
PRE.rel-comp |

BEL.tuple-object-ref { member-property-list }

result-val-comp **::=** BEL.obj-attr BEL.equality-operator JAV.expression

member-prop-list **::=** **{** member-property
[**,** member-property]* **}**

member-property **::=** ATT.attribute-name
{ **(** BEL.collection-index **)** }
{ **:** JAV.expression }

con-configuration **::=** **,** { **{** } config-prop-name **:** ID.literal
[, config-prop-name **:** ID.literal]* { **}** }

config-prop-name **::=** **fc** | **bc** |
factChangeSupport |
beliefChangeSupport |
changesupport

2.35.3 Semantics

Fact certainty (fc)

The fact certainty is a number ranging from 0 to 100 and represents the percentage of chance that a fact will be created based on the consequence. A fact certainty of 0% means that no fact will be created, 100% means that a fact will be created at all times.

Belief certainty (bc)

The belief certainty is a number ranging from 0 to 100 and represents the percentage of chance that a belief will be created based on the consequence. A belief certainty of 0% means that no belief will be created, 100% means that a belief will be created at all times.

New style syntax

The syntax of consequences has been updated and extended to provide a more compact way to conclude multiple attribute values for the same object and to support integration with the properties of Java objects. The new syntax and its semantics will be explained according to the major types of consequences that it supports.

Single comparison consequences

The old-style syntax supports the conclusion of facts and/or beliefs based on a single comparison expression. The determination of whether facts and/or beliefs are

concluded is based on the values of optional fact certainty and belief certainty configuration properties. For example, the old-style syntax for concluding that the John's full name is "John Doe" as both a belief and a fact is:

```
conclude((John.fullName = "John Doe"), bc:100, fc:100);
```

Note that parentheses are used to surround the value comparison and the bc:100 and fc:100 proprties are specified as additional arguments to the conclude statement. The old-style syntax is still supported, but a new-style syntax is now preferred for consistency with the expanded functionality of the conclude statement.

The new-style syntax for the above conclude statement is:

```
conclude(John.fullName = "John Doe", {bc:100, fc:100});
```

Note that parentheses are no longer used to surround the value comparison and the bc and fc configuration properties are enclosed by brackets.

Multiple attribute consequences

It is common to conclude values for more than one attribute of a concept. The new syntax provides a way of specifying the new values for the attributes in a single conclude statement. For example, to conclude that the John's full name is "John Doe" and his age is 43, we could use the following:

```
conclude(John{fullName:"John Doe", age: 43}, { bc:100, fc:100});
```

Note that a single comma is used to separate the subject and set of attribute/value pairs from the set of configuration property/value pairs.

For attributes that have a collection type, the attribute name can be followed by a collection index in parentheses to conclude a new value for the collection attribute at that index. For example, if car1 has a "parts" map attribute relating it to Brahms objects for its various component parts we could associate an "engine" object with the following:

```
conclude(car1 { parts("engine") : engine11}, {bc:100, fc:100} );
```

The new syntax also supports setting the property values of a Java object. When the object's class follows the naming conventions of the Java Bean Specification for properties. The properties of the object are accessed by public accessor methods with names set*PropertyName*, get*PropertyName*(for non-boolean properties), and is*PropertyName* (for Boolean properties), where *PropertyName* is the name of the attribute/property used in Brahms. Generalizing the Java Bean definition of a property, a property name may also be resolved to a method or field with that exact name (see the Attribute ATT section for details).

Unlike Brahms concepts or Java classes, Java objects don't have names that can be

used to refer to them in Brahms code. Instead, they are referenced using Brahms variables or parameters that have been declared with a Java type. For example, suppose a Java class named Person is defined in the package 'gov.nasa.arc.brahms.example' and defines properties "name" and "available" according to the Java Bean conventions for naming property accessor methods. (See Appendix A.2 for a listing of the Person source file.) In order to access objects of the Person class in a Brahms source file, the Brahms source file can have a jimport statement in the imports section (see the Import Declaration section):

```
jimport gov.nasa.arc.brahms.example.Person;
```

Then, in the body of a workframe, a local variable can be declared and initialized to a new instance of the Person class:

```
java(Person) oPerson = new Person();
```

Now that the oPerson variable references the new Person object, we can set its properties using a conclude statement:

```
conclude(oPerson {name:"John Doe",available:true},
            {bc:100, fc:100} );
```

The execution of the this conclude statement will not only set the values of the properties in the oPerson object, but may also create beliefs representing those property values in the agent executing the frame and as well as facts in the world state. Whether beliefs or facts are created is controlled by the values of the belief-certainty and fact-certainty. Whether the values of the properties of the Java object are set to the new values depends on whether facts are created as the state of the Java object should be mirrored by the facts in the world state whereas an agent's beliefs about the property values of an object can vary from the actual values.

Reading the attributes of a Java or Brahms object

A conclude may be used to read property values from a Java object and then create beliefs representing those property values in the executing Brahms agent or object and/or create facts in the world state. This can be done for all of the bean properties of a Java object with a conclude statement like the following:

```
conclude(oPerson, {bc:100, fc:100} );
```

As usual, whether beliefs or facts are created depends on the values of the belief-certainty and fact-certainty. If is not desirable to read the values of all the bean properties and "reify" them as Brahms beliefs or facts, the specific properties to be read can be listed in the conclude statement:

```
conclude(oPerson { name } {bc:100, fc:0} );
```

This conclude statement will read only the value of the name property of the Java object referenced by the oPerson variable and will create a belief of the form *object*.name = *namevalue*, but will not create any facts.

The new conclude syntax also supports reading the beliefs of a Brahms object and creating corresponding beliefs in the agent or object that is executing the frame. If car1 is a Car object then the conclude statement:

```
conclude(car1, {bc:100, fc:0} );
```

will retrieve all beliefs from the car1 object and create corresponding beliefs in the agent or object executing the frame. If a set of attribute names is specified in the conclude statement, as in

```
conclude(car1 { color, model }, {bc:100, fc:0} );
```

only the attribute value beliefs with those attribute names will be retrieved from the object and created as beliefs in the executing agent or object.

For a Brahms object, a collection index can also be given to retrieve only beliefs for a map attribute and the given index:

```
conclude(car1 { parts("engine") }, {bc:100, fc:0} );
```

Unknown and 'null' values

Whenever a null value is read from a Java bean property it will be represented as 'unknown' in the corresponding belief and or fact. Conversely, when 'unknown' is specified for the value of a property of a Java object as in the statement:

```
conclude(oPerson {name: unknown}, {bc:100, fc:100} );
```

then a reference-valued property of the Java object will be set to 'null'. If 'unknown' is specified for a property that has a numeric primitive type, such as int or float, the property will be set to a zero value of that type. Finally, if 'unknown' is specified for a property with the Boolean primitive type the property will be set to 'false'. In general, the value corresponding to 'unknown' is the default inititialization value used by Java for the property.

Detecting changes to properties of Java objects

When a value is changed in a Java object outside of Brahms or when just a method is called on a Java object to change the value Brahms would not know about the value change, i.e. any beliefs about the property will remain unchanged. If values change only from within Brahms just using the 'conclude' action will be sufficient.

An alternative is to re-conclude the properties that we know could have changed:

```
conclude(oPerson { name }, {bc:100, fc:100});
```

This triggers re-reading the value of the name property and concluding that value as both a belief and fact. The following would force a re-read of all properties of a Java object and cause the belief/fact assertion for all of them:

```
conclude(oPerson, {bc:100, fc:100});
```

To trigger an automatic notification the Java class must provide methods to register PropertyChangeListeners and send PropertyChangeEvents whenever the value of a property changes. Appendix A.3 lists the Java source for the Person class with modifications for property change support. Now whenever you perform a conclude:

```
conclude(oPerson, {bc:100, fc:100});
```

Brahms will detect whether property change support is available and register a property change listener with that object. Whenever the a PropertyChangeEvent is fired, Brahms will update both the beliefs and facts about the properties (depending on the values set for bc and fc at the time of assertion.

If you do not wish that automatic notification takes place or wish to control which values are automatically changed, the beliefChangeSupport and factChangeSupport configuration properties may be used

```
conclude(oPerson, {bc:100,
                   fc:100,
                   beliefChangeSupport: false,
                   factChangeSupport: true });
```

This will trigger automatic value changes only for facts. To disable both, a the changeSupport configuration property can be used:

```
conclude(oPerson, {bc:100, fc:100, changeSupport: false });
```

This will disable automatic value notification for both beliefs and facts. If bc or fc is set to 0 then the appropriate change support will automatically be disabled for that Java object.

Defaults

fc = 100
bc = 100

Constraints

1. In the comparison the left hand side attribute type and the right hand side value-type or right hand side attribute type of a value-expression must be the same, except in the case of an attribute being of a collection type. If the left hand side attribute is of a collection type and an index is specified any value type can be assigned. If no index is specified then only the value unknown or a value resolving to a collection type can be defined on the right hand side. If the right hand side is of a collection type and an index is specified then the compiler will assume that the type it resolves to is compatible with the left hand side type, the virtual machine will at run-time ensure type compatibility. If the right hand side resolves to a type that is incompatible with the left hand side type then an error is reported in the log and the consequence will fail.

2. In the comparison the left hand side and right hand side types in a relational expression must match the types as defined for the relation used in the relational expression. Only the right hand side can be defined to use an object-attribute-index. The compiler will allow this without type checking, the compiler will assume that a value of the correct type is returned for the relation. The virtual machine will at run-time ensure that type compatibility holds, if not it will generate an error message in the log and fail the consequence. The use of object-attribute-index on the left hand side would resolve to an unknown type not allowing the compiler to verify whether the relation is declared for that type and is therefore not permitted.

3. The values of fact-certainty and belief-certainty range from 0 to 100 and represent a percentage.

4. A consequence defined in the body of a thoughtframe can only conclude beliefs. The fact certainty argument will be ignored, a warning will be generated in case the belief-certainty is set to 0.

5. The values of factChangeSupport, beliefChangeSupport, and changeSupport must be of type boolean and be one of true or false.

2.36 DETECTABLE (DET)

2.36.1 Description

A detectable is a declarative statement defining first which state changes an agent or object can detect and second what action results from detecting the state change.

Detecting facts is a two-phase process. In the first phase, the agent or object detects the fact and the fact becomes a belief for the agent or object. No matter what the right hand side is in the form OA=V and OA=O the fact will become a belief for the agent or object. To make this behavior clearer, the detection condition may be stated without the equality and right hand side (OA) or the right hand side may be given as a wildcard character (OA=?). In the case of an ORO, the right hand side and truth value are also ignored in determining the matching facts and they also may be omitted from the condition (OR) or given as the wildcard character (OR?) to indicate that any fact with the specified relation and matching the left hand side should match the condition no matter what it has for a right hand side object or truth value.

In the second phase, the beliefs of the agent or object are matched with the detectable definition and if there is a positive match, the detectable action is executed. In contrast to the fact detection phase, the right hand side of a condition of the form OA=V, OA=O, or ORO is not ignored when matching beliefs. But if the right hand side of the condition is omitted or given as the wildcard '?' any right hand side value or object is allowed in matching beliefs.

Note that these two phases are independent for agents and objects, i.e. regardless of whether the fact is present in the world, the second phase is performed. This means that if the agent or object has received a matching belief through a communication, the belief will trigger the action of the detectable.

2.36.2 Syntax

detectable **::=** detectable-name **{**
{ **when** **(** [**whenever** | ID.unsigned] **)** }
detect **(** **(**resultcomparison **)** { **,** detect-certainty } **)**
{ **then** detectable-action } **;**
}

detectable-name **::=** ID.name

resultcomparison **::=** [detect-val-comp | detect-rel-comp]

detect-val-comp **::=** obj-attr |
obj-attr BEL.evaluation-operator PRE.expression |

obj-attr BEL.equality-operator ID.literal-symbol |
obj-attr BEL.equality-operator ID.literal-string |
obj-attr BEL.equality-operator sgl-object-ref

detect-rel-comp **::=** detectable-object REL.relation-name |
detectable-object REL.relation-name sgl-object-ref
{ **is** ID.truth-value } |

obj-attr **::=** detectable-tuple { **(** BEL.collection-index **)** }

detectable-tuple **::=** detectable-object **.** ATT.attribute-name

detectable-object **::=** BEL.tuple-object-ref | **<** ID.name **>**

sgl-object-ref **::=** BEL.sgl-object-ref | **<** ID.name **>** | **?**

detect-certainty **::=** **dc** **:** ID.unsigned

detectable-action **::=** **continue** | **impasse** | **abort** | **complete** | **end_activity**

2.36.3 Semantics

When

For each detectable it must be specified when the agent or object can detect a certain fact. There are two options:

whenever:

This means that the detectable is checked every time a new fact is asserted in the world and for an agent also every time a new belief is asserted.

at a specified time:

This specifies exactly when the detectable needs to be activated. The value must be any value of 0 or greater. This value specifies the time relative to the start time of the workframe in which the detectable is specified at which the detectable is to be activated.

< Class Variable >

Agents and objects do not always have the ability to know in advance about what concepts they can detect facts. To allow these agents and objects to detect facts about unknown concepts it is possible to define the class of concepts for which facts need to be detected. On the left hand side of the detectable condition instead of specifying the name of a variable, parameter or concept instance (agent, object, conceptual object, or area) the class of concept is specified in between the brackets < and > such as `<BaseGroup>`. This will have the agent or object detect all facts of which the left hand side concept instance is an instance of the specified class or an instance of any of its sub classes. The matches can be restricted by specifying a more specific class in the detectable condition. The agent or object will in that case only detect those instances that are instances of that class or any of its subclasses, but not any of its superclasses, even if the superclass defines that attribute and facts for concept instances of that super class exist. A bracketed class name may also be used on the right hand side of a detectable condition to indicate that, in the belief matching step, beliefs may have any right hand side concept that is an instance of that class or any of its subclasses.

Wildcard

A wildcard character, '?', may be used on the right-hand side of a detectable condition to indicate that the right-hand side value or object of a matching fact (in step one) or a matching belief (in step 2) is ignored. Normally, this may be stated more directly by simply omitting the operator and right-hand side of a value comparison or by omitting the right-hand side object of a relation comparison. However, the wildcard may occasionally be useful in a relation comparison together with 'is false' or 'is unknown' to match beliefs with truth values other than 'true'.

Detect-certainty

The detect-certainty is a number ranging from 0 to 100 and represents the percentage of chance that a fact will be detected based on the detectable. A detect-certainty of 0% means that the fact will never be detected and basically means that the detectable is switched off. A detect-certainty of 100% means that a fact will always be detected based on the detectable.

Detectable action

There are 5 different detectable actions possible:

continue:

Has no effect, only used for having agents or object detect facts and turn them into beliefs.

impasse:

Impasses the workframe on which the agent or object is working until the impasse is resolved.

abort:

Terminates the workframe on which the agent or object is working immediately.

complete:

Terminates the workframe on which the agent or object is working immediately, but still executes all remaining consequences defined in the workframe. All remaining activities are skipped.

end_activity:

This action type is only meaning full when used with composite activities. Causes the composite activity on which the agent or object is working to be ended.

Defaults

when	=	whenever
dc	=	100
action	=	continue

Constraints

1. In the comparison the left hand side attribute type and the right hand side value-type or right hand side attribute type of a value-expression must be the same. Object-attribute-index is considered type compatible with any type and is permitted on both the left and right hand side. For detectables the right hand side is ignored and therefore no errors or warnings will ever be generated even if the right hand side resolves to a value that is type incompatible with the left hand side. However for the trigger the right hand side is relevant and if the value resulting from evaluating the right hand side is type incompatible with the left hand side a warning is generated in the log and the detectable action is not executed.

2. In the comparison the left hand side and right hand side types in a relational expression must match the types as defined for the relation used in the relational expression.

3. An ID.name when used in a class variable must be the name of a concept class, i.e. a group, class, conceptual class or area definition.

4. The value of the detect-certainty ranges from 0 to 100 and represents a percentage.

5. The end-activity action type can only be used when a detectable is defined in a composite activity.

2.37 TRANSFER DEFINITION (TDF)

2.37.1 Description

A transfer-definition describes what belief(s) are to be communicated between two actors. The transfer-definition also describes whether the belief has to be sent to the actor with whom the initiating actor is communicating or whether the belief has to be received from the actor with whom it is communicating. The transfer definition can specify the exact belief(s) that are to be communicated or can specify a CommunicativeAct defining the message that is to be communicated. A CommunicativeAct is a much more formal representation of a message based on the CommunicativeAct specification defined by the Foundation for Intelligent Physical Agents (FIPA). When a CommunicativeAct is specified the transfer will consist of all the beliefs in that CommunicativeAct's belief set. An actor can only send a CommunicativeAct to another actor and never receive a CommunicativeAct from another actor (the initiating actor cannot force another actor to communicate). The receive action is therefore only valid to read all the beliefs of a CommunicativeAct.

2.37.2 Syntax

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

transfer-action **::=** **send** | **receive**

communicative-act **::=** OBJ.object-name | PAC.param-name

2.37.3 Semantics

Transfer-action

The transfer action defines the direction of the communication. The 'send' action states that belief(s) of the initiating agent or object matching the transfer definition are transferred from the initiating agent or object to the non-initiating agent or object. The 'receive' action states that belief(s) of the non-initiating agent or object matching the transfer definition are transferred from the non-initiating agent or object to the initiating agent or object. Note that if the transfer definition specifies a condition as an object/attribute tuple (OA=V) the right hand side value is ignored in the matching process. To make this behavior clearer, the condition may be stated without the equality and right hand side (OA) or the right hand side may be given as a wildcard character

(OA=?). In the case of an ORO, the right hand side is not ignored in the matching process but it may be omitted from the condition (OR) or given as the wildcard character (OR?) to indicate that any right hand side object should match the condition.

< Class Variable >

To provide more flexibility in determining the beliefs to be transferred in a communication a transfer definition condition may use bracketed class variables. A transfer definition condition, instead of specifying the name of a variable, parameter or concept instance (agent, object, conceptual object, or area), may specify the class of concept in between the brackets < and > such as `<BaseGroup>`. A class variable may be used anywhere a concept reference may appear in the condition. It may be used either on the left-hand or right-hand sides of the condition and may appear as a standalone object reference or paired with an attribute name. This will have the transfer condition match all beliefs that have a concept instance in a position corresponding to the class variable that is an instance of the specified class or an instance of any of its subclasses. The matches can be restricted by specifying a more specific class in the transfer definition condition. The agent or object will in that case only send beliefs involving those instances that are instances of that class or any of its subclasses, but not any of its superclasses, even if the superclass defines the attribute or relation and beliefs for concept instances of that super class exist.

Wildcard

A wildcard character, '?', may be used on the right-hand side of a transfer definition condition to indicate that the right-hand side value or object of a matching belief is ignored. Normally, this may be stated more directly by simply omitting the operator and right-hand side of a value comparison or by omitting the right-hand side object of a relation comparison. However, the wildcard may occasionally be useful in a relation comparison together with 'is false' or 'is unknown' to transfer beliefs with truth values other than 'true'.

Constraints

1. In the comparison the left hand side attribute type and the right hand side value-type or right hand side attribute type of a value-expression must be the same. Object-attribute-index is considered type compatible with any type and is permitted on both the left and right hand side. Since the right hand side of the condition is ignored, no errors or warnings will ever be generated event if the value resulting from the right hand side is type incompatible.

2. In the comparison the left hand side and right hand side types in a relational expression must match the types as defined for the relation used in the relational expression.

3. An ID.name when used in a class variable must be the name of a concept class, i.e. a group, class, conceptual class or area definition.

4. If a communicative-act is specified and the action is send then the object-name or value passed to the parameter identified by param-name must be an object instance that is an instance of brahms.communication.CommunicativeAct.

5. If a communicative-act is specified and the action is receive then the object-name or value passed to the parameter identified by param-name must be an object instance that is an instance of brahms.communication.CommunicativeAct and the value for the communication activities 'with' property must be identical to the communicative-act specified in the transfer definition to indicate the reading of the communicative-act's beliefs.

2.38 DELETE (DEL)

2.38.1 Description

The delete action is an action used to reclaim memory obtained for agents, objects and conceptual objects created at simulation/run-time, the delete action cannot be used on agents, objects, conceptual objects that are created at design-time. The deletion of an agent, object, or conceptual object results in the deregistration of the element from the directory service (in case of a distributed system), removal of references to the element for the calling agent/object, and removal of the element from the model. Elements declared as part of the model at compile time cannot be deleted. Delete operations on static model elements are no-ops and will cause the Brahms virtual machine to print a warning.**Element Creation and References**

Agents, objects and conceptual objects are created either when a model is loaded when they are defined as part of the model, when created using one of the create activities or created through the Java API. In distributed mode these elements are registered in the directory service. References to these elements are made in beliefs, facts, and as part of variable contexts maintained for frames defining variables that have their repeat property set to false. References to agents or objects are held using beliefs by either these agents or objects themselves (beliefs about themselves) or references to agents, objects, and conceptual objects are held by other agents or objects (beliefs about those elements). The world state maintains references to elements through facts. In distributed mode beliefs referencing these elements can be communicated to a remote agent or object. Objects that have no frames can be communicated by value to a remote agent or object effectively creating a duplicate copy of the object (this happens when communicating CommunicativeActs).

The delete action may also be applied to Java objects that are referenced as the object or value of beliefs or facts or as part of a frame variable context. Deletion of a Java

object only cleans up references to the Java object from the Brahms virtual machine. The Java object will not be garbage collected by the Java virtual machine if there are remaining references to the object from other Java objects.

Reference Counting

To identify when an object can truly be deleted the Brahms virtual machine maintains a reference count for every element per agent/object holding beliefs with a reference to the element.

The reference count will go up:

1. when an agent/object creates the element
2. when an agent/object asserts a belief with a reference to the element
3. when an external agent acquires a memory reference (IActiveInstance::acquireMemoryReference(), IConceptualInstance::acquireMemoryReference())

The count will go down:

1. when an agent invokes 'delete <element>'
2. when an external agent releases a memory reference (IActiveInstance::releaseMemoryReference(), IConceptualInstance::releaseMemoryReference())

Notes regarding the reference count:

1. The agent/object that creates the element and that asserts beliefs with references to the element will only increment the reference count by 1, not 2. Any agent/object can at maximum hold one reference to the element.
2. References held by remote agents/objects are not counted since we can't guarantee network availability, remote system uptime, and proper notifications of element deletion in a remote Brahms virtual machine.
3. Object copies (due to transfer by value) will have their own reference count in the Brahms virtual machine in which they were created as a copy and will therefore require separate deletion.
4. The element for which a reference count is being maintained has itself no impact on the reference count even when beliefs about the element are being asserted in that element about that element.

5. Multiple invocations of the delete action on the same element by the same agent/object will have no effects on the reference count. Only the first delete will cause the reference count to go down by 1.

6. If an external agent creates a new dynamic element automatically a reference to it for the external agent will be created! This means that the external agent **must** release any references to elements it creates using either (IActiveInstance::releaseMemoryReference() or IConceptualInstance::releaseMemoryReference().

7. When an external agent is notified of a belief (via invocation on onReceive) **no** automatic reference is obtained for any of the dynamic elements referenced in the belief. It is up to the external agent to acquire and release the necessary references.

Hard and Soft Reference

For the purposes of element deletion the Brahms virtual machine differentiates between hard and soft references.

A hard reference is a reference held by an agent/object that has frames. This agent/object is able to invoke a delete of the element at the appropriate time.

A soft reference is a reference held by an object that has no reasoning capabilities (a data object). It has beliefs in its beliefset that reference the element.

If there are no more hard references to an element and only soft references then the element will only be deleted when all agents/objects that have references to the element to be deleted themselves are marked for deletion and have no more hard references to those elements. If object O1 and object O2 have references to eachother in their beliefsets under normal circumstances neither would ever be deleted since the reference count never goes to 0. Assuming O1 and O2 are data objects then the references to eachother would be soft references. If no hard references exist for either of these two objects then the Brahms virtual machine will garbage collect them and perform the final deletion on both O1 and O2.

Results of a Delete

When an agent or object invokes a delete on an element E the Brahms virtual machine will take the following actions:

1. Retract all beliefs in which E is referenced, whether it be on the left hand side or right hand side of the belief.

2. Remove all previously executed frame contexts that hold a reference to E in a variable context when the frame has the repeat property set to false.

3. Reduce the hard reference count by 1 for the caller. If delete was invoked earlier the reference count remains unchanged.

When the Hard Reference Count goes to 0

When there are no soft references to the element E:

1. Retract all facts referencing E, whether it be on the left hand side or right hand side of the fact.

2. Deregister E from the directory service.

3. Remove E from the model.

When there are soft references to the element E:

1. Mark E for garbage collection.

2. Verify all soft references that reference E and delete any elements marked for garbage collection that only had a soft reference remaining to E.

Can an element be brought back?

Yes. If an object still has soft references the facts about the object are not yet retracted. This means that object detection is still possible. When an agent/object detects the object and asserts a new belief about the object the hard reference count is incremented and the object is no longer marked for garbage collection.

Remote References

If an element E is created in Brahms virtual machine VM1 and a belief referencing E is communicated to an agent/object in Brahms virtual machine VM2, E is deleted in VM1 and VM1 no longer holds any hard or soft references to E then the element will be deleted. The reference to E held in VM2 becomes stale. Any operations on E in VM2 can result in exceptions. Operations that cause exceptions:

1. reading of beliefs from E

2. communication of beliefs referencing E to a remote agent, an exception will be raised in the receiving virtual machine when it attempts to resolve E by looking it up in the directory service (E is longer registered in the directory service and cannot be located)

If E is a data object that was communicated by value then a separate reference count is maintained for this 'copy'. NOTE however that this copy is not registered in the directory service and if its original has been deleted then exceptions will be raised when:

1. beliefs referencing E are communicated to a remote agent/object. The exception is raised in the receiving virtual machine when it attempts to resolve E using the directory service (E is longer registered in the directory service and cannot be located)

Note that transmitting E in its entirety by value will cause no problems since when transmitting the entire object all relevant information to create a copy of it will be included as part of the message. This is generally only done with CommunicativeActs. It is safe to communicate a copy of a CommunicativeAct using a communicate activity to a remote agent even if its original has been deleted. Just communicating a belief about the CommunicativeAct to a remote agent however will raise an exception in the receiving virtual machine.

Any agent/object holding references to a copy of an element must call 'delete' for that element to ensure that it is deleted just like any other element.

Invoking delete on an element E for which the original does not exist in the virtual machine or where E is not a copy the delete will:

1. Retract all beliefs referencing E from the beliefset of the agent/object invoking delete

2. Remove all previously executed frame contexts that hold a reference to E in a variable context when the frame has the repeat property set to false.

There will be no reference count modifications since the original/copy is not held in that virtual machine.

2.38.2 Syntax

delete-action **::=** **delete** [VAR.variable-name | PAC.param-name] **;**

2.38.3 Semantics

Constraints

1. The variables and parameters referenced in a delete statement must be of a Group, Class,Conceptual Class, or Java type since delete is not supported on any other types.

2.39 JAVA EXPRESSION (JAV)

2.39.1 Description

A defined subset of Java statements and expressions may be used in the body of a workframe or thoughtframe. A workframe may include assignment statements and method invocation statements in addition to local variable declarations with Java initializer expressions. Both thoughtframes and workframes may use Java expressions in conclude statements for the right-hand values of attribute value comparisons.

A Java expression may contain Brahms object references (BEL.sgl-object-ref), and object attribute references (BEL.obj-attr). Java expressions are thus a generalization of Brahms expressions as defined in the Preconditions section (PRE.expression).

2.39.2 Syntax

assignment-statement **::=** assign-lhs **=** expression **;**

method-invocation-statement **::=** method-invocation **;**

assign-lhs **::=** [VAR.variable-name | PAR.parameter-name]
[**[** expression **]**]*

expression **::=** binary-expression **|**
method-invocation |
constructor-invocation |
array-creation |
array-access |
BEL.obj-attr |
BEL.sgl-object-ref |
ID.literal

binary-expression **::=** term [**+** | **-** | ***** | **/**] term

term **::=** method-invocation |
array-access |
BEL.obj-attr |
BEL.sgl-object-ref |
ID.number

method-invocation **::=** ID.qualified-name
{ **.** nonwild-type-arguments ID.name }
({ expression-list } **)**

constructor-invocation ::= **new**
{ nonwildcard-type-arguments }
ATT.java-class-or-interface-type-def
({ expression-list } **)**

nonwild-type-arguments ::= **<** ATT.java-ref-type-def
[**,** ATT.java-ref-type-def]* **>**

expression-list ::= expression [**,** expression]*

array-creation ::= **new** ATT.java-class-or-interface-type-def
[dimension-expressions [**[ ]**]* |
[**[ ]**]* array-initializer]

dimension-expressions ::= [**[** expression **]**]+

array-initializer ::= **{**
{ initializer-expression [, initializer-expression]* }
}

initializer-expression ::= expression | array-initializer

array-access ::= [VAR.variable-name | PAR.parameter-name]
[**[** expression **]**]+

2.39.3 Semantics

The Java expressions that may be used in Brahms are a subset of the full Java expression language. The key restrictions are the following:

1. The only kinds of Java expressions allowed are method invocations, constructor invocations, array creation expressions, array access expressions, simple binary arithmetic expressions, and literals. Array initializer expressions may be used in local variable declarations and array creation expressions. Java expressions may also include Brahms object references (BEL.sgl-object-ref), and object attribute references (BEL.obj-attr).

2. Instance method invocations may only be made where the target object reference is a Brahms variable or parameter. This restriction rules out a sequence of method invocations separated by '.' characters. Static method invocations may be made using the name of a Java class followed by '.' and the static method name.

3. Array access expressions must reference the array by means of a Brahms variable or parameter that has a Java array type.

4. Arithmetic expressions are limited to top-level expressions with two operands and the operators '+', '-', '*', and '/'. So there is currently no support for more than two operands or for nested arithmetic expressions.

Invocations of generic methods and constructors are supported. Type arguments may be supplied explicitly (see non-wildcard-type-arguments) or will be inferred by the Brahms compiler for most common cases.

Unknown and 'null' values

Whenever a null value is returned from a Java method invocation or array access in a context that requires a Brahms value it is converted to 'unknown'. Conversely, 'unknown' may be passed as an argument in a Java method invocation, constructor invocation, array creation, array initialization, or array access where there is an expected Java type for that argument. The 'unknown' value will be converted to the default initialization value that Java uses for variables and fields of that type. For reference types this will be the 'null' value, for numeric primitive types the value zero, and for the Boolean primitive type the value 'false'.

Multi-valued expressions and list conversion

Due to the effect of collect-all variables and parameters that are bound at runtime to collect-all variables, it is possible for an expression to evaluate to multiple values. In addition to collect-all variables and parameters, this is possible for object/attribute tuples that contain a collect-all variable or parameter, method invocations and array accesses whose target object or array is specified by a collect-all variable or parameter, and binary expressions one of whose terms can be multi-valued.

By default, if a multi-valued expression is used as an argument to a method invocation, constructor invocation, array creation, array access, or array initializer expression, only the first value is passed as the actual argument value. However, for arguments to method invocations and constructor invocations, it is sometimes useful to accumulate into a Java List all the values resulting from the evaluation of a potentially multi-valued argument expression. This will result in a single List value being passed as an argument to a method or constructor invocation whose corresponding parameter is typed as a List or Collection. The Brahms compiler determines that this "list conversion" should take place when the type of the method or constructor parameter is assignable from a List type whose type parameter is the type of the argument expression. This requires that the method or constructor parameter be typed as a List, Collection, or Iterable.

For assignment statements and local variable declarations with an initializer expression "list conversion" is also performed if the variable is given a List, Collection, or Iterable type that is assignable from a List type whose type parameter is the type of the right-hand expression. Otherwise, only the first value from the evaluation of the right-hand expression is assigned to the variable.

2.40 THE 'UNKNOWN' VALUE

The Brahms language provides support for the 'unknown' value and 'unknown' truth-value in beliefs, facts, preconditions, consequences, detectables and transfer definitions.

The 'unknown' value is a type independent value and can be assigned to any variable, parameter, attribute or relation. The model builder must be aware that the value 'unknown' can at runtime potentially be assigned to a property of an activity (such as max_duration). Properties such as max_duration do not allow the value 'unknown'. If such a value is assigned anyway at runtime then the virtual machine will generate a warning in the virtual machine's log and use the default value for that property. If however no default value is available for the property an error will be generated and the agent/object in which the error occurred will be halted.

The 'unknown' value can be tested for in preconditions, can be communicated in communication and broadcast activities, can be detected using detectables and can be concluded using consequences. Note that when the value 'unknown' is concluded for a relation that all the previously existing values for that relation are retracted. If an agent concludes that it doesn't know the value of the relation (unknown) it is necessary to remove all existing values for the relation. The reverse is valid as well. If an agent concludes a value for a relation other then 'unknown' and the value 'unknown' was already present for that relation, then the 'unknown' value will be retracted and replaced by the new value.

The value 'unknown' for the truth-value can be used to specify that the truth-value for a relation is unknown or uncertain. The 'unknown' truth-value can also be tested for in preconditions and can be used in consequences to conclude the truth-value.

Before reporting any errors to the support team when a model doesn't behave as expected make sure to check the vm.log file first. This log file might list warnings or errors that relate to assigning the value 'unknown' to a property of an activity that doesn't allow for the value unknown.

2.41 COLLECTION TYPES

The Brahms language lacked support for multi-valued attributes, only relations allowed for multiple values. To address this issue the language will now receive support in the form of collection types. The type currently supported in the language is the map type.

2.41.1 Map

The map collection type is a type that can only be used as part of the declaration of attributes. It allows for the assignment of multiple values to the attribute where each value is addressable using an index or key. The attribute's values are index/value or key/value pairs. The index or key can be any positive integer or string value.

2.41.1.1 Declaring Maps

A map is declared by declaring an attribute of type 'map'.

```
attributes:
  public map myMap;
```

2.41.1.2 Creating Map Values

A map value is an index/value or key/value pair. Where we specify index we also imply key. Map values like any other attribute values are represented as beliefs and/or facts. A map is created when at least one belief or fact exists defining a map value. The creation of these beliefs/facts is done like any other beliefs/facts, through initial beliefs/facts or by concluding the beliefs. The index of a map value can only be an integer or a string. The value as part of the index/value pair can be any value of any type.

Using initial beliefs/facts map values can be created as:

```
initial_beliefs:
  (current.myMap(1) = 10);                // int
  (current.myMap(2) = 100L);              // long
  (current.myMap(240) = 10.0);            // double
  (current.myMap(3) = true);              // boolean
  (current.myMap(4) = SomeSymbol          // symbol
  (current.myMap(5) = "Some String"       // string
  (current.myMap(6) = MyAgent             // agent
  (current.myMap(7) = MyObject            // object
```

```
    (current.myMap(8) = unknown             // unknown
    (current.myMap("intValue") = 10);       // int
    (current.myMap("longValue") = 100L);    // long
    (current.myMap("doubleValue") = 10.0);  // double
    (current.myMap("booleanValue") = true); // boolean
    (current.myMap("symbolValue") = SomeSymbol    // symbol
    (current.myMap("stringValue") = "Some String" // string
    (current.myMap("agentReference") = MyAgent    // agent
    (current.myMap("objectReference") = MyObject  // object
    (current.myMap("unknownValue") = unknown      // unknown

  initial_facts:
    (current.myMap(1) = 10);                // int
    (current.myMap(2) = 100L);              // long
    (current.myMap(240) = 10.0);            // double
    (current.myMap(3) = true);              // boolean
    (current.myMap(4) = SomeSymbol          // symbol
    (current.myMap(5) = "Some String"       // string
    (current.myMap(6) = MyAgent             // agent
    (current.myMap(7) = MyObject            // object
    (current.myMap(8) = unknown             // unknown
    (current.myMap("intValue") = 10);       // int
    (current.myMap("longValue") = 100L);    // long
    (current.myMap("doubleValue") = 10.0);  // double
    (current.myMap("booleanValue") = true); // boolean
    (current.myMap("symbolValue") = SomeSymbol    // symbol
    (current.myMap("stringValue") = "Some String" // string
    (current.myMap("agentReference") = MyAgent    // agent
    (current.myMap("objectReference") = MyObject  // object
    (current.myMap("unknownValue") = unknown      // unknown
```

The numeric indices do not need to be sequential, they can be any valid positive integer value. Any string can be used for the index of a map as well. Generally when a numeric index value is used we refer to it as an index, when a string index value is used we refer to it as a key. The numeric value can be referred to as a key and the string value as an index as well so in this document will just refer to all of them as an index.

Besides defining map values through initial beliefs and facts the map values can also be created using consequences. The following consequences show the most simplistic way to create new map values. Just like with initial beliefs and facts any value can be assigned to an index of a map.

```
conclude((current.myMap(100) = "Some Value"));
conclude((current.myMap("newValue") = 25.0));
```

It is also possible to change the values of an index for a map. The value can be changed to any other value, it does not necessarily have to be of the same value type as the previously assigned value. The following consequences show examples of this, the first consequence just changes the value, not the type (both strings), the second consequence changes the value type from double to boolean.

```
conclude((current.myMap(100) = "Some Other Value"));
conclude((current.myMap("newValue") = true));
```

When a value is replaced the virtual machine will first retract the old belief/fact for that index/value pair followed by asserting the new belief/fact.

Variables declared for the workframe or thoughtframe can also be used for either or both the index and value in a consequence. A variable that is intended to be used for the index must be of type int or string. The variables must also be bound to a valid value or the virtual machine will generate an error.

```
variables:
  foreach(string) sKey;
  foreach(symbol) sSymbolValue;
  foreach(int) nIntValue;
when( … )
do {
  conclude((current.myMap(25) = sSymbolValue));
  conclude((current.myMap(sKey) = nIntValue));
} // end do
```

Parameters declared for activities can also be used for either or both the index and value in a consequence. They work in the same way as variables. The parameter must be resolvable to a value, either an actual value passed as an argument for the parameter or a variable passed as an argument for the parameter that is bound to a value. An error is generated by the virtual machine if this is not the case. Parameters used for the index of a map value must be declared to be of type int or string.

```
composite_activity myActivity(string key,
                              symbol symbolValue,
                              int intValue) {
  workframes:
    workframe wf_myWorkframe {
      do {
        conclude((current.myMap(25) = symbolValue));
        conclude((current.myMap(key) = intValue));
      } // end do
    } // wf_myWorkframe
} // myActivity
```

It is also possible to assign values to an index of a map by resolving a belief or fact on the right hand side of a consequence. The right hand side can be an object/attribute tuple or the right hand side can consist of a reference to the index of a map that is resolved to a value. Again variables can be used anywhere in these consequences both on the left and right hand side.

```
conclude((current.myMap(30) = current.location));
conclude((current.myMap("value") = current.myMap("intValue"));
```

It is ***not*** possible to assign a map to an index of a map. The compiler will not permit this and will generate a compiler error. The following is not permitted:

```
conclude((current.myMap("someMap") = current.myMap));
```

2.41.1.3 Using Map Values in Preconditions

The use of map values in preconditions is no different then using regular values. Preconditions now allow you to specify an index for attributes of type map to allow for the proper matching of beliefs in the preconditions. Variables can be declared for both the index and the index value.

```
knownval(current.myMap(1) = 10)
knownval(current.myMap("stringValue") = "Some String")
knownval(current.myMap(1) = current.myMap("intValue")
knownval(current.myMap(nIndexVar) = 10)
knownval(nValue = current.myMap(nIndexVar))
knownval(current manages current.myMap("agentReference"))
knownval(current manages current.myMap(nIndexVar))
knownval(current.myMap = unknown)
known(current.myMap)
known(current.myMap(1))
known(current.myMap(nIndexVar))
```

All of these preconditions are valid preconditions. The virtual machine ensures that the variables used in these preconditions get the proper matching binding. A few notes about the known preconditions. In the case of known(current.myMap) the precondition will only evaluate to true if the belief (current.myMap = unknown) exists, if beliefs with index/value pairs exist for the map then that precondition evaluates to false. For the known precondition specifying the index the precondition will only evaluate to true if the index value matches with a belief that has that same index value or it evaluates to true for any index value that is matched for the index if a variable is specified for the index that has not been bound in earlier preconditions.

Note that since map values are not typed the virtual machine will do run-time type checking to make sure that any values resulting from the evaluation of a object-attribute-index expression is type compatible with the opposite side of the condition if an attribute or variable is specified or if a relation is used.

```
attributes:
  public int numEmployees;
relations:
  public BaseGroup manages;

knownval(current.numEmployees = current.myMap("stringValue"))
knownval(current manages current.myMap("intValue"))
```

These preconditions will generate warnings in the log file and will evaluate to false, since the map value for the “stringValue” index is a string and not an integer and since the map value for “intValue” is an integer and not an agent reference with the agent being a

member of BaseGroup.

2.41.1.4 Using Map Values in Concludes

Map values can be used in the right hand side of consequences to conclude a new value of a single valued attribute or to conclude a new value of a relation. The values of index/value pairs are not strongly typed for maps and any value is supported. However attributes that are not of a collection type and relations are strongly typed. Care is therefore required when assigning a value to an attribute using a map value.

```
attributes:
  public int numEmployees;

conclude((current.numEmployees = current.myMap("intValue"));
conclude((current.numEmployees = current.myMap("doubleValue"));
```

The first consequence will succeed since the agent currently beliefs the integer value 10 for the index "intValue" for myMap which is of the correct type for numEmployees. For the second one however the virtual machine will generate an error in the log file since double values are not type compatible with integers and the double value can therefore not be assigned to numEmployees.

```
relations:
  public BaseGroup manages;

conclude((current manages current.myMap("agentReference"));
conclude((current manages current.myMap("objectReference"));
```

In this case with the use of relations the first consequence again will succeed since the agent currently beliefs MyAgent as the value for the index "agentReference" for myMap which is an agent that is always a memberof BaseGroup and is therefore type compatible with the manages relation. The second consequence however returns MyObject from the map when using the index "objectReference" which is of type BaseClass and not type compatible with BaseGroup. The virtual machine will generate an error in the log file and not execute the consequence.

2.41.1.5 Copying Maps

The language makes it easy to copy the map values of one map to another map. This is done by using consequences.

Assume that we have two attributes declared myMap and myMapCopy both of type map. Also assume that we have assigned values to myMap already as we've shown in the examples for initial beliefs and assume that this consequence is executed for an agent.

```
conclude((current.myMapCopy = current.myMap));
```

This consequence will copy all beliefs with the index/value pairs for myMap to myMapCopy, creating the identical index/value pairs for myMapCopy. If myMapCopy had any indices that were identical to those in myMap then the values for those indices for myMapCopy will be replaced with those of myMap.

Before the conclude we have the following beliefs:

```
(current.myMap(1) = 10)
(current.myMap("stringValue") = "Some String")
(current.myMapCopy(1) = 50)
```

After the conclude we have the following beliefs:

```
(current.myMap(1) = 10)
(current.myMap("stringValue") = "Some String")
(current.myMapCopy(1) = 10)
(current.myMapCopy("stringValue") = "Some String")
```

The value 50 for index 1 for myMapCopy was retracted and the new value 10 was asserted and a new belief for index "stringValue" with value "Some String" was asserted for myMapCopy. Note that since we didn't specify belief or fact certainties both were defaulted to 100% and therefore also facts were asserted for the myMapCopy index/value pairs.

2.41.1.6 Clearing Maps

A consequence is also used to clear a map of all of its index/value pairs. This is done by concluding the value unknown for the map attribute without specifying an index.

```
conclude((current.myMap = unknown));
```

This retracts all index/value pair beliefs and facts for myMap and asserts the new belief/fact:

```
(current.myMap = unknown)
```

When a new index/value pair for this cleared map is concluded then first the unknown belief/fact for that map is retract followed by the assertion of the new index/value pair.

2.41.1.7 Communicating Maps/Map Values

Using broadcast and communicate activities it is possible to communicate beliefs from one agent to another. Since index/value pairs for maps are represented as beliefs they are communicated in the same way.

In the transfer definitions of the broadcast and communicate activities either single map values can be communicated or an entire map.

```
communicate communicateMapValues(string key, SomeGroup agt) {
  …
  about:
    send(current.myMap(1) = unknown),
    send(current.myMap("intValue") = unknown),
    send(current.myMap(key) = unknown),
    send(current.myMap = unknown),
    receive(agt.otherMap(1) = unknown),
    receive(agt.otherMap("intValue" = unknown),
    receive(agt.otherMap(key) = unknown),
    receive(agt.otherMap = unknown);
} // communicateMapValues
```

When an index is specified only the belief matching the OA(index) will be sent/received. The index can be specified using a parameter declared for the activity. If no index is specified for the map all beliefs about the map will be sent/received. This makes it easy to send/receive the entire contents of a map.

2.41.1.8 Detecting Map Values

Map values can also be detected in detectables and can be used to trigger detectable actions. They operate in a similar way as in preconditions. For detection of facts on the left hand side of the detectable condition is relevant. On that left hand side can be specified whether a single map value is to be detected by specifying the index for the map value to be detected or whether all map values need to be detected.

```
detect((current.myMap(1) = unknown));
detect((current.myMap("intValue") = unknown));
detect((current.myMap(someVariable) = unknown));
detect((current.myMap = unknown));
```

The first three detectable conditions detect a single value, with the third one using a variable to specify the index to be detected. That variable must have been bound in the preconditions. The last detectable condition detects all map values for myMap.

For triggers the right hand side value must match with the detected belief or belief matching the left hand side. The right hand side of the detectable can also specify a reference to a map value. The following are all valid detectable/trigger conditions.

```
detect((current.myMap(1) = current.myMap("intValue"));
detect((current.numEmployees = current.myMap(1));
detect((current manages current.myMap("agentReference"));
```

In the last two conditions there is a potential for type mismatch. For the condition using an attribute on the left hand side no warnings or errors are generated if there is a type mismatch. The virtual machine just compares the right hand side value of the belief matching (current.numEmployees = ?) with the right hand side value of the belief (current.myMap(1) = ?). If they match the detectable action is triggered, if not the action is not performed. For the condition using a relation a warning is generated if the map value is not type compatible with the relation due to the fact that relations are handled differently then attributes by the virtual

machine. The detectable action is not triggered in that case, only when the resulting reference matches that of the right hand side reference of the belief (current manages ?).

2.42 JAVA INTEGRATION

The Brahms language supports several forms of integration with the Java language:

1. Brahms has built-in types that correspond to the primitive Java types: byte, char, short, int, long, float, double, and Boolean. The Brahms string and symbol types correspond to the Java String type (see the Attribute (ATT) section).

2. A Brahms attribute, variable, or parameter can be declared to have a Java type that is either a Java class or interface. Values of the attribute, variable, or parameter are then required to be Java objects of that type (see the Attribute (ATT), Variable (VAR), and Primitive Activity (PAC) sections).

3. The bodies of Java workframes can contain references to a special kind of activity that is implemented by user-defined Java code (see the Java Activity (JAC) section).

4. Brahms supports beliefs and facts about the values of the properties of a Java object (see the Consequence (CON) section).

5. The bodies of Brahms workframes can contain occurrences of a subset of Java expressions, including method invocations, constructor invocations, array creation and references (see the Java Expression (JAV) section).

This section presents an extended example illustrating 1, 2, 4, and 5. The example is a simplistic hiring scenario in which a manager considers two persons for a position, selects one as a candidate, and refers that person to an HR manager. A manager is represented by an agent belonging to the Brahms Manager group, but the persons considered for the position are represented by instances of the Person Java class. Brahms code for the example is listed in Appendix A.1 and Java code for the Person class is listed in Appendix A.3.

In order to reference Java types without having to use fully qualified names, the Brahms source file uses several jimport statements (see the Import Statement (IMP) section):

```
jimport brahms.base.util.Log;
jimport gov.nasa.arc.brahms.example.Person;
jimport java.util.List;
jimport java.util.ArrayList;
jimport java.util.Collections;
```

These statements reference a Java utility class that provides logging functionality, the example Person class, and classes and interfaces from the Java Collections Framework that will be used in the example.

The Brahms Manager group declares two attributes:

```
group Manager {

  attributes:
    public java(Person) selected;
    public Manager hrRep;
```

The 'selected' attribute is declared to have a Java type; it will have as its value the candidate Person object selected for hiring. The 'hrRep' attribute has as its type the Brahms Manager group. A line manager will have as the value of its 'hrRep' attribute the HR manager to whom it should refer a selected candidate.

The main workframe in the example is 'create_and_refer', which is executed by a Manager to create two candidate Person objects, select one, and then refer the selected candidate to the appropriate HR manager:

```
workframes:
  workframe create_and_refer {
        variables:
              forone(Manager) hrguy;
        when ((current.hrRep = hrguy))
        do {
              java(Person) oCandidate = new Person("Wally");
              java(List<Person>)loCandidates
                    = new ArrayList<Person>();
              loCandidates.add(oCandidate);
              Log.info("%s added %s", current, oCandidate);
              oCandidate = new Person("Dilbert");
              loCandidates.add(oCandidate);
              Log.info("%s added %s", current, oCandidate);
              Collections.sort(loCandidates);
              java(Person) oSelected = loCandidates.get(0);
              conclude(oSelected {name,available},
                        {bc:100, fc:100});
              oSelected.setAvailable(true);
              refer(oSelected, hrguy);
        }
  } // create_and_refer
```

The 'create_and_refer' workframe will be executed for a Manager agent that has an associated HR manager. We will now consider each of the body elements of the 'create_and_refer' workframe in turn:

```
java(Person) oCandidate = new Person("Wally");
```

is an example of a local variable declaration with an initializer expression (see the Variable (VAR) section). The oCandidate variable is declared to have the Java type Person and is initialized to hold a reference to a newly created instance of Person with name "Wally".

```
java(List<Person>) loCandidates = new ArrayList<Person>();
```

Next, a local variable named 'loCandidates' is declared to have a Java type that is the generic type java.util.List with a type parameter that is the Person class. This is an example of how the Java Collections framework may be used with Brahms. The loCandidates variable is initialized with a newly created instance of the java.util.ArrayList implementation class. The loCandidates variable will be used to accumulate a List of candidate Person objects.

```
loCandidates.add(oCandidate);
```

adds the value of the oCandidate variable to the List of candidate Persons. This is an example of invoking a Java instance method on a Java target object – in this case, the List that is the value of the loCandidates variable.

```
Log.info("%s added %s", current, oCandidate);
```

invokes the 'info' static method of the Log class to write out a message to the console and to the Brahms log file. The 'info' and 'debug' static methods of the Log utility class use the Java format string conventions for their first argument.

The next three statements:

```
oCandidate = new Person("Dilbert");
loCandidates.add(oCandidate);
Log.info("%s added %s", current, oCandidate);
```

create a second candidate Person object named "Dilbert", add it to the list of candidates, and write out a log message. Note that, unlike a variable declared in the variables: section of a workframe, a local variable can be assigned a new value even if it already has a value.

To select from the list of candidate Person objects, we will rely on the fact that the Person class implements the Comparable interface and use a static sort method from the java.io.Collections class to order the list with the most promising candidate appearing first in the list. For this simplistic example, Person objects are ordered based on the lexicographic ordering of their names. An expanded example could use more realistic criteria for the ordering.

```
Collections.sort(loCandidates);
```

Once the list of candidates has been sorted, we can bind a local variable to the first element of the list:

```
java(Person) oSelected = loCandidates.get(0);
```

So far, the 'create_and_refer' workframe has created two new Java objects, added them to a list, and sorted the list. In order for Brahms to be able to trigger workframes and thoughtframes based on the properties of a Java object, the values of those properties must be mirrored as Brahms beliefs and/or facts. This can be done with a conclude statement:

```
conclude(oSelected {name,available}, {bc:100, fc:100});
```

As described in the Consequence (CON) section, this will retrieve the values of the name and available properties from the selected Person object and create corresponding beliefs in the Manager agent's belief set and facts in the world state. In addition, the agent will be registered as a PropertyChangeListener on that Person object so that future updates to the name and available properties will cause corresponding updates to the beliefs and facts.

When initially created, a Person object has the default value of false for the available property. To assert that the selected candidate is available, a second conclude statement could be used:

```
conclude(oSelected {available: true}, {bc:100, fc:100});
```

Instead, we illustrate the change detection functionality by making a direct call to a Java set method:

```
oSelected.setAvailable(true);
```

This call could also have been made from a Java activity. In either case calling the setAvailable method will cause the agent to be notified of the change so that it can update the belief and/or fact representing the value of the available property.

Now that a candidate Person has been selected and beliefs for its properties have been concluded, the selected candidate can be referred to the HR manager using an activity reference:

```
refer(oSelected, hrguy);
```

The refer activity has the definition:

```
activities:
communicate refer(java(Person) selected, Manager referTo) {
      max_duration: 100;
      type: phone;
      with: referTo;
      about: send(selected.name = ?),
             send(selected.available = ?);
} // refer
```

This illustrates how beliefs about the selected Java object can be communicated to another Brahms agent.

Finally, the update_selected workframe shows how beliefs about a Java object's properties can trigger further activity:

```
workframe update_selected {
      variables:
            foreach(java(Person)) person;
            foreach(string) name;
      when ((person.available = true) and
            (person.name = name))
      do {
            conclude((current.selected = person), bc:100, fc:0);
            Log.info("%s has a selected candidate: %s",
                  current, name);
            delete person;
      }
} // update selected
```

This workframe relies on the fact that only the selected candidate Person object has had beliefs created for its properties and had its available property set to true. When fired, the update_selected workframe concludes a value for the Manager agent's selected attribute and logs a message to that effect. It also calls the delete operation on the selected Person object (see the Delete (DEL) section). This retracts any beliefs that the agent has involving that particular Person object and, if no other agent has any beliefs remaining that involve that object, it also retracts any facts involving the object from the world state. The purpose of using the delete operation on a Java object is to remove the agent's references to the object when an agent is finished with it so that the Java object may eventually be garbage collected by the Java VM when no references to it remain.

3. KEYWORDS

This chapter gives an overview of all the keywords defined for the Brahms language. None of these keywords can be used as identifiers when building models.

ActiveClass
ActiveConcept
ActiveInstance
Agent
Area
AreaDef
Class
Concept
ConceptualClass
ConceptualConcept
ConceptualObject
GeographyConcept
Group
Object
abort
about
action
activities
agent
and
area
area1
area2
areadef
assigned
attributes
bc
boolean
broadcast
byte
char
class
collectall
communicate
complete
composite_activity
conceptual_class
conceptual_object
delete
destination
detect
detectable
detectables
detectArrivalIn
detectArrivalInSubAreas
detectDepartureIn
detectDepartureInSubAreas
display
distance
div
do
double
end_activity
end_condition
extends
factframe
false
fc
float
foreach
forone
gesture
get
group
impasse
import
icon
inhabitants
initial_beliefs
initial_facts
instanceof
int
is
java
jimport
known
move
name
new
nowork
not
object
package
partof
path
primitive_activity
priority
private
protected
public
put
quantity
random
receive
relation
relations
repeat
resource
resources
send
short
source
string
super
symbol
then
thoughtframe
thoughtframes
time_unit
to
toSubAreas
true
type
unassigned

conclude
continue
cost
create_agent
create_area
create_object
current
dataframe
dc
knownval
listof
location
long
map
max_duration
memberof
min_duration
mod
unknown
variables
with
when
whenever
workframe
workframes

4. BASE MODEL

Every Brahms model will import this Base Model into its model. No explicit import statement is required for importing the Base Model. In this chapter the Base Model is defined using the syntax as described in this document.

```
/*
 * The base model is the base for every model to be build in
 * Brahms. This model contains standard relations, attributes,
 * etc. required for a basic model. This model is imported/merged by
 * default.
 */
       /*****************************************************************
        *                  NASA CONFIDENTIAL INFORMATION                *
        *               (c)1997-2007 Nasa Ames Research Center          *
        *                        All Rights Reserved                    *
        *                                                               *
        *  This program contains confidential and proprietary information  *
        *  of NASA Ames Research Center, any reproduction, disclosure,     *
        *  or use in whole or in part is expressly prohibited, except as   *
        *  may be specifically authorized by prior written agreement.      *
        *                                                               *
        *****************************************************************/
       package brahms.base;



       /**
        * This group serves as the base for every group and agent in a brahms
        * model and provides groups and agents with a minimum work set.
        *
        * library brahms.base
        */
       group BaseGroup {

         attributes:
           public BaseAreaDef location;

         relations:
           public Group isMemberOf;
           public ActiveConcept contains;
           public Exception thrownException;

         activities:
         //
         // ***** Directory Activities *****
         //
         /**
          * The findAgent activity tries to locate and load an agent or agent reference.
          * A search strategy can be specified to indicate where the activity
          * can search for the agent, memory, disk and/or in the directory service.
          * <p>
          * Five different strategies are supported for finding an agent:
          * <ul>
          *    <li>MEMORY - only look into the model's cache for the instance
          *          don't load it from disk or the directory service</li>
          *    <li>MEMORY_DISK - first look for the instance in the model's cache,
          *          if it doesn't exist load it from the local disk</li>
          *    <li>MEMORY_DIRECTORY - first look for the instance in the model's cache,
          *          if it doesn't exist try to locate it in the directory
          *          service and load a reference to it <bold>(default)</bold></li>
          *    <li>MEMORY_DISK_DIRECTORY - first look for the instance in the model's
          *          cache, if it doesn't exist try to load if from the local disk,
          *          if it can't be found there, try to locate it in the directory
          *          service and load a reference to it</li>
          *    <li>MEMORY_DIRECTORY_DISK - first look for the instance in the model's
          *          cache, if it doesn't exist try to locate it in the directory
          *          service and load a reference to it, if it can't be found in the
          *          directory service try to load if from the local disk</li>
          * </ul>
          *
```

```
 * @param name the fully qualified name of the agent to be found</li>
 * @param strategy the search strategy to be used</li>
 * @return agt a variable to store the located agent</li>
 * @return success a boolean variable indicating success or failure for the search</li>
 */
java findAgent(string name, symbol strategy, ActiveInstance agt, boolean success) {
  max_duration: 1;
  class: "brahms.base.directory.FindAgent";
  when: start;
} // findAgent

  /**
   * The findObject activity tries to locate and load an object or object reference.
   * A search strategy can be specified to indicate where the activity
   * can search for the object, memory, disk and/or in the directory service.
   * <p>
   * Five different strategies are supported for finding an object:
   * <ul>
   *    <li>MEMORY - only look into the model's cache for the instance
   *          don't load it from disk or the directory service</li>
   *    <li>MEMORY_DISK - first look for the instance in the model's cache,
   *          if it doesn't exist load it from the local disk</li>
   *    <li>MEMORY_DIRECTORY - first look for the instance in the model's cache,
   *          if it doesn't exist try to locate it in the directory
   *          service and load a reference to it <bold>(default)</bold></li>
   *    <li>MEMORY_DISK_DIRECTORY - first look for the instance in the model's
   *          cache, if it doesn't exist try to load if from the local disk,
   *          if it can't be found there, try to locate it in the directory
   *          service and load a reference to it</li>
   *    <li>MEMORY_DIRECTORY_DISK - first look for the instance in the model's
   *          cache, if it doesn't exist try to locate it in the directory
   *          service and load a reference to it, if it can't be found in the
   *          directory service try to load if from the local disk</li>
   * </ul>
   *
   * @param name the fully qualified name of the object to be found</li>
   * @param strategy the search strategy to be used</li>
   * @return obj a variable to store the located object</li>
   * @return success a boolean variable indicating success or failure for the search</li>
   */
  java findObject(string name, symbol strategy, ActiveInstance obj, boolean success) {
    max_duration: 1;
    class: "brahms.base.directory.FindObject";
    when: start;
  } // findObject


//
// ***** Agent Activities *****
//

/**
 * The createExternalAgent activity dynamically creates a new
 * external agent. It requires the fully qualified Java class
 * name with the implementation for the Java agent and optionally
 * specify the base name to assign to the external agent. The
 * Brahms virtual machine will add additional information to the
 * name to guarantee name uniqueness.
 * <p>
 * The Java class must implement the Java interface:
 * <code>gov.nasa.arc.brahms.vm.api.jagt.IExternalAgent</code>
 *
 * @param classname the fully qualified Java classname with the agent's
 *     implementation (must implement IExternalAgent)
 * @param agentname the base name to be assigned to the agent
 * @return out the reference to the created agent
 */
java createExternalAgent(string classname, string agentname, ActiveInstance out) {
  class: "brahms.base.system.CreateExternalAgentActivity";
} // createExternalAgent

/**
 * The createExternalAgent activity dynamically creates a new
 * external agent. It requires the fully qualified Java class
 * name with the implementation for the Java agent and optionally
 * specify the base name to assign to the external agent. The
 * Brahms virtual machine will add additional information to the
 * name to guarantee name uniqueness.
 * <p>
```

```
 * The Java class must implement the Java interface:
 * <code>gov.nasa.arc.brahms.vm.api.jagt.IExternalAgent</code>
 *
 * @param classname the fully qualified Java classname with the agent's
 *     implementation (must implement IExternalAgent)
 * @param agentname the base name to be assigned to the agent
 * @return out the reference to the created agent
 * @return outfqn the qualified name assigned to the agent
 */
java createExternalAgent(string classname, string agentname, ActiveInstance out, symbol outfqn) {
  class: "brahms.base.system.CreateExternalAgentActivity";
} // createExternalAgent


//
// ***** Belief and Fact Activities *****
//

/**
 * The readBeliefs activity allows an agent to read beliefs
 * about a specified subject's attribute or relation.
 *
 * @param subject the Concept about which to read beliefs
 * @param attribute the name of the attribute or relation about which
 *        to read beliefs, this attribute or relation must be
 *        declared for the type of the declared subject
 */
java readBeliefs(Concept subject, string attribute) {
  class: "brahms.base.system.ReadBeliefsActivity";
} // readBeliefs

/**
 * The retractBelief activity allows an agent to retract beliefs
 * about a specified subject's attribute or relation.
 *
 * @param subject the Concept about which to retract a belief
 * @param attribute the name of the attribute or relation about which
 *        to retract beliefs, this attribute or relation must be
 *        declared for the type of the declared subject
 */
java retractBelief(Concept subject, string attribute) {
  // max_duration set to 1 and when set to start to ensure that
  // retraction takes place before the end of the activity, otherwise
  // if the activity is the last activity in a repeatable workframe
  // the belief retraction event is scheduled just after the end
  // workframe event, which results in the workframe becoming
  // available again and after being available it is made unavailable,
  // this context switch is unnecessary
  max_duration: 1;
  class: "brahms.base.system.RetractBeliefsActivity";
  when: start;
} // retractBelief

/**
 * The retractBeliefValue activity allows an agent to retract beliefs
 * about a specified subject's attribute or relation and value.
 *
 * @param subject the Concept about which to retract a belief
 * @param attribute the name of the attribute or relation about which
 *        to retract beliefs, this attribute or relation must be
 *        declared for the type of the declared subject
 * @param value the optional Concept value to be retracted when a
 *       relation is used, by default all values for the relation
 *       are removed
 */
java retractBeliefValue(Concept subject, symbol attribute, Concept value) {
  // max_duration set to 1 and when set to start to ensure that
  // retraction takes place before the end of the activity, otherwise
  // if the activity is the last activity in a repeatable workframe
  // the belief retraction event is scheduled just after the end
  // workframe event, which results in the workframe becoming
  // available again and after being available it is made unavailable,
  // this context switch is unnecessary
  max_duration: 1;
  class: "brahms.base.system.RetractBeliefsActivity";
  when: start;
} // retractBeliefValue

/**
```

```
    * The retractFact activity allows an agent to retract facts
    * about a specified subject's attribute or relation.
    *
    * @param subject the Concept about which to retract a fact
    * @param attribute the name of the attribute or relation about which
    *        to retract facts, this attribute or relation must be
    *        declared for the type of the declared subject
    */
   java retractFact(Concept subject, string attribute) {
     // max_duration set to 1 and when set to start to ensure that
     // retraction takes place before the end of the activity, otherwise
     // if the activity is the last activity in a repeatable workframe
     // the fact retraction event is scheduled just after the end
     // workframe event, which results in the workframe becoming
     // available again and after being available it is made unavailable,
     // this context switch is unnecessary
     max_duration: 1;
     class: "brahms.base.system.RetractFactsActivity";
     when: start;
   } // retractFact

   /**
    * The retractFactValue activity allows an agent to retract facts
    * about a specified subject's attribute or relation and value.
    *
    * @param subject the Concept about which to retract a fact
    * @param attribute the name of the attribute or relation about which
    *        to retract facts, this attribute or relation must be
    *        declared for the type of the declared subject
    * @param value the optional Concept value to be retracted when a
    *       relation is used, by default all values for the relation
    *       are removed
    */
   java retractFactValue(Concept subject, symbol attribute, Concept value) {
     // max_duration set to 1 and when set to start to ensure that
     // retraction takes place before the end of the activity, otherwise
     // if the activity is the last activity in a repeatable workframe
     // the fact retraction event is scheduled just after the end
     // workframe event, which results in the workframe becoming
     // available again and after being available it is made unavailable,
     // this context switch is unnecessary
     max_duration: 1;
     class: "brahms.base.system.RetractFactsActivity";
     when: start;
   } // retractFactValue

} // BaseGroup



/*********************************************************************
 *                   NASA CONFIDENTIAL INFORMATION                   *
 *               (c)1997-2007 Nasa Ames Research Center              *
 *                         All Rights Reserved                       *
 *                                                                   *
 *  This program contains confidential and proprietary information   *
 *  of NASA Ames Research Center, any reproduction, disclosure,      *
 *  or use in whole or in part is expressly prohibited, except as    *
 *  may be specifically authorized by prior written agreement.       *
 *                                                                   *
 *********************************************************************/
package brahms.base;

/**
 * class   BaseClass
 *
 * This class serves as the base for every class in a brahms model
 * and provides classes with a minimum work set.
 *
 * library brahms.base
 */
class BaseClass {

  attributes:
    public BaseAreaDef location;

  relations:
    public Class isInstanceOf;
```

```
  public ActiveConcept contains;
  public Exception thrownException;


activities:

//
// ***** Directory Activities *****
//
/**
 * The findAgent activity tries to locate and load an agent or agent reference.
 * A search strategy can be specified to indicate where the activity
 * can search for the agent, memory, disk and/or in the directory service.
 * <p>
 * Five different strategies are supported for finding an agent:
 * <ul>
 *    <li>MEMORY - only look into the model's cache for the instance
 *          don't load it from disk or the directory service</li>
 *    <li>MEMORY_DISK - first look for the instance in the model's cache,
 *          if it doesn't exist load it from the local disk</li>
 *    <li>MEMORY_DIRECTORY - first look for the instance in the model's cache,
 *          if it doesn't exist try to locate it in the directory
 *          service and load a reference to it <bold>(default)</bold></li>
 *    <li>MEMORY_DISK_DIRECTORY - first look for the instance in the model's
 *          cache, if it doesn't exist try to load if from the local disk,
 *          if it can't be found there, try to locate it in the directory
 *          service and load a reference to it</li>
 *    <li>MEMORY_DIRECTORY_DISK - first look for the instance in the model's
 *          cache, if it doesn't exist try to locate it in the directory
 *          service and load a reference to it, if it can't be found in the
 *          directory service try to load if from the local disk</li>
 * </ul>
 *
 * @param name the fully qualified name of the agent to be found</li>
 * @param strategy the search strategy to be used</li>
 * @return agt a variable to store the located agent</li>
 * @return success a boolean variable indicating success or failure for the search</li>
 */
java findAgent(string name, symbol strategy, ActiveInstance agt, boolean success) {
  max_duration: 1;
  class: "brahms.base.directory.FindAgent";
  when: start;
} // findAgent

/**
 * The findObject activity tries to locate and load an object or object reference.
 * A search strategy can be specified to indicate where the activity
 * can search for the object, memory, disk and/or in the directory service.
 * <p>
 * Five different strategies are supported for finding an object:
 * <ul>
 *    <li>MEMORY - only look into the model's cache for the instance
 *          don't load it from disk or the directory service</li>
 *    <li>MEMORY_DISK - first look for the instance in the model's cache,
 *          if it doesn't exist load it from the local disk</li>
 *    <li>MEMORY_DIRECTORY - first look for the instance in the model's cache,
 *          if it doesn't exist try to locate it in the directory
 *          service and load a reference to it <bold>(default)</bold></li>
 *    <li>MEMORY_DISK_DIRECTORY - first look for the instance in the model's
 *          cache, if it doesn't exist try to load if from the local disk,
 *          if it can't be found there, try to locate it in the directory
 *          service and load a reference to it</li>
 *    <li>MEMORY_DIRECTORY_DISK - first look for the instance in the model's
 *          cache, if it doesn't exist try to locate it in the directory
 *          service and load a reference to it, if it can't be found in the
 *          directory service try to load if from the local disk</li>
 * </ul>
 *
 * @param name the fully qualified name of the object to be found</li>
 * @param strategy the search strategy to be used</li>
 * @return obj a variable to store the located object</li>
 * @return success a boolean variable indicating success or failure for the search</li>
 */
java findObject(string name, symbol strategy, ActiveInstance obj, boolean success) {
  max_duration: 1;
  class: "brahms.base.directory.FindObject";
  when: start;
} // findObject
```

```
//
// ***** Agent Activities *****
//

/**
 * The createExternalAgent activity dynamically creates a new
 * external agent. It requires the fully qualified Java class
 * name with the implementation for the Java agent and optionally
 * specify the base name to assign to the external agent. The
 * Brahms virtual machine will add additional information to the
 * name to guarantee name uniqueness.
 * <p>
 * The Java class must implement the Java interface:
 * <code>gov.nasa.arc.brahms.vm.api.jagt.IExternalAgent</code>
 *
 * @param classname the fully qualified Java classname with the agent's
 *     implementation (must implement IExternalAgent)
 * @param agentname the base name to be assigned to the agent
 * @return out the reference to the created agent
 */
java createExternalAgent(string classname, string agentname, ActiveInstance out) {
  class: "brahms.base.system.CreateExternalAgentActivity";
} // createExternalAgent

/**
 * The createExternalAgent activity dynamically creates a new
 * external agent. It requires the fully qualified Java class
 * name with the implementation for the Java agent and optionally
 * specify the base name to assign to the external agent. The
 * Brahms virtual machine will add additional information to the
 * name to guarantee name uniqueness.
 * <p>
 * The Java class must implement the Java interface:
 * <code>gov.nasa.arc.brahms.vm.api.jagt.IExternalAgent</code>
 *
 * @param classname the fully qualified Java classname with the agent's
 *     implementation (must implement IExternalAgent)
 * @param agentname the base name to be assigned to the agent
 * @return out the reference to the created agent
 * @return outfqn the qualified name assigned to the agent
 */
java createExternalAgent(string classname, string agentname, ActiveInstance out, symbol outfqn) {
  class: "brahms.base.system.CreateExternalAgentActivity";
} // createExternalAgent


//
// ***** Belief and Fact Activities *****
//

/**
 * The readBeliefs activity allows an agent to read beliefs
 * about a specified subject's attribute or relation.
 *
 * @param subject the Concept about which to read beliefs
 * @param attribute the name of the attribute or relation about which
 *        to read beliefs, this attribute or relation must be
 *        declared for the type of the declared subject
 */
java readBeliefs(Concept subject, string attribute) {
  class: "brahms.base.system.ReadBeliefsActivity";
} // readBeliefs

/**
 * The retractBelief activity allows an agent to retract beliefs
 * about a specified subject's attribute or relation.
 *
 * @param subject the Concept about which to retract a belief
 * @param attribute the name of the attribute or relation about which
 *        to retract beliefs, this attribute or relation must be
 *        declared for the type of the declared subject
 */
java retractBelief(Concept subject, string attribute) {
  // max_duration set to 1 and when set to start to ensure that
  // retraction takes place before the end of the activity, otherwise
  // if the activity is the last activity in a repeatable workframe
  // the belief retraction event is scheduled just after the end
  // workframe event, which results in the workframe becoming
```

```
  // available again and after being available it is made unavailable,
  // this context switch is unnecessary
  max_duration: 1;
  class: "brahms.base.system.RetractBeliefsActivity";
  when: start;
} // retractBelief

/**
 * The retractBeliefValue activity allows an agent to retract beliefs
 * about a specified subject's attribute or relation and value.
 *
 * @param subject the Concept about which to retract a belief
 * @param attribute the name of the attribute or relation about which
 *        to retract beliefs, this attribute or relation must be
 *        declared for the type of the declared subject
 * @param value the optional Concept value to be retracted when a
 *       relation is used, by default all values for the relation
 *       are removed
 */
java retractBeliefValue(Concept subject, symbol attribute, Concept value) {
  // max_duration set to 1 and when set to start to ensure that
  // retraction takes place before the end of the activity, otherwise
  // if the activity is the last activity in a repeatable workframe
  // the belief retraction event is scheduled just after the end
  // workframe event, which results in the workframe becoming
  // available again and after being available it is made unavailable,
  // this context switch is unnecessary
  max_duration: 1;
  class: "brahms.base.system.RetractBeliefsActivity";
  when: start;
} // retractBeliefValue

/**
 * The retractFact activity allows an agent to retract facts
 * about a specified subject's attribute or relation.
 *
 * @param subject the Concept about which to retract a fact
 * @param attribute the name of the attribute or relation about which
 *        to retract facts, this attribute or relation must be
 *        declared for the type of the declared subject
 */
java retractFact(Concept subject, string attribute) {
  // max_duration set to 1 and when set to start to ensure that
  // retraction takes place before the end of the activity, otherwise
  // if the activity is the last activity in a repeatable workframe
  // the fact retraction event is scheduled just after the end
  // workframe event, which results in the workframe becoming
  // available again and after being available it is made unavailable,
  // this context switch is unnecessary
  max_duration: 1;
  class: "brahms.base.system.RetractFactsActivity";
  when: start;
} // retractFact

/**
 * The retractFactValue activity allows an agent to retract facts
 * about a specified subject's attribute or relation and value.
 *
 * @param subject the Concept about which to retract a fact
 * @param attribute the name of the attribute or relation about which
 *        to retract facts, this attribute or relation must be
 *        declared for the type of the declared subject
 * @param value the optional Concept value to be retracted when a
 *       relation is used, by default all values for the relation
 *       are removed
 */
java retractFactValue(Concept subject, symbol attribute, Concept value) {
  // max_duration set to 1 and when set to start to ensure that
  // retraction takes place before the end of the activity, otherwise
  // if the activity is the last activity in a repeatable workframe
  // the fact retraction event is scheduled just after the end
  // workframe event, which results in the workframe becoming
  // available again and after being available it is made unavailable,
  // this context switch is unnecessary
  max_duration: 1;
  class: "brahms.base.system.RetractFactsActivity";
  when: start;
} // retractFactValue
```

```
} // BaseClass



/*****************************************************************
 *                   NASA CONFIDENTIAL INFORMATION                *
 *               (c)1997-1999 Nasa Ames Research Center           *
 *                         All Rights Reserved                    *
 *                                                                *
 *  This program contains confidential and proprietary information  *
 *  of NASA Ames Research Center, any reproduction, disclosure,     *
 *  or use in whole or in part is expressly prohibited, except as   *
 *  may be specifically authorized by prior written agreement.      *
 *                                                                *
 *****************************************************************/
package brahms.base;


/**
 * This class serves as the base for every conceptual class in a
 * brahms model and provides conceptual classes with a minimum work set.
 *
 * library brahms.base
 */
conceptual_class BaseConceptualClass {

  relations:
    public BaseClass isAConceptualObjectOf;

} // BaseConceptualClass



/*****************************************************************
 *                   NASA CONFIDENTIAL INFORMATION                *
 *               (c)1997-2001 Nasa Ames Research Center           *
 *                         All Rights Reserved                    *
 *                                                                *
 *  This program contains confidential and proprietary information  *
 *  of NASA Ames Research Center, any reproduction, disclosure,     *
 *  or use in whole or in part is expressly prohibited, except as   *
 *  may be specifically authorized by prior written agreement.      *
 *                                                                *
 *****************************************************************/
package brahms.base;

/**
 * areadef BaseAreaDef
 *
 * This areadef serves as the base for every area definition in a
 * brahms model and provides conceptual classes with a minimum work set.
 *
 * library brahms.base
 */
areadef BaseAreaDef {

  relations:
    public BaseAreaDef isSubAreaOf;
    public BaseAreaDef hasSubArea;

} // BaseAreaDef
```

APPENDIX A: JAVA INTEGRATION EXAMPLE

A.1 Brahms Group and Agent Definitions

```
/*****************************************************************
 *                    NASA CONFIDENTIAL INFORMATION                *
 *                (c)2009 Nasa Ames Research Center          *
 *                         All Rights Reserved                     *
 *                                                                 *
 *  This program contains confidential and proprietary information  *
 *  of NASA Ames Research Center, any reproduction, disclosure,     *
 *  or use in whole or in part is expressly prohibited, except as   *
 *  may be specifically authorized by prior written agreement.      *
 *                                                                 *
 *****************************************************************/

package example;

/*****************************************************************
 * The RecruitExample model shows how Java objects can be created and
 * referenced in a Brahms model. This example defines a Manager group
 * with a communicate activty named refer for referring a candidate to
 * an HR manager, and workframes create_and_refer, report_person, and
 * update selected
 *
 * Build this example using the Brahms compiler V3.0 alpha
 * and then run the example using the Brahms virtual machine
 * V3.0 alpha. Make sure to include the path of this example
 * in the library path of the compiler and the virtual machine (modify
 * the vm.cfg file). Also make sure that the build path of the java
 * class Person is in the class path of the java virtual machine. You can
 * add the build path to the virtual machine's classpath by editing the
 * vm.cfg file. Add an entry 'class_path=<your path>'. If the entry
 * is already present, modify it to include your class path.
 *****************************************************************/

jimport brahms.base.util.Log;
jimport gov.nasa.arc.brahms.example.Person;
jimport java.util.List;
jimport java.util.ArrayList;
jimport java.util.Collections;

group Manager {

  attributes:
    public java(Person) selected;
    public Manager hrRep;

  activities:
  /** Communicates beliefs about the candidate Person to a Manager */
  communicate refer(java(Person) candidate, Manager referTo) {
    max_duration: 100;
    type: phone;
    with: referTo;
    about: send(candidate.name = ?),
       send(candidate.available = ?);
  } // refer

  workframes:
  /**
  * Creates two instances of the Java Person class, adds them to a list, sorts the list,
  * and the picks the first element of the list to refer to the manager's HR representative.
  */
  workframe create_and_refer {
    variables:
      forone(Manager) hrguy;
    when ((current.hrRep = hrguy))
    do {
         java(Person) oCandidate = new Person("Wally");
         java(List<Person>)loCandidates = new ArrayList<Person>();
         loCandidates.add(oCandidate);
         Log.info("%s added %s", current, oCandidate);
         oCandidate = new Person("Dilbert");
         loCandidates.add(oCandidate);
```

```
        Log.info("%s added %s", current, oCandidate);
        Collections.sort(loCandidates);
        java(Person) oSelected = loCandidates.get(0);
        conclude(oSelected {name,available}, {bc:100, fc:100});
        oSelected.setAvailable(true);
      refer(oSelected, hrguy);
    }
  } // create_and_refer

  workframe report_person {
    priority: 10;
    variables:
      foreach(java(Person)) person;
      foreach(string) name;
      foreach(boolean) available;
    when ((person.name = name) and
       (person.available = available))
    do {
      Log.info("%s knows about %s with availability %b", current, name, available);
    }
  } // report_person

  workframe update_selected {
    variables:
      foreach(java(Person)) person;
      foreach(string) name;
      when ((person.available = true) and
      (person.name = name))
    do {
      conclude((current.selected = person), bc:100, fc:0);
      Log.info("%s has a selected candidate: %s", current, name);
      delete person;
    }
  } // update selected
} // Manager

agent PointyHair memberof Manager {
  initial_beliefs:
    (current.hrRep = Catbert);
    (current.selected = unknown);
} // PointyHair

agent Catbert memberof Manager {
} // Catbert
```

A.2 Java Person Class Definition

```
/*****************************************************************
 *                   NASA CONFIDENTIAL INFORMATION                *
 *               (c)2009 Nasa Ames Research Center                   *
 *                        All Rights Reserved                      *
 *                                                                 *
 *  This program contains confidential and proprietary information  *
 *  of NASA Ames Research Center, any reproduction, disclosure,     *
 *  or use in whole or in part is expressly prohibited, except as   *
 *  may be specifically authorized by prior written agreement.      *
 *                                                                 *
 *****************************************************************/
package gov.nasa.arc.brahms.example;

/**
*  A simple class representing a a person who is a potential hire
*  with name and availability properties
*/
public class Person implements Comparable<Person> {

   /*
   * Constructors
   */
  /**
  * Construct a new Person object with default name and availability
  */
  public Person() {
  } // Person

  /**
   * Construct a new Person object with the given name
```

```
     *
     * @param name the person's name
     */
    public Person(String name) {
      m_sName = name;
    } // Person

    /*
     * Instance Attributes
     */
    /** The person's name */
    private String  m_sName = "Jack";
    /** Whether the person is currently available */
    private boolean m_bAvailable = false;

    /*
     * Instance Methods
     */
    /**
     * Set the person's name
     *
     * @param the String name to be set for the person
     */
    public void setName(String name) {
      m_sName = name;
    } // setName

    /**
     * Retrieve the person's name
     *
     * @return String the person's name
     */
    public String getName() {
      return m_sName;
    } // getName

    /**
     * Sets whether the person is available for hire
     *
     * @param avail true if available for hire, else false
     */
    public void setAvailable(boolean avail) {
      m_bAvailable = avail;
    } // setAvailable

    /**
     * Determines whether the person is available for hire
     *
     * @return boolean true if available for hire, else false
     */
    public boolean isAvailable() {
      return m_bAvailable;
    } // isAvailable

    /**
     * Compares this Person with the specified Person for order.  Returns a
     * negative integer, zero, or a positive integer as this Person is less
     * than, equal to, or greater than the specified Person
     * @param other the Person to be compared to
     * @return -1 if this Person comes before the specified Person, 0 if
     *    this Person is equal in the ordering, 1 if this Person comes
     *    after the specified Person
     * @see java.lang.Comparable#compareTo(java.lang.Object)
     */
    public int compareTo(Person other) {
      return m_sName.compareTo(other.m_sName);
    } // compareTo

    /**
     * Return a string representation of this Person
     *
     * @return String the string representing this Person
     */
    public String toString() {
      return m_sName;
    } // toString

  } // Person
```

A.3 Java Person Class with PropertyChangeSupport

```
/*****************************************************************
 *                   NASA CONFIDENTIAL INFORMATION                *
 *               (c)2009 Nasa Ames Research Center                    *
 *                        All Rights Reserved                      *
 *                                                                 *
 *  This program contains confidential and proprietary information  *
 *  of NASA Ames Research Center, any reproduction, disclosure,     *
 *  or use in whole or in part is expressly prohibited, except as   *
 *  may be specifically authorized by prior written agreement.      *
 *                                                                 *
 *****************************************************************/
package gov.nasa.arc.brahms.example;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

/**
*  A simple class representing a a person who is a potential hire
*  with name and availability properties. The class has been augmented
*  with PropertyChangeSupport.
*/
public class Person implements Comparable<Person> {

   /*
   * Constructors
   */
  /**
  * Construct a new Person object with default name and availability
  */
  public Person() {
  } // Person

  /**
   * Construct a new Person object with the given name
   *
   * @param name the person's name
   */
  public Person(String name) {
    m_sName = name;
  } // Person

  /*
   * Instance Attributes
   */
  /** The person's name */
  private String  m_sName = "Jack";
  /** Whether the person is currently available */
  private boolean m_bAvailable = false;
  /**
   * Contains a PropertyChangeSupport instance to support notification of listeners
   * when Person properties are changed.
   */
  private PropertyChangeSupport m_oPCS = new PropertyChangeSupport(this);

  /*
   * Instance Methods
   */
  /**
   * Set the person's name
   *
   * @param the String name to be set for the person
   */
  public void setName(String name) {
    String sOldValue = m_sName;
    m_sName = name;
    m_oPCS.firePropertyChange("name", sOldValue, m_sName);
  } // setName

  /**
   * Retrieve the person's name
   *
   * @return String the person's name
   */
  public String getName() {
    return m_sName;
  } // getName
```

```
  /**
   * Sets whether the person is available for hire
   *
   * @param avail true if available for hire, else false
   */
  public void setAvailable(boolean avail) {
    boolean bOldValue = m_bAvailable;
    m_bAvailable = avail;
    m_oPCS.firePropertyChange("available", bOldValue, m_bAvailable);
  } // setAvailable

  /**
   * Determines whether the person is available for hire
   *
   * @return boolean true if available for hire, else false
   */
  public boolean isAvailable() {
    return m_bAvailable;
  } // isAvailable

  /**
   * Adds a PropertyChangeListener to the list of property change listeners.
   * The listener is registered for all properties.
   *
   * @param listener a PropertyChangeListener instance that implements a propertyChange method
   * @see java.beans.PropertyChangeSupport#addPropertyChangeListener(PropertyChangeListener)
   * @see java.beans.PropertyChangeListener
   */
  public void addPropertyChangeListener(PropertyChangeListener listener) {
    m_oPCS.addPropertyChangeListener(listener);
  } // addPropertyChangeListener

  /**
   * Removes a PropertyChangeListener for a PropertySupport bean's list of property change listeners.
   * The listener is unregistered for all properties.
   *
   * @param listener a PropertyChangeListener instance that implements a propertyChange method
   *             @see             java.beans.PropertyChangeSupport#removePropertyChangeListener(String,
PropertyChangeListener)
   * @see java.beans.PropertyChangeListener
   */
  public void removePropertyChangeListener (PropertyChangeListener listener) {
    m_oPCS.removePropertyChangeListener(listener);
  } // removePropertyChangeListener

  /**
   * Compares this Person with the specified Person for order.  Returns a
   * negative integer, zero, or a positive integer as this Person is less
   * than, equal to, or greater than the specified Person
   * @param other the Person to be compared to
   * @return -1 if this Person comes before the specified Person, 0 if
   *    this Person is equal in the ordering, 1 if this Person comes
   *    after the specified Person
   * @see java.lang.Comparable#compareTo(java.lang.Object)
   */
  public int compareTo(Person other) {
    return m_sName.compareTo(other.m_sName);
  } // compareTo

  /**
   * Return a string representation of this Person
   *
   * @return String the string representing this Person
   */
  public String toString() {
    return m_sName;
  } // toString

} // Person
```