



Brahms Tutorial

TM01-0002

Version 1.2

30 March 2011

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2001-2011 NASA Ames Research Center. All Rights Reserved.

**Technical Memorandum
TM01-0002**

BRAHMS TUTORIAL

VERSION 1.2

CONTACT

Brahms Contact

William Clancey – Project Lead (650) 604-2526

ABSTRACT

This document is a guide to programming in Brahms, an agent-oriented modeling language.

DATE: 30 March 20

KEYWORDS: Brahms, Tutorial

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

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2001-2011 NASA Ames Research Center. All Rights Reserved.

CONTRIBUTORS

Alessandro Acquisti

William J. Clancey

Ron van Hoof

Mike Scott

Maarten Sierhuis

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

APPROVED

William Clancey

Date

Project Lead

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

Revision History

Version	Contact	Action	
Version 0.1 Draft 02/01/2001	Alessandro Acquisti 510/823-5008	New	Initial draft.
Version 0.2 Draft 02/21/2001	Alessandro Acquisti 510/823-5008	Add	Added sections Q&A and proposed schedule of Tutorial.
Version 0.4 Draft 03/15/2001	Alessandro Acquisti 510/823-5008	Add, Change	Extensive revision of structure and content. Added material from: 1) Maarten Sierhuis's PhD thesis; 2) Brahms Language Specification TM99-0008; 3) Brahms Installation readme.txt file; 4) Brahms 2001 TM01-0001 Project Plan. Added initial code.
Version 0.5 Draft 03/29/2001	Alessandro Acquisti 510/823-5008	Change	Extensive revision of structure and content. Revised chapter 4, sections 1-4
Version 0.6 Draft 04/01/2001	Alessandro Acquisti 510/823-5008	Change	Extensive revision of structure and content. Revised chapter 4, sections 5 <i>et seq.</i>
Version 0.7 Draft 04/10/2001	Alessandro Acquisti 510/823-5008	Add, Change	Added tutorial code and discussion to chapter 4.
Version 0.8 Draft 04/15/2001	Alessandro Acquisti 510/823-5008	Add, Change	Added links to Tutorial Files. Cleaned up various sections after Ron's comments
Version 0.9 Draft 04/17/2001	Alessandro Acquisti 510/823-5008	Add, Change	Added Bill, Maarten, Ron, and Charis' comments. General revision and updates.
Version 0.9.4 Request for Comments 05/12/2001	Alessandro Acquisti 510/823-5008	Add, Change	Added hyperlinks, index. Changed color and format for code sections, figures, tables. Edited 'Validation' chapter. Added new links to code.
Version 0.9.5 Request for Comments 05/26/2001	Alessandro Acquisti 510/823-5008	Change	General clean-up. Modified sections to reflect transition from SimAgent 1.0 to Agent Environment.
Version 0.9.6 Request for Comments 06/12/2001	Alessandro Acquisti 510/823-5008	Change	Added new screenshot figures and corrected some areas.
<i>Actions Taken</i> are: New = new document, Add/Delete/Change = a section or topic has been added, or deleted, or changed.			

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

Version	Contact	Action	
Version 0.9.7 Request for Comments 06/25/2001	Alessandro Acquisti 510/823-5008	Add	Added Ron's answers to language questions from Brahms Forums.
Version 0.9.8 Request for Comments 06/30/2001	Alessandro Acquisti 510/823-5008	Add, Change	Added reference section. Changed Agent Viewer description. Changed other parts in response to comments and suggestions from test-users.
Version 0.9.9 Request for Comments 07/11/2001	Maarten Sierhuis 510/604-4917	Add, Change	Made changes and added details in several sections.
Version 0.9.9.4 Request for Comments 07/11/2001	Alessandro Acquisti 510/823-5008	Add, Change	Added new material originated from feedback of new Brahms users. Extensively revised previous material. Adapted code. Altered the structure and order of Chapters 2 and 3.
Version 0.9.9.5 Request for Comments 01/01/2003	Alessandro Acquisti 510/823-5008	Add, Change	Corrected typos in various sections and inserted Composer description.
Version 1.0 07/01/2003	Alessandro Acquisti 510/823-5008	Add, Change	General clean up and revision.
Version 1.1 10/24/2005	Bin Zhang, Alessandro Acquisti	Add, Change	Updated to conform to new Brahms Composer. New language specifications *not* yet added.
Version 1.2 3/30/2011	Ron van Hoof	Change	Updated to conform to new Brahms Agent Environment. New language specifications *not* yet added.
<i>Actions Taken are: New = new document, Add/Delete/Change = a section or topic has been added, or deleted, or changed.</i>			

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

TABLE OF CONTENTS

1. INTRODUCTION	1-1
1.1 PURPOSE	1-1
1.2 INTENDED AUDIENCE.....	1-2
1.3 DOCUMENT SUMMARY.....	1-2
1.4 DOCUMENT CONVENTIONS	1-2
1.5 ACKNOWLEDGEMENTS	1-2
1.6 CONTACTS AND HELP.....	1-3
2. OVERVIEW OF BRAHMS AND THE ATM SCENARIO	2-4
2.1 WHAT IS BRAHMS? AN INTRODUCTION TO ITS THEORETICAL FOUNDATIONS AND CONCEPTS.....	2-4
2.2 ANATOMY OF A BRAHMS MODEL: THE ATM SCENARIO	2-6
2.3 OBJECT-ORIENTED PROGRAMMING AND BRAHMS.....	2-9
2.3.1 The Atm case in object-oriented programming	2-9
2.3.2 The Atm case in Brahms	2-10
3. INSTALLATION AND COMPONENTS	3-13
3.1 BRAHMS OVERVIEW	3-13
3.2 INSTALLATION	3-13
3.2.1 Installing Brahms Agent Environment	3-13
3.2.2 Installing MySQL.....	3-14
3.2.3 Installing the License File	3-14
3.2.4 Choosing the Brahms Model Directory.....	3-15
3.2.5 The Atm Files	3-15
3.2.6 To Summarize: What you Will Need	3-16
3.3 DESCRIPTION OF COMPONENTS	3-16
3.3.1 Introduction: The Life of a Brahms Simulation	3-16
3.3.2 The Brahms Composer: Opening, Creating, and Building a Model	3-17
3.3.3 The Brahms Composer and the Virtual Machine: running a Model.....	3-24
3.3.4 Brahms Agent Viewer.....	3-26
3.4 SUMMARY OF STEPS	3-30
3.5 A NOTE ON DEBUGGING... ..	3-31
3.6 KNOWN BUGS IN BRAHMS AGENT ENVIRONMENT	3-31
3.7 CONTACTING THE BRAHMS PROJECT TEAM FOR TECHNICAL SUPPORT.....	3-31
3.8 OTHER IMPORTANT DOCUMENTS	3-32
3.9 LATEST CHANGES.....	3-32
3.10 DOCUMENT INDEX	3-33
4. ATM SCENARIO	4-34
4.1 STRUCTURE OF THE SCENARIO.....	4-35
4.2 EXPECTATIONS AND GOALS.....	4-35
4.3 LESSON I: GETTING STARTED	4-37
4.3.1 Introduction.....	4-37
4.3.2 Task.....	4-37
4.3.3 Description: compilation unit.....	4-37
4.3.4 Tutorial.....	4-38

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

4.3.5	Syntax.....	4-40
4.4	LESSON II: GEOGRAPHY.....	4-41
4.4.1	Introduction.....	4-41
4.4.2	Task.....	4-41
4.4.3	Description.....	4-41
4.4.4	Syntax.....	4-42
4.4.5	Tutorial.....	4-43
4.5	LESSON III: GROUPS, AGENTS AND ATTRIBUTES.....	4-49
4.5.1	Introduction.....	4-49
4.5.2	Task.....	4-49
4.5.3	Description.....	4-49
4.5.4	Syntax.....	4-53
4.5.5	Tutorial.....	4-54
4.6	LESSON IV: FACTS AND BELIEFS.....	4-58
4.6.1	Introduction.....	4-58
4.6.2	Task.....	4-58
4.6.3	Description.....	4-58
4.6.4	Syntax.....	4-60
4.6.5	Tutorial.....	4-60
4.7	LESSON V: WORKFRAMES AND PRIMITIVE ACTIVITIES.....	4-64
4.7.1	Introduction.....	4-64
4.7.2	Task.....	4-64
4.7.3	Description.....	4-64
4.7.4	Syntax.....	4-73
4.7.5	Tutorial.....	4-74
4.8	LESSON VI: CLASSES, OBJECTS AND RELATIONS.....	4-88
4.8.1	Introduction.....	4-88
4.8.2	Task.....	4-88
4.8.3	Description.....	4-88
4.8.4	Syntax.....	4-91
4.8.5	Tutorial.....	4-91
4.9	LESSON VII: THOUGHTFRAMES AND OTHER ACTIVITIES.....	4-100
4.9.1	Introduction.....	4-100
4.9.2	Task.....	4-100
4.9.3	Description.....	4-100
4.9.4	Syntax.....	4-102
4.9.5	Tutorial.....	4-103
4.10	LESSON VIII: VARIABLES.....	4-108
4.10.1	Introduction.....	4-108
4.10.2	Task.....	4-108
4.10.3	Description.....	4-108
4.10.4	Syntax.....	4-111
4.10.5	Tutorial.....	4-111
4.11	LESSON IX: COMPOSITE ACTIVITIES.....	4-117
4.11.1	Introduction.....	4-117
4.11.2	Task.....	4-117
4.11.3	Description.....	4-117
4.11.4	Syntax.....	4-119
4.11.5	Tutorial.....	4-120
4.12	LESSON X: MULTI-AGENT, RANDOMNESS, AND COMPLEX INTERACTIONS.....	4-125
4.12.1	Introduction.....	4-125

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

4.12.2	Task.....	4-125
4.12.3	Description.....	4-125
4.12.4	Syntax.....	4-133
4.12.5	Tutorial.....	4-134
4.13	LESSON XI: DETECTABLES, PRIORITIES AND THE COMPLETE SCENARIO	4-136
4.13.1	Introduction.....	4-136
4.13.2	Task.....	4-136
4.13.3	Description.....	4-137
4.13.4	Syntax.....	4-140
4.13.5	Tutorial.....	4-141
4.14	CONCLUDING ISSUES.....	4-145
4.14.1	How to build your next model	4-145
4.14.2	Debugging tips.....	4-146
4.14.3	Validation.....	4-147
4.14.4	Further issues and exercises	4-147
5.	VALIDATION	5-149
5.1	MODELING WORK PRACTICE	5-149
5.2	COMPUTATIONAL MODELS IN SIMULATION	5-150
5.3	TYPES OF MODELING SYSTEMS.....	5-151
5.4	VERIFICATION AND VALIDATION.....	5-153
5.4.1	The purpose of verification and validation	5-153
5.4.2	The verification and validation process	5-154
5.4.3	Data validation.....	5-155
5.4.4	Conceptual model validation	5-155
5.4.5	Experimentation validation	5-158
6.	INDEX	6-159
7.	REFERENCES AND OTHER LINKS	7-161

FIGURES

FIGURE 1 - THE COMPOSER: THE GRAPHIC INTERFACE	3-18
FIGURE 2 - THE COMPOSER: THE INTERNAL TEXT EDITOR	3-19
FIGURE 3 - OPENING THE COMPOSER	3-20
FIGURE 4 - IMPORTING THE ATM COMPLETE SCENARIO IN THE COMPOSER	3-21
FIGURE 5. SELECT A HISTORY FILE TO PARSE IN THE AGENT VIEWER.....	3-27
FIGURE 6. CREATING HISTORY DATABASE.....	3-27
FIGURE 7. PARSING HISTORY FILE INTO HISTORY DATABASE	3-27
FIGURE 8. AGENT VIEWER APPLICATION – AS OF JULY 2004.....	3-28
FIGURE 9. ATMMODEL.B IN NOTEPAD.....	4-39
FIGURE 10 - CREATING GEOGRAPHY AREAS IN THE COMPOSER.....	4-43
FIGURE 11. BELIEFS AND FACTS VENN DIAGRAM.....	4-60
FIGURE 12. THE SIMULATION ENGINE (VIRTUAL MACHINE)	4-78
FIGURE 13. THE AGENT VIEWER AND THE ATM SCENARIO	4-80
FIGURE 14. THE FIRST ACTIVITY IN THE AGENT VIEWER	4-81
FIGURE 15. THE EXPLANATION FACILITY IN THE AGENT VIEWER.....	4-82
FIGURE 16. MORE COMPLEX ACTIVITIES	4-86
FIGURE 17. THE AGENT VIEWER AND OTHER ACTIVITIES	4-99
FIGURE 18. WORKFRAME-ACTIVITY HIERARCHY	4-118
FIGURE 19. STATE-TRANSITION DIAGRAM FOR FRAME INSTANTIATIONS	4-129
FIGURE 20. MULTI-TASKING IN BRAHMS.....	4-133
FIGURE 21 - A SCREENSHOT FROM THE COMPLETE ATM SCENARIO.....	4-144
FIGURE 22 - A ZOOM IN OF THE COMPLETE ATM SCENARIO.....	4-144
FIGURE 23. SIMULATION MODEL VERIFICATION AND VALIDATION IN THE MODELING PROCESS (BORROWED FROM (ROBINSON, 1999)).....	5-155
FIGURE 24. AN EXAMPLE OF CONCEPTUAL MODEL FOR THE SIMULATION OF AN APOLLO MISSION	5-156
FIGURE 25. BRAHMS COMPILER-DEBUG CYCLE	5-157
FIGURE 26. BRAHMS MODEL DEVELOPMENT CYCLE	5-157
FIGURE 27 BLACK-BOX VALIDATION: COMPARISON WITH THE REAL SYSTEM (FROM (ROBINSON, 1994)).....	5-158

1. INTRODUCTION¹

1.1 PURPOSE

This tutorial will introduce you to the “Brahms” language and development environment and their use for agent-oriented modeling. Brahms can be used to model complex work practice scenarios. This tutorial will present the main aspects of the language and discuss the usage of the modeling constructs available.

Because Brahms is an agent-oriented language, it requires a different way of thinking than object-oriented and procedural languages. In many books on object-oriented and procedural languages the Atm (automatic teller machine) scenario is used as an introduction to the respective language. This tutorial will use the Atm example in order to showcase many of Brahms modeling constructs and to show how they differ from traditional object-oriented or procedural languages.

This tutorial will introduce many language constructs and specifications and describe their application to the Atm scenario. The discussion of a Brahms implementation of the Atm scenario will be the core of this tutorial. You, the reader, will be challenged to complete exercises that consist in writing pieces of Brahms code to model increasingly complex aspects of the scenario. You will also be given working examples and pieces of code with which to compare your solutions. This tutorial, however, does not replace the Brahms Language Specification document (available online at http://agentisolutions.com/documentation/language/ls_title.htm), which is a comprehensive guide to the modeling capabilities of Brahms. The Brahms Language Specification document fully defines the formal syntax and related semantics of all the modeling concepts. In addition, while this tutorial is self-contained and does also cover elements of syntax and semantics, it should be regarded simply as an *introduction* to Brahms. Agent-based modeling of social phenomena is more an art than a science – and this is particularly true of Brahms, whose modeling richness can be used in several different ways. This tutorial will teach you by example how to use the language concepts, so that by the end of this tutorial you will be able to build simple to reasonably complex Brahms models. However, for a deeper mastering of the language, this document should be used in coordination with the Brahms Language Specification document.

Finally, the Brahms language is still in development. It is often updated with new functionalities, which are described in the Language Specification document available online. This tutorial may *not* include the latest developments of the language. The current **version of this tutorial, 1.2**, reflects the status of the language as of **July 2003**.

¹ Sources of this chapter: Brahms 2001 TM01-00001 Project Plan; communications with Brahms development team.

1.2 INTENDED AUDIENCE

This document is to be used by future Brahms modelers as a guide in their development of Brahms models. Some level of previous programming experience is expected. Knowledge of object-oriented languages and rule-based languages is preferable, but not essential.

1.3 DOCUMENT SUMMARY

This tutorial is a self-contained introduction to the Brahms language. It starts by presenting an overview of the Brahms programming language and a bird's eye view of the Atm scenario. It then shows the reader how to install the components of the Brahms development system. The Atm scenario then begins, with simple concepts and examples that progressively build up in complexity and difficulty. In each scenario subsection the reader will be introduced to a set of new language concepts and asked to use them in increasingly realistic Brahms models. Experienced programmers might move quickly through the initial sections of the Atm scenario (Chapter 4, Sections 3 – 8) and spend more time on the more complex sections (Sections 9 –11). While the Atm scenario will be especially useful to people familiar with procedural or object-oriented languages, it will also be understandable by those with no prior knowledge of procedural and object-oriented languages.

1.4 DOCUMENT CONVENTIONS

This font is used for text (descriptions, explanations, etc.)

`This font is used for code, as well as folder names, file names, etc.`

1.5 ACKNOWLEDGEMENTS

Many thanks to Boris Brodsky, Charis Kaskiris, Laleh Haghshenass, Amber Lo, and Chin Seah, Bin Zhang, for beta testing and useful feedback.

1.6 CONTACTS AND HELP

Updated versions of this document are available from the AgentISolutions website: www.agentisolutions.com. Questions about this tutorial and typos should be notified to Alessandro Acquisti (acquisti@agentisolutions.com).

Printed on:

3/31/11 3:08 PM

This is an uncontrolled copy when printed.

Refer to the NX Brahms location for the latest version.

2. OVERVIEW OF BRAHMS AND THE ATM SCENARIO

2.1 WHAT IS BRAHMS? AN INTRODUCTION TO ITS THEORETICAL FOUNDATIONS AND CONCEPTS

Brahms models may be thought of as statements in a new formal language developed for describing work practice. The online language specifications (at http://agentisolutions.com/documentation/language/ls_title.htm) show the conventional notation and constructs used to express the syntax for the modeling language. 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.

Brahms can model and simulate work practices. Brahms models are written in the agent-oriented language that will be described in this document. The run-time component - the simulation engine - can execute a Brahms model, also referred to as a simulation run. A Brahms model can be used to simulate human-machine systems for what-if experiments, for training, for “user models,” or for driving intelligent assistants and robots. Brahms is different from task and functional analysis. A traditional task or functional analysis of work leaves out the logistics, especially how environmental conditions come to be detected and how problems are resolved. Without consideration of these factors, it is not possible to accurately model how work and information actually flows, or to properly design software agents that help automate human tasks or interact with people as their collaborators. What is wanted is instead a model that includes aspects of reasoning found in an information-processing model, plus aspects of geography, agent movement, and physical changes to the environment found in a multi-agent simulation. A model of “work practice” focuses on informal, circumstantial, and located behaviors by which *synchronization* occurs, such that the task contributions of humans and machines flow together to accomplish goals.

Brahms makes this kind of models possible. Brahms relates knowledge-based models of cognition (e.g., task models) with discrete simulations and the behavior-based subsumption architecture. Brahms is centered on the concept of “agents.” Agents’ behaviors are organized into activities, inherited from groups to which agents belong. Most importantly, activities locate behaviors of people and their tools in time and space, such that resource availability and informal human participation can be taken into account. A model of activities doesn’t necessarily describe the intricate details of reasoning or calculation, but instead captures aspects of the social-physical context in which reasoning occurs. Thus Brahms differs from other multi-agent systems by incorporating chronological activities of multiple agents, conversations, as well as descriptions of how information is represented, transformed, reinterpreted in various physical modalities.

The Brahms language is structured around the following concepts:

- Agents and Groups
- Objects and Classes
- Beliefs and Facts
- Workframes
- Activities
- Thoughtframes
- Geography

which can be related one to the other in the following way:

Groups contain
agents who are located and have
beliefs that lead them to engage in
activities that are specified by
workframes that consist of
preconditions of beliefs that lead to
actions, consisting of
communication actions
movement actions
primitive actions
other composite activities
consequences of new beliefs and world facts
thoughtframes that consist of
preconditions and
consequences

The Atm scenario will drive you through all of these concepts. It will first present them – one by one - and then it will ask you to use them in increasingly realistic Brahms representations of a scenario where students get money from Atms and spend it at various restaurants.

The goals of the rest of this chapter are to offer a bird's eye view of the language, introduce the Atm scenario that will be used in the tutorial, and highlight differences between the Brahms modeling philosophy and that of object-oriented languages.²

2.2 ANATOMY OF A BRAHMS MODEL: THE ATM SCENARIO

A Brahms file is a file written according to the rules of the Brahms language and identified with the `.b` extension (the Brahms programming environment, called the "Composer," uses the `.bmd` extension to refer to a complete Brahms model written with the Composer. The Composer can be downloaded from the AgentISolutions website). Most of your Brahms models (including the Atm case that you will build through this tutorial) will consist of more than one Brahms file. Unlike most other languages like C, Java or Pascal, Brahms does not have a `main` method that serves as the starting point for a Brahms simulation. While Brahms does not prevent you from placing more than one concept in one Brahms file, it is recommended that you create one `.b` file for each "concept" that you specify for your model. A concept can be a group, agent, class, object, areadef, area, path, conceptual class or conceptual object - but we will go back to this in detail in the next sections. It is also recommended that you always create one Brahms file (`.b`) for your model that will import all the other `.b` files that are to be part of your model. Also this aspect of the language will be explained in the next sections.

The Atm scenario discussed in the following pages will be used to introduce you to the various components of the language and the main differences between object-oriented and agent-oriented programming. The goal of the scenario exercises is to model the following system:

Model a day in the life of a college student. Notoriously, college students spend most of their time studying, but get hungrier as the time goes by. When their hunger reaches a certain threshold, students have to move to one of the restaurants around their location. Students choose restaurants according to how much money they are carrying: the richer they are, the more expensive a restaurant they will choose. If a student does not have enough money even for the cheapest restaurant, she will decide to pass first by an Atm of the bank where she has an account.

When she arrives at the Atm, the student inserts her bankcard and tries to remember the PIN associated to her account. The Atm allows its user 3 attempts to digit the correct PIN, before refusing the card altogether. The Atm communicates with the central bank computer to verify the correctness of the information provided by the user. If the bank computer informs to the Atm that the PIN is correct and that the user has enough balance in her account, the Atm will dispense the cash and will print an invoice with the account number and the remaining balance.

² For additional information about the theoretical foundations of Brahms, cfr. 3.8.

Students need to have enough balance in their accounts to take out cash: if they attempt to take out more money than they have, the bank computer will notify the students (through the Atm) of the remaining balance of the account. The student will then need to modify her request accordingly, and only take out the remaining dollars.

Don't worry – you will not have to model all of this right now! Chapter 4 will drive you through this scenario step by step, progressively adding components and activities and increasing realism and complexity. Still, to get you started and give you a sense for the language that you will be discovering, you will be now given an example of code coming from the Atm case itself. Clearly, you will not yet be able to "read" this code. Still, it may be useful to you to browse through it, trying to recognize some of keywords that we have described above (such as "agent," which are the subjects of "activities," which are triggered by "workframes," etc.), to get familiarized with the concept and components of a Brahms file. This file describes (some of) the activities of the "Student" agent:

```
package gov.nasa.arc.brahms.atm;
group Student {
  attributes:
    public boolean male;
    public double howHungry;
    public int perceivedtime;
    public Diner chosenDiner;
    public boolean hasTakenCash;

  relations:
    public Account hasAccount;
    public Cash hasCash;
    public BankCard hasBankCard;

  initial_beliefs:
    (current.hasTakenCash = false);
    (Boa_Atm.location = BankOfAmericaUniversityBranch);
    (WF_Atm.location = WellsFargoUniversityBranch);

  activities:
    move MoveToLocation(Building loc) {
      priority: 1;
      location: loc;
    }
    primitive_activity FeelHungry() {
      priority: 1;
    }
}
```

```
    }
workframes:
  workframe wf_MoveToLocationForCash {
    repeat: true;
    variables:
      forone(Cash) cs;
        forone(Atm) at;
        forone(Bank) bk;
        forone(Building) bd;
    when( knownval(current hasCash cs) and
          knownval(at.location = bd ))
    do {
      MoveToLocation(bd);
      conclude((current.readyToLeaveAtm = false),
bc:100);
    }
  }
thoughtframes:
  thoughtframe tf_ChooseBlakes {
    repeat: true;
    variables:
      forone(Cash) cs;

    when( knownval(current hasCash cs) and
          knownval(cs.amount > 15.00) and
          knownval(current.checkedDiner = false) and
          knownval(Campanile_Clock.time < 20))

    do {
      conclude((current.chosenDiner = Blakes_Diner),
bc:100);
    }
  }
} // Student
```

2.3 OBJECT-ORIENTED PROGRAMMING AND BRAHMS

The code above is taken from the Atm model that you will build during this tutorial. The Atm scenario is a classical case in object-oriented programming (OO). Comparing its implementation in OO languages and in an agent-based language such as Brahms will be particularly instructive.

2.3.1 THE ATM CASE IN OBJECT-ORIENTED PROGRAMMING

2.3.1.1 THE SCENARIO IN AN OBJECT-ORIENTED FRAMEWORK

Consider the Atm scenario described by Rumbaugh *et al.* in *Object-Oriented Modeling and Design*, 1991 (ch. 2-6). The goal of the OO programmer is to “design the software to support a computerized banking network including both human cashers and automatic teller machines (Atms) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank’s computers. Human cashiers enter account and transaction data. Automatic teller machines communicate with a central computer which clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts.” (*op. cit.*, p. 151).

Therefore, the Atm scenario can be represented as a set of objects with their attributes and relations: a consortium, consisting of banks, which hold accounts of customers; customers have cash cards, through which they access their accounts during transactions at the Atm, which accesses the central computer to communicate with the Bank (*cf. op. cit.*, ch. 8).

2.3.1.2 THE “MODELS” IN OBJECT-ORIENTED PROGRAMMING

The object modeling technique that Rumbaugh *et al.* propose to code the Atm scenario uses three kinds of “model:” the object model, that describes the objects in the system and their relationship; the dynamic model, that describes the interactions among objects in the system; and the functional model, that describes the data transformations of the system. Note that the term “models” here refer to descriptions of different aspects of the system, and clearly cross-linked. Through them, a modeler can design an Atm system as a system of objects with attribute and relations that exchange information within themselves.

2.3.2 THE ATM CASE IN BRAHMS

2.3.2.1 THE SCENARIO IN BRAHMS

At the core of Brahms there is the concept of “agent.” An agent represents an interactive system, a “subject” with a certain “behavior” who is 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, and agents engage in activities depending on “facts” of the world as well as the “beliefs” they have about those facts. Mapping the Atm scenario from objected oriented programming into Brahms, therefore, means that the modeler must adopt a “holistic” approach to modeling, trying to explain and simulate individual behavior through actions and decisions, and selecting what to keep inside the picture and what to take out. An exemplar Brahms embodiment of the Atm scenario discussed above could be the following (note how the abstract scenario has been made more specific; note also that relevant Brahms concepts and keywords are indicated within parentheses – their meanings will be discussed in the next sections):

Alex (agent), a student (group) at Berkeley (location), has cash (object) and an account (object) at the Bank of America (object, instance of the class Bank) that he can access by using his cashcard (object) at an Atm machine (object with location). Alex studies (activity) most of the time (workframe), but as the time goes by he also feels (belief) the urge (workframe) to eat (activity). There are some restaurants in Berkeley, with different menus and prices (attribute). When he is hungry (attribute), Alex checks (thoughtframe) how much money he is carrying (fact). Depending on his financial situation, he can then decide to (workframe) move (activity) to one of restaurants and eat, or instead go first to the Atm to get some cash (workframe, activity). At the Atm, Alex will insert his cashcard into the Atm machine, which will read the card and ask for its pin. The Atm then will communicate with the Bank, verify the card and the provided information, and complete or abort the transaction. To make things more interesting, there is another student, Kim, who does the same things Alex does but likes different restaurants, has a different time schedule, and has opened her account at a different Bank from the one Alex is using..

As it might be already clear from this short example (and as it will certainly get clearer the more you will progress into your tutorial), there is a deep conceptual difference between the way the Atm scenario can be dealt with in Brahms and the object-oriented approach. To write a proper Brahms model, you will have to think about the *why* and *how* of the various actions agents and objects perform. This is a more complete and more holistic approach. It is a human centered computing (HCC) view as opposed to the software engineering view. The latter might be content with focusing on use cases. The HCC view instead studies why and how the human is using the machine, before designing the Atm machine itself. This also translates in language differences between Brahms and other object-oriented languages: Brahms is an agent-oriented language with elements of rule-based languages, where “activities” are not the same things as “methods,” and “workframes” are not the same as “if..then” constructs of imperative languages. It is important to keep this in mind while modeling in Brahms, in order to avoid errors and misunderstandings.

2.3.2.2 THE “MODELS” IN BRAHMS

Let's go back to the concept “models” (see 2.3.1.2), by which we refer to “views” or “descriptions” of different aspects of a scenario we want to model and simulate. The Atm scenario can be decomposed into various Brahms “models”. In fact, in Brahms more models can be used than the 3 ones discussed in Rumbaugh *et al.* Each “model” is a view of a specific aspect that is important in a Brahms simulation: the agents, the activities, the objects, etc.

What follows is a list of ‘conceptual’ models through which you can *interpret* and *think through* your Brahms simulations. Note that they are not explicit pieces of code: you can rather see them similarly to the “views” in the UML approach. Similarly to UML views, when you will know more about the Brahms language, you will see that thinking your project first in terms of these Brahms models or ‘views’ will improve the speed, efficiency and precision of your Brahms coding.

1. Agent model; represents the groups, agents and their relationship. *For example, Alex and Kim are both students, that is, members of the group Student.*
2. Activity model; represents the activities that can be performed by agents and objects. *Activity like going to the bank to get money, or going to eat, with their decomposition into smaller units.*
3. Communication model; represents the communication between agents and objects. *The communications that take place, for example, between a student and the Atm machine, or the Atm machine and the bank.*
4. Timing model; represents the constraints and relationships between activities - if any exist. *The staging of the simulations: for example, before going to the restaurant, an agent will check if he has enough money, and in case might go to the bank to get more...*
5. Knowledge model; represents the knowledge (initial beliefs and thoughtframes) of agents and objects. *Agents have beliefs that describe their knowledge of the world. Alex might or might not know where the Atm machine for his bank is located. He needs that information in order to get cash out of the bank. Agents can also deduce new beliefs, based on inference rules, such as: “If I don't have enough money to pay for my lunch, then I first have to go to the bank to get money.”*
6. Object model; represents the (conceptual) classes and objects in the world, used as resources by agents or used to track information flow. *Objects like the Atm machine, the Bank Card, the Cash, etc.*

7. Geography model; represents the geographical environment in which agents and objects perform their activities, specifying areas and possible travel paths. *The geography of Berkeley, with details about where the restaurants are, where the Atm machines are, etc.*

Your first exercise in the tutorial will be to write down a draft of how the various concepts which are part of the Atm scenario should fit into these models. Note: you are not expected to be writing code! Quite the opposite, you can choose the format of your choice (a list of items, drawings with boxes and lines, or tools like Visio, Mifflin,³ etc.) to represent conceptually how you see the scenario's components and their interactions. Ask yourself questions like: what concepts should I put in the agent model? What concepts are important in the knowledge model, because they refer to – say - the knowledge agents have about their environment? Note that this is an exercise *less* about the Brahms language specification than about its underlying philosophy. In other words, you do not need to know the language to complete this exercise. Rather, you must start thinking in terms of “work practice,” activities situated in specific environments, as well as they “why” and “how” of things.

You might start with the information provided above. Currently, without knowing much of Brahms syntax and structure, it is likely that you will not be able to neatly fit the scenario concepts described above in the proper model/view. Don't worry - you will see that as you progress in the tutorial, you will be able to come back to these models and refine them. Ok: you should give it your initial try now!

³ See <http://www.compendium.org>.

3. INSTALLATION AND COMPONENTS⁴

3.1 BRAHMS OVERVIEW

Brahms models are written in an agent-oriented language that has a well-defined syntax and semantics. The Brahms language is a “parsed” language: you write the code and then the parser generates an internal object representation for the “run-time” component. Using this language, a Brahms modeler can create *Brahms models*. The run-time component - the *simulation engine* - can execute a Brahms model, also referred to as a *simulation*. This chapter will teach you how to install and use the various components that form the Brahms system and allow you to write Brahms model and then run them as Brahms simulations.

3.2 INSTALLATION

Brahms development environment is bundled into an application called the “Brahms Agent Environment.”⁵ The environment also requires MySQL and a license file to function properly.

3.2.1 INSTALLING BRAHMS AGENT ENVIRONMENT

First, go to <http://www.agentisolutions.com/download/download.htm> and download the installation file for Brahms Agent Environment for the operating system of your choice (operating systems currently supported are: Windows XP/Vista/Win7, Linux, and Mac OS X). The Brahms Agent Environment installation package includes several components:

- The Composer, which is an IDE (Integrated Development Environment) that allows you to build Brahms models.
- The Compiler, which is an application which parse the code you have created with the Composer into code that is readable by the Virtual Machine.
- The Virtual Machine (or Simulation Engine), that literally reads and runs the code parsed by the Compiler and produces a “simulation run” of your model.

⁴ Sources: Brahms Installation readme.txt file; communications with Brahms development team.

⁵ There exists also a version called “ProfessionalAgent.” In this tutorial will be focus on the features available in the PersonalAgent application. For more information, visit <http://www.agentisolutions.com>.

- The Agent Viewer, that allows you to visualize and study the simulation run you have just created (integrated in the Composer).

After you download the Agent Environment, use the file [setup.exe](#) to launch the installation program that will guide you through the steps to install “Agent Environment.” Agent Environment is the core of the Brahms language: you will use its components to write, compile, and run your Brahms models. For the installation location we recommend C:\Brahms\AgentEnvironment. Installation of the Brahms Agent Environment in C:\Program Files will result in issue on Windows Vista and Windows 7 due to its User Account Control not permitting an application to write data to sub folders of Program Files.

The installation program will install one executable and some scripts in a [bin](#) directory, including: one for the Brahms Composer ([Composer.exe](#)), one for the Brahms Compiler ([bc.bat](#)), and one for the Simulation Engine (Virtual Machine: [bvm.bat](#)).

The [setup.exe](#) program will also install the Brahms Agent Viewer that you will be able to access through the Composer. The Agent Viewer will give you a graphical representation of the Brahms models that you have successfully compiled and run with Agent Environment.

3.2.2 INSTALLING MYSQL

Next, download MySQL 5.1. In order to actually see your Brahms simulations’ results, you need MySQL 5.1, which is *not* distributed with Brahms but that can be downloaded for free from <http://dev.mysql.com/downloads/>. Save and install the version of MySQL 5.1 that is compatible with your operating systems. (Please read the [AgentViewer_Readme.html](#) file in the Brahms directory - which is [c:\program files\brahms\AgentEnvironment](#) by default - for detailed MySQL 5.1 installation instructions.)

3.2.3 INSTALLING THE LICENSE FILE

Finally, go back to the AgentISolutions website and register in order to obtain a Brahms license file. Go to the page: <http://www.agentisolutions.com/download/registration.htm> and provide your email address. The license file is needed to run the Brahms simulation engine. After you register, you will receive your license file by email. Next, you should copy it to the new [Brahms\AgentEnvironment\](#) directory that the Agent Environment installation application has created.

3.2.4 CHOOSING THE BRAHMS MODEL DIRECTORY

By default, `setup.exe` will install all Agent Environment files in a folder `Brahms\AgentEnvironment\` in the `c:\Program Files` directory. In this same directory the Composer will save by default your Brahms models. We recommend installation in `C:\Brahms\AgentEnvironment\` especially on Vista and Windows 7 due to limitations imposed by the User Account Control not permitting applications to write data inside of sub folders of Program Files.

It might be useful to create an alternative folder for your projects and their files. Pros: you might want to keep your models (the “data”) in a different location (e.g., a different drive) than your application. Cons: you will have to do some “personalization” of a couple of components (not to worry: the steps are discussed below). In the rest of this tutorial we will follow the “hard” path and create our own model folder. This will be a nice way to start learning how Brahms works.

3.2.5 THE ATM FILES

Together with this Tutorial and the installation file for the Agent Environment, you should also download a zipped set of folders containing the Atm model code for different stages of the scenario evolution. If you have not downloaded them yet, do it now! Currently you can find the Atm model files here:

http://www.agentisolutions.com/documentation/tutorial/Brahms_Tutorial_Files.zip

(there will be a zipped file containing 4 folders, referring to different stages in the evolution of the tutorial). After you download the zipped files, we suggest that you unzip them to the following folder (that you need to create):

```
c:\Brahms\Projects\
```

The first folder – `Brahms` – can be used as a location for all of your Brahms working (i.e., non-application) files. The subfolder `Projects` will be used to contain the actual Brahms files that are part of your future Brahms models. The unzipping process will create a folder `c:\Brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\` that will contain the files belonging to various versions of the Atm model – the ones that we provide for you, and the ones that you will develop by yourself. Hence, from now on and for the rest of this tutorial, we suggest that you will save all *your* Brahms files for your Atm project inside the folder:

```
c:\Brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\
```

You will use the files we provide to you to study the application of the language constructs, to compare your code with ours, and for help when you will have to write your own Brahms code to model various aspects of the Atm scenario.

You can also use those files to start seeing how a Brahms model works. The folder called 'Final' contains the complete version of the scenario: that is, what your model should look like by the end of this tutorial. The other folders contain the scenario as modeled up till different sections of this tutorial, that is, they are the "solutions" to earlier sections of the tutorial (the name of the folder refers to which chapter's stage of the scenario the code describes). These folders offer you a way to study and compare the evolution of the tutorial from the simple activities of the first Lessons to the complex interactions of the last ones. You may open these files right now and see their content. You will find:

- `.b` files: they are the Brahms model files.
- `.bcc` files: they are the files produced by the "Brahms Compiler."
- `Atm.....txt` files: they are files containing the history of the events in the simulation, and they are produced by the "Brahms Virtual Machine", or "Simulation Engine."
- `EventInformation.txt` file: this file contains a human readable output of the events in the simulation, and they are produced by the "Brahms Virtual Machine", or "Simulation Engine."

3.2.6 To SUMMARIZE: WHAT YOU WILL NEED

To recap: to use Brahms and this Tutorial, you will need to:

- Download and install the Agent Environment (see above)
- Download and install MySQL 5.1 (see above)
- Obtain and install the Brahms license file (see above).
- Download and unzip the Tutorial files (see above)

Ok? Let's move on then!

3.3 DESCRIPTION OF COMPONENTS

3.3.1 INTRODUCTION: THE LIFE OF A BRAHMS SIMULATION

In this section we will consider the different components of the Brahms system. The "big picture" to keep in mind is the following:

1. A Brahms simulation starts with somebody – you! – writing Brahms code. You will use the Brahms Composer for this scope. In the Composer you can write code by using the internal text editor, or through the graphic interface that allows you to create concepts and modify parameters by moving your mouse and dragging/dropping items on the Composer windows. The final output of this stage will be a (set of) Brahms files, i.e. lines of code written according to the Brahms language rules and saved with the `.b` extension (the Composer also saves your entire model as a single entity in the `.bmd` format. More on this below)
2. Then, while still inside the Composer, you can “build” your `.b` files. Building your model means that you use the internal parser to produce new files based on the `.b` files you have written. The new files are `.bcc` files that can be read by the Brahms Simulation Engine. The new files are produced if no compilation errors are encountered.
3. The next step is to “run” your model (now contained in the `.bcc` files) into a simulation: you can do this from the Composer environment (which in turn will use a component called Brahms Simulation Engine, aka Virtual Machine). If all goes fine, the result will be an “event” file (saved with the `.txt` extension) that contains all the “history” of your simulation.
4. Last step: seeing your simulation. Again, from within the Composer, you can “view” with the Agent Viewer - you will have to create a new Brahms database based on the `.txt` event file and then open it through a graphical interface provided inside the Composer.

We will now discuss in details these various components by trying to run the Atm model files available online..

We will assume that you have already completed the installation steps described above: you installed the Agent Environment, you installed MySQL, you obtained your License File, and you downloaded the Tutorial files.

In the rest of this section we will test the installation and the behavior of your Brahms system by making use to the Tutorial files.

3.3.2 THE BRAHMS COMPOSER: OPENING, CREATING, AND BUILDING A MODEL

The Brahms Composer allows you to create Brahms models through a powerful and intuitive graphic interface. The complete Composer’s manual is available from the AgentiSolutions website (<http://www.agentisolutions.com/>) and is a great companion to this tutorial. In this section we only provide a quick introduction to its usage and capabilities, but we refer the reader to the Composer manual for a more accurate description.

In theory, any text editor would be sufficient to create Brahms models. You could just insert code for each “concept” in your model (such as groups, agents, locations, objects, etc.) into a different file, and give each of these files the extension `.b` (see Section 4.3). However, with the Brahms Composer you can create much more easily and quickly Brahms concepts and edit them. Editing can be done through the Composer’s graphic interface (Figure 1), or through the internal text editor (Figure 2). The Composer automatically creates `.b` files for you, as well as `.bmd` files. The former are the actual Brahms files that you create, edit, and parse in order to obtain a simulation run. Typically, a Brahms model – such as the Atm scenario – is composed of several agents, objects, etc., and therefore of several such files. The latter are files used only by the Composer to manipulate your model as a whole – that is, there is only one such file for each model of yours.

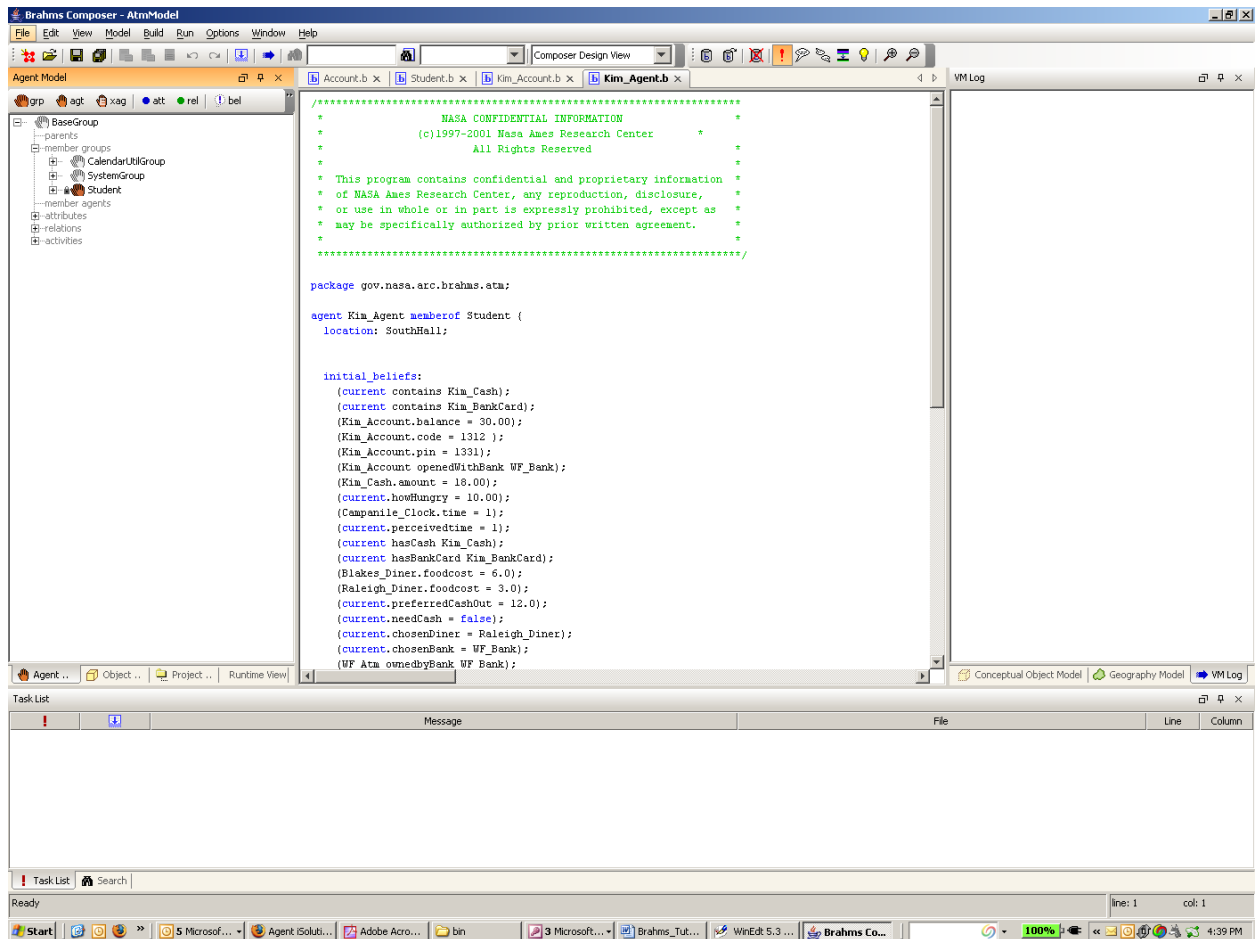


Figure 1 - The Composer: The graphic interface

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

When you start the Composer, you have the option to start a new model, open an existing one, or import the files of an existing model (Figure 3). The first option will be discussed in Section 4.3. The difference between the other two options is that “opening” refers to accessing a `.bmd` file previously created by the Compose, while “importing” refers to importing into the Composer a set of Brahms `.b` files (composed, as discussed above, possibly with a simple text editor), starting with the `.b` file in your model that has an “import” statement for all the other files (“import” statements are discussed in Section 4.3).

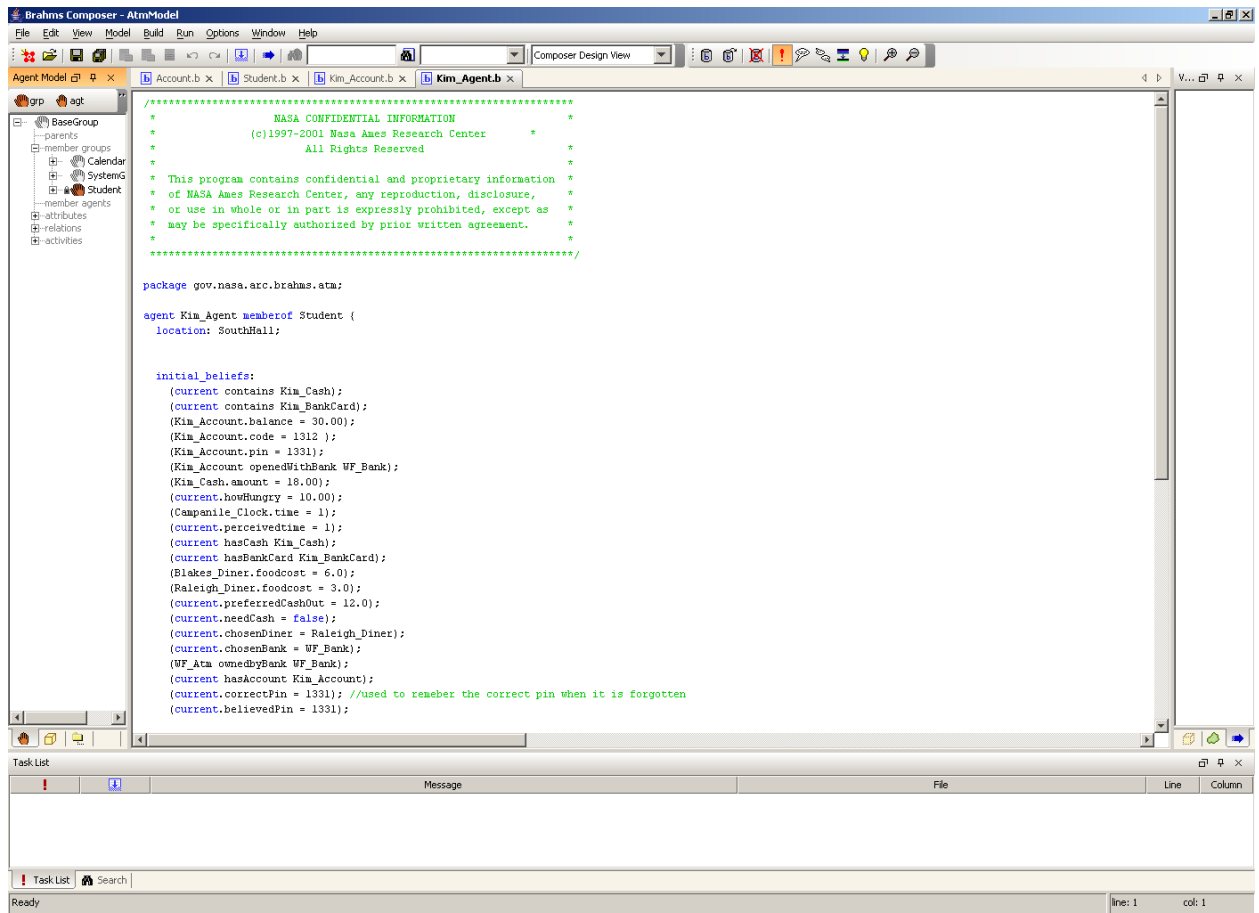


Figure 2 - The Composer: The internal text editor

To learn about the Composer and to test the installation of your files, we will try to use both latter options (“import,” and “open”) with the Atm scenario files that you have downloaded. Note, however, that in the rest of this tutorial, we will focus more on understanding how the language works and can be used, and less on how to use the Composer (although examples and references will be made throughout the text).

Printed on: This is an uncontrolled copy when printed.

3/31/11 3:08 PM Refer to the NX Brahms location for the latest version.

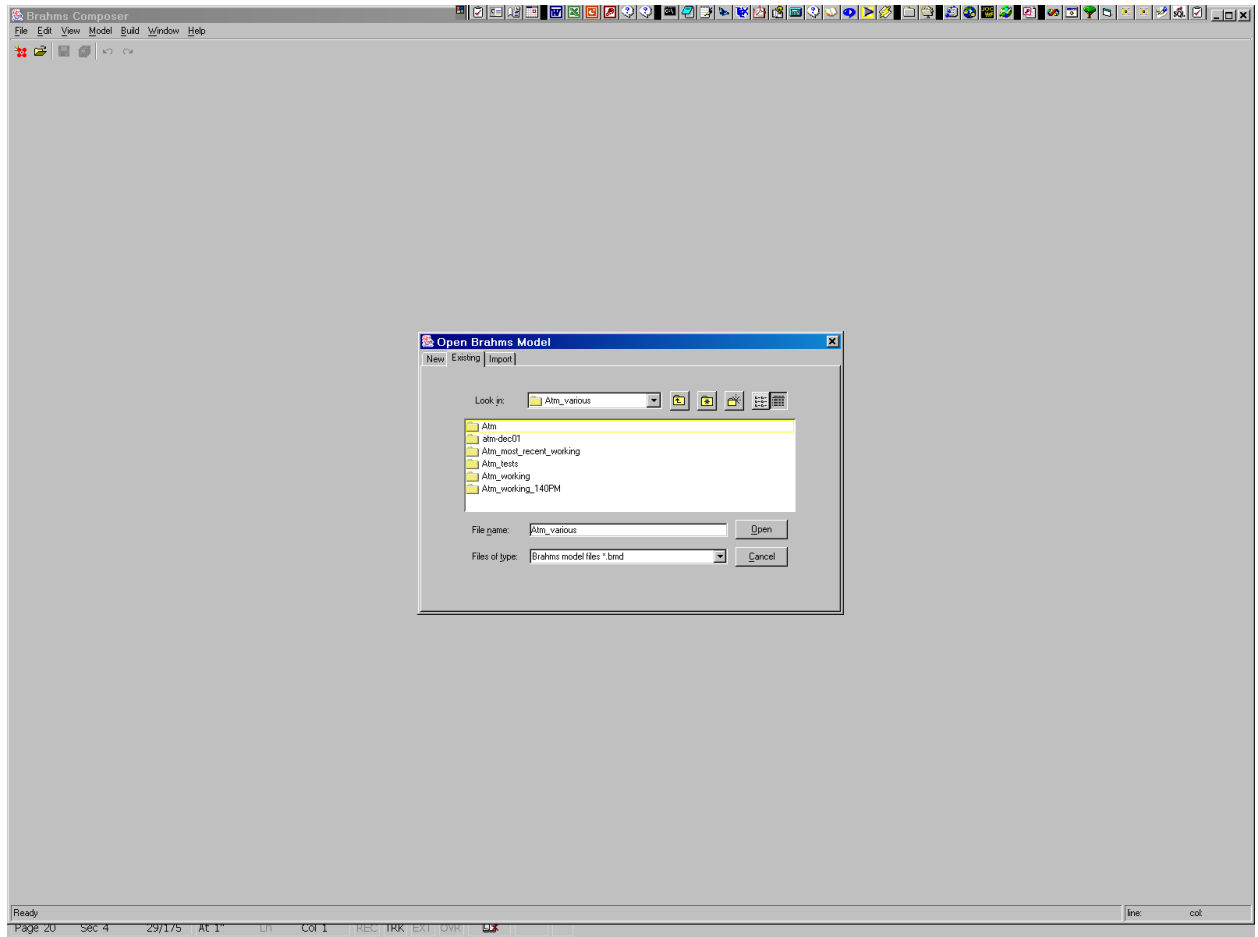


Figure 3 - Opening the Composer

Ok. So, we are going to assume that you have followed the steps above (installed the Brahms components, created the new directories, and downloaded the Atm Model files). This should mean that your Brahms application files have been saved in a `\Brahms\AgentEnvironment\` folder in your root directory and that your model files have been saved in `c:\Brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\`.

Now, launch the Composer (look for the `bin` folder and click on `Composer.exe`) choose “Import” tab in the Composer dialog window (see Figure 4), look for the `c:\Brahms\Projects\AtmModel` folder, and then look into the folder:

```
c:\Brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm
```

Basically, we want to test the files for the “Final_source” model. In this folder, select as “File_name” the file `AtmModel.b`. If a library path is request, add: `c:\Brahms\Projects\AtmModel\final_source` (we will explain soon why). Click on open. You should see something like Figure 1 (although not necessarily all windows will be opened).

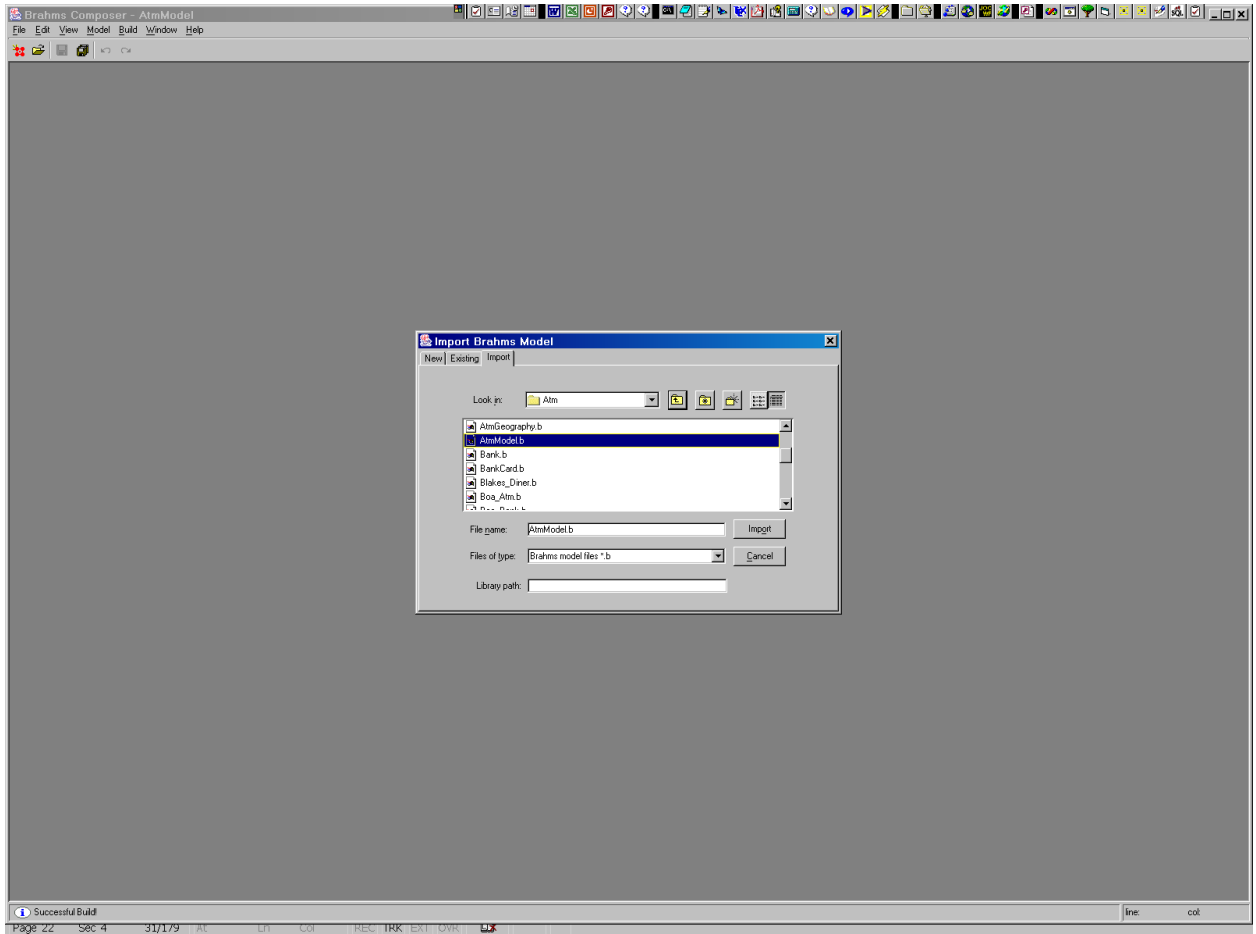


Figure 4 - Importing the Atm complete scenario in the Composer

Now go to the “File” menu (top left) and choose: “Save Project.” Congratulation, you have just imported and then saved again your first Brahms model! Now exit the Composer (in the “File” menu, choose “Exit”).

Start the Composer again. This time, however, do not choose the “Import” tab but rather the “Existing” tab. Look for the `c:\Brahms\Projects\AtmModel\final_source` folder and open the `AtmModel.bmd` file. This is the file which did not exist before and that has been created by the Composer when you chose to save your project. Again, as your model opens, you should see something like Figure 1. You have just learnt two ways to achieve the same goal – open in the Composer an existing Brahms model!

In addition, the Composer also grants you access to the Brahms Compiler.

By opening or compiling the files in the Brahms Composer, the modeler will be able to check whether they are syntactical correct or whether there are other errors in the code, and to produce xml files that can be fed into the Brahms Virtual Machine.

Before we see how this works, we must clarify the issue of that library path we mentioned above.

As the name says, with the library path we tell the Brahms Compiler what path to use when compiling files. By default, the Composer will look for the `Models\lib` folder in the `AgentEnvironment` folder, and if you wish so, you could certainly use this folder. However, several users prefer to keep working projects separated from application files, in different folders or even in different drives, and in this tutorial we want to teach you how to use your Brahms model files in a different directory. This is why you were asked in the previous section to create the set of nested directories inside `c:\Brahms\Projects\AtmModel\final_source`. This will be a useful exercise to start learning about how Brahms work – and will also make your future Brahms projects more flexible and reusable.

Technical note: The above approach comes at a little cost, however. If you want to use a particular directory to work on the files of your Brahms models (as we are forcing you to do in our exercise; in our case, the directory is: `c:\Brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm`), you must add a few parameters in the Brahms environment. In particular, you may have to modify a `vm.cfg` file to read in the library path that you are going to use for your models. This file is located in the `AgentEnvironment` directory created during installation, and by default it is set to:

```
library_path=<installation directory>/Models
```

“Models” refers precisely to the directory inside `AgentEnvironment` we were mentioning above. Now, let us assume that instead you want to use a `c:\Brahms\Projects\AtmModel\final_source` folder to save all your models (included the `Atm`). Then you should modify your `library_path` information to look like:

```
library_path=c:/Brahms/AgentEnvironment/Models/lib;  
c:/Brahms/Projects/AtmModel/final_source;
```


Ok.⁶ Now note: use of '/' as the path separator is important even though Windows uses '\' as its path separator, as an alternative you can use a double '\(\!'. As you see, you can add other directories to this path by separating the directories with a semicolon (;). In this case we have left two directories – hence, if you like, you might keep on using the default *Models/lib* directory as the location for your model files.

Anyway – let's go back to the Composer. You opened or imported your Atm model. Now, what do you do with it? Well, you can compile it so that the Simulation Engine will be able to run a simulation of your model. To compile your code from the Brahms Composer, simply go to the "Build" menu when your model is open, and choose "Build Model." There you go – if all goes well, at the bottom of the Composer window a message will appear telling you that the build was successful. Some xml files will have been created – but we will discuss this later, in 4.3.

Technical note: if you prefer to go "manual" and directly access the underlying Compiler, you should execute the following from a MS-DOS command line:

```
c:\Brahms\AgentEnvironment\bc.bat -dtd c:\Brahms\AgentEnvironment\DTD -lp
c:\Brahms\AgentEnvironment\Models\lib;C:\Brahms\Projects\AtmModel\final_source
C:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\AtmModel.b
```

where the first part of the command (up to *bc.bat*) calls the Brahms Compiler executable, the second part (*-dtd c:\Brahms\AgentEnvironment\DTD*) tells the Compiler the path for the XML document type definition files referenced in the Brahms Compiled Code (bcc files), the third part (*-lp c:\Brahms\AgentEnvironment\Models\lib;C:\Brahms\Projects\AtmModel\final_source*) tells the Compiler the library path, and the last part passes to it as an argument the *AtmModel.b* file – which is a Brahms file that imports all the other files of the Atm project (or 'package': more about this in 4.3.3, where you can also see an example of code that imports other files for a project).⁷

*Additional technical note: if you want to go "manual," then, in order to use the Compiler (as well as the Virtual Machine, that we will discuss in a moment) from other places on your machine (without having to spell out all the directory structure), make sure to add the Brahms *AgentEnvironment* directory to your system PATH environment variable: that is, open your *autoexec.bat* file and add the directory where the *bc.bat* and *bvm.bat* files can be found to the line with the PATH: command.⁸*

⁶ Why did we leave about the 'c:\brahms\Projects\AtmModel\final_source' directories? This will become clearer in section 4.3.3, but if you want a rough explanation now, keep on reading. Basically, the Brahms language uses 'packages' that are mapped to a directory in the file system. The package declaration represents a hierarchical directory structure and is written in the Brahms model files themselves. For example, the Atm files will be written as part of the package *gov.nasa.arc.brahms.atm* that maps to the directory *gov\nasa\arc\brahms\atm* in the file system. Hence, you do not need to specify the package in the configuration files – you only need to specify where this (and any future packages) can be found!

⁷ This parser creates the xml files from the .b files; it is possible to have a .b file that imports all the files in a package, with the expression *import package.**; If the parser starts from this file, it will automatically compile all the others (cf. 4.3.3 on the anatomy of a Brahms file).

⁸ To add or changes values of environment variables in windows 2000/XP/Vista/Win7 use the following procedure:

1. Click on **Start > Settings > Control Panels**.
2. Double-click on **System**.
3. On the **Advanced** tab, click **Environment Variables**.
4. Select the user or system variable you want to change as follows:
5. Click **New** to add new variable name and value.

Printed on: This is an uncontrolled copy when printed.

3/31/11 3:08 PM Refer to the NX Brahms location for the latest version.

Either way (click on the “Build” button on the Composer panel, or use the command line), the Brahms Compiler will start and the model will be compiled. The outcome of the compilation will be a set of bcc files produced in the very same `c:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\` folder.⁹

3.3.3 THE BRAHMS COMPOSER AND THE VIRTUAL MACHINE: RUNNING A MODEL

After successfully compiling your model, go to the “Run” menu in the Composer and click: “Run Model.” You should see a number of messages in “VM Log” window, such as:

```
INFO : Starting engine for 'gov.nasa.arc.brahms.atm.WF_Atм'  
INFO : Starting engine for 'gov.nasa.arc.brahms.atm.Boa_Bank'  
INFO : Virtual machine started...  
INFO : Stopping virtual machine  
INFO : Stopping scheduler  
INFO : Stopping engine for 'gov.nasa.arc.brahms.atm.Alex_Agent'  
INFO : Stopping engine for 'gov.nasa.arc.brahms.atm.Kim_Agent'  
INFO : Stopping engine for 'gov.nasa.arc.brahms.atm.Campanile_Clock'  
INFO : Stopping engine for 'gov.nasa.arc.brahms.atm.Boa_Atм'  
INFO : Stopping engine for 'gov.nasa.arc.brahms.atm.WF_Bank'  
INFO : Stopping engine for 'gov.nasa.arc.brahms.atm.WF_Atм'  
INFO : Stopping engine for 'gov.nasa.arc.brahms.atm.Boa_Bank'  
INFO : Stopped event notifier
```

But what does it mean to “run” a Brahms model? When you select “Run” from the Composer, a component called Brahms Virtual Machine runs the simulation by reading the bcc files produced by the compiler and producing a text file with the history events captured during the simulation. This text file can be parsed by the Agent Viewer application to produce a complete event history database of the simulation.

-
6. Click **Edit** to modify the highlighted variable (i.e. PATH) If you are not logged as an **administrator** to the local computer, you can only create/change/delete **User Variables**. These variables will only be accessible to the particular user. No reboot is required. You just need to open a new terminal window if you had one open for the changes to take effect in the terminal window.

⁹ A complete guide to the Composer is available from the AgentiSolutions website (<http://www.agentisolutions.com/>). The information presented in this section *only* covers the most basic steps needed to start working on the ATM scenario.

Printed on: This is an uncontrolled copy when printed.

3/31/11 3:08 PM Refer to the NX Brahms location for the latest version.

Technical note: the Virtual Machine can be also called from MS-DOS (rather than from within the Composer) by executing `bvm.bat` and by giving it the relevant `bcc` filename, without the `bcc` extension.¹⁰

Given that the Brahms Compiler saves the `xml` files in the same directory where the `.b` files are saved, the Virtual Machine is called through `bvm.bat`, which must be given the filename of the file that imports all other files. For example, in the case of the `Atm` model, supposing that the `.b` files have been created in the `c:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm` directory, the Compiler will save there also the `bcc` files, and the necessary command to run the simulation will be:

```
bvm -cf C:\Brahms\AgentEnvironment\config\vm.cfg gov.nasa.arc.brahms.atm.AtmModel
```

where `AtmModel`, again, represents the `bcc` file created by the Brahms file that imports all the other `Atm` project files. The example above assumes that `bvm.bat` - that resides in the `Brahms\AgentEnvironment` directory - can be called from anywhere because the `PATH` variable in the `autoexec.bat` file has been already modified. Furthermore, remember that the `vm.cfg` file has a library path that must contain the path where your `xml` files can be found:

```
library_path=c:/Brahms/AgentEnvironment/models/lib;C:/Brahms/Projects/AtmModel/final_source
```

Anyway: the Virtual Machine will first initialize and load the concepts of your model, and then it will start the engine. When it stops,¹¹ a history file with a `.txt` extension will have been created in the `AgentEnvironment/Databases` folder, with the format: 'model_name_date_time': e.g., `Atmmodel_20010418_100523.txt`. Everything that happened in the simulation is in this file.

Technical note: it is possible to add an undocumented flag `-ui` as an argument to the `bmv.bat` command in order to halt the simulation at the modeler's discretion. An applet will start when the Virtual Machine is running the simulation; this applet will let you halt the process and still obtained a file that the Agent Viewer - discussed below - can parse.

3.3.3.1 IMPORTANT: WHEN THINGS DON'T WORK...

At this stage, there are a couple of things that might be wrong. If a history file is not created, then most likely you don't have the proper setup in your library path. In the `AgentEnvironment\config` directory you will find a `vm.cfg` file. Again, check that the library path contains the path where your `xml` files can be found:

```
library_path=c:/Brahms/AgentEnvironment/models/lib;C:/Brahms/Projects/AtmModel/final_source
```

The folder `C:/Brahms/Projects/AtmModel/final_source` has been created automatically when you unzipped tutorial files. (note that you can separate multiple paths using the semicolon or comma). The second thing that could be the cause for the problem is an incorrect use of packages. We will discuss packages in 4.3.3 but for the moment it will suffice to recall that, if your Brahms files have statements like:

```
package gov.nasa.arc.brahms.atm;
```

then your model would have to be loaded using:

¹⁰ Soon the Virtual Machine will be also accessible directly from inside the Composer.

¹¹ By default, Brahms simulations will run until the model stops executing because no agent or object has any activity left to perform. Therefore, it is also possible for a simulation to go on indefinitely if any of its agents or objects never ceases being in an activity. This is not the case of the models provided for the `Atm` tutorial, so you do not have to worry about this for the moment.

```
bvm -cf C:\Brahms\AgentEnvironment\config\vm.cfg  
gov.nasa.arc.brahms.atm.<modelfilename>
```

which in our case is:

```
bvm -cf C:\Brahms\AgentEnvironment\config\vm.cfg  
gov.nasa.arc.brahms.atm.AtmModel
```

because the file `AtmModel.b` contains an import statement for all other `.b` files in the `Atm` directory (note that the directory in which 'projects' is specified must be in the library path as well).

Another problem could be that the license file has not been installed (see 3-13). Also, if things are not working, you may want to check your `vm.cfg` file and check that the `library_path` is set to the directory where your projects are saved.

Yet one more problem could be that something in the way the Brahms Composer deals with projects has changed since the time this Tutorial was written. Since the Brahms language and the Brahms environment are still evolving, it is possible that, for example, the paths used by the Brahms Composer to save or run projects have been modified. If you do not find files where this Tutorial says you should find them, first try to look for alternative and simpler locations (for example, rather than searching in `c:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\`, try looking into `c:\brahms\Projects\AtmModel`). If unsuccessful, please contact us for help at support@agentsolutions.com.

3.3.4 BRAHMS AGENT VIEWER

The final step is to open the Agent Viewer from within the Composer, and 'parse' the history of events text file that can be found in the `AgentEnvironment\Databases` folder.

What you need to do is, first, make sure that you have installed MySQL 5.1 and that its Client is running on your machine. (in order to watch simulation results, make sure that the `MySQLAdmin.exe` is running – that will show as a “green” semaphore in your system tray on Windows operating systems).

Next, go to the File/Agent Viewer menu in the Composer, and “create” a new database by selecting the history of events text file that can be found in the `AgentEnvironment\Databases` folder.

The Agent Viewer will create a new database with the same name as the history text file (something similar to `AtmModel_20010415_134559`). The database can be opened by clicking on the File/Agent Viewer/Open Database menu option. (Agent Viewer by default will try to open the most recently created database file, so you will simply have to click on “yes” in the selection window.)

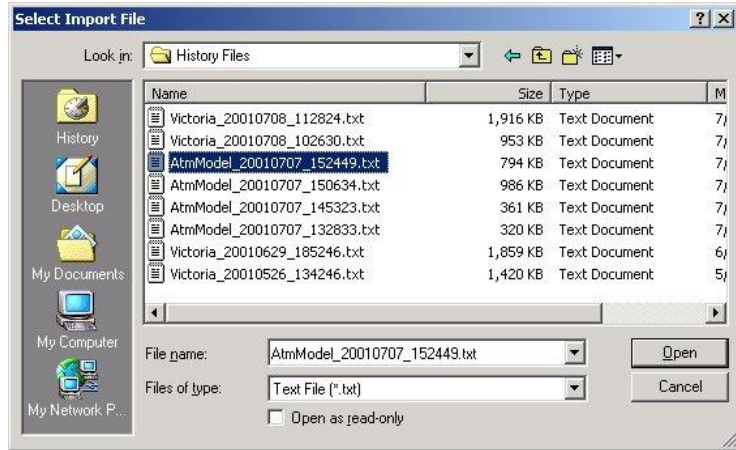


Figure 5. Select a history file to parse in the Agent Viewer

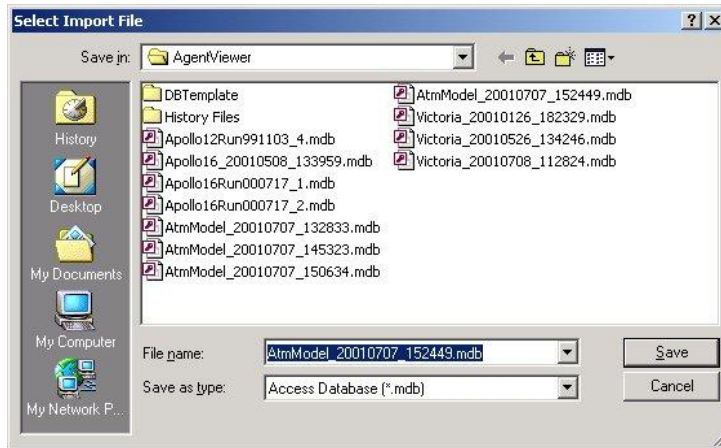


Figure 6. Creating history database

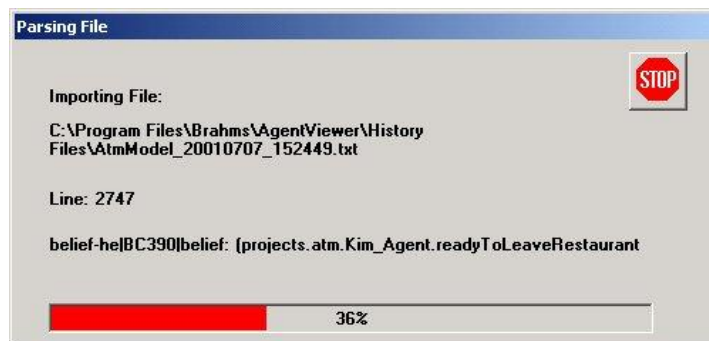


Figure 7. Parsing history file into history database

Finally, go to the 'View' menu inside the Composer and look for "Time Line View." The operations of the model are visually verified using the Agent Viewer Time Line View. The Agent Viewer is a separate application inside the Composer that uses the simulation history data to display a 2-dimensional graphical time-line view of the activities of agents and objects. The following timeline figures are all screenshots from selected agents and objects in the Agent Viewer. Using the Agent Viewer application the modeler can investigate the simulation run and the properties of the model.

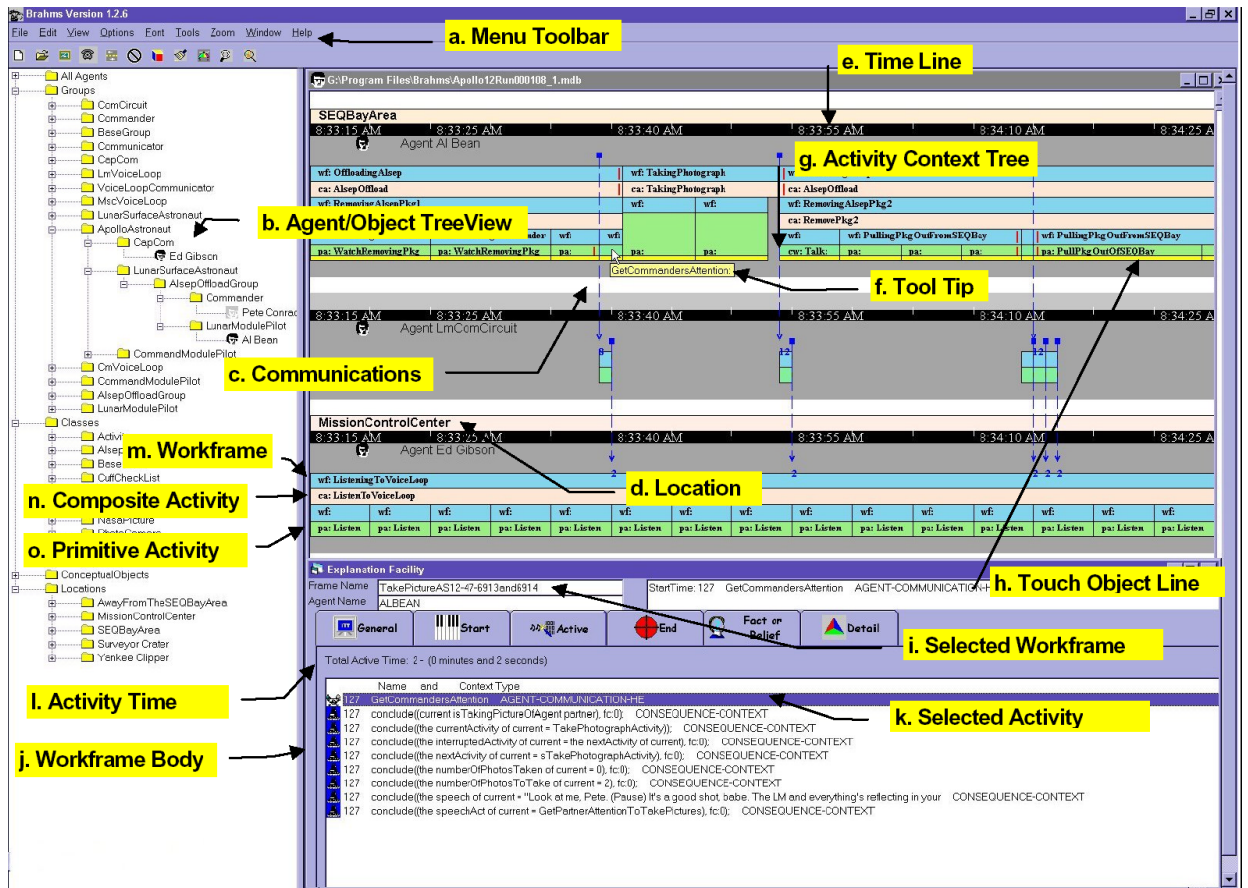


Figure 8. Agent Viewer Application – as of July 2004

Figure 8 shows the Agent Viewer. Using this application the end-user can select which agents and objects to view in the time-line view, and investigate the exact behavior of those agents and objects during the simulation):

- a. Using the menu-bar, the end-user can parse the simulation history data into a history database, and open a history database for viewing.
- b. When the database is opened all the agents and objects are loaded into the tree view. Using the tree view, the end-user can select which agents and/or objects (s)he wants to view in the time-line view.
- c. By selecting to view the agent/object communication, the (blue) arrows show all the communication activities, and the direction of the communication (sender and receivers). The communicated beliefs are also accessible by clicking on the square at the top of the sender side of the communication arrow.
- d. For each agent/object the "current" location is shown. When the agent/object moves to a new location, it is shown as a change in the location name and color.
- e. The time-line can show the time in different time-intervals, therewith zooming in and out.
- f. The tool-tip pops up when the mouse is moved over "hot spots". The hot spots are those areas where more information is available than can be shown on the screen. By moving the mouse over those areas the hidden information pops up in a tool-tip, such as the name of a workframe or activity.
- g. The Activity-Context Tree is the central piece of the agent/object time-line. It shows the workframe and activities hierarchy of the agent or object.
- h. The touch-object line is a (yellow) line that is shown when the agent/object is using certain objects in its activity. "Touch objects" are used to calculate the time those objects are used in activities.
- i. The explanation facility view is used to display more detailed information about the execution of workframes. By clicking on any workframe (light blue in color), an explanation facility window is opened for the workframe at hand.
- j. By selecting the "Active" tab in the explanation facility view, the executed statements in the workframe body are shown.
- k. You can select the statements in the workframe body to get more info.
- l. When you select a statement in the body of the workframe, the total time the activity was active is shown. Using the other tabs in this view, you can find out the exact time the workframe became available, as well as the exact time it became active and ended.
- m. Workframes are situated-action rules that execute activities. The top of an Activity-Context tree is always a workframe. You can recognize a workframe by the "wf:" symbol, followed by the name of the workframe. When the zoom-level is too high to contain the name of the workframe it is left out of the display. Using the tool-tip the user can find out the name.
- n. Composite Activities are executed by workframes, and contain lower-level workframes. You can recognize Composite Activities by the "ca:" symbol followed by the name of the activity. When the

zoom-level is too high to contain the name of the activity it is left out of the display. Using the tool-tip the user can find out the name.

- o. Primitive Activities are executed by workframes, and are always at the bottom of the Activity-Context Hierarchy. You can recognize Primitive Activities by the following symbols, depending on the type of primitive activity: “pa:” (for a primitive activity), “mv:” (for a move activity), “cw” (for a communicate activity), “co:” (for a create object activity), followed by the name of the activity. When the zoom-level is too high to contain the name of the activity it is left out of the display. Using the tool-tip the user can find out the name.

Using the Agent Viewer it becomes possible to visually inspect the simultaneous behavior of the agents and objects, and compare the expected behavior from the conceptual model with the actual behavior during the simulation.

3.4 SUMMARY OF STEPS

To summarize, these are the steps necessary to write, compile and run the Atm scenario (and by extension, any Brahms simulation):

1. Download and install the Brahms Agent Environment
2. Download and install MySQL 5.1
3. Obtain a Brahms license file.
4. Download and unzip the Tutorial files.
5. Open the Atm files for the “Final_source” model inside the Composer (by opening the `atm.bmd` file or by importing the `.b` files), or use the Composer to write the Brahms file(s), i.e. files with the `.b` extension written according to the rules and specifications of the Brahms language.
6. Compile/build them inside the Composer (or, if you want to go “manual”, do it by using `bc.bat`; for example, `c:\Brahms\AgentEnvironment\bc.bat -dtd c:\Brahms\AgentEnvironment\DTD -lp c:\Brahms\AgentEnvironment\Models\lib;C:\Brahms\Projects\AtmModel\final_source C:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\AtmModel.b` where `AtmModel.b` is a file that imports the other files in the `gov.nasa.arc.brahms.atm` package - more on package in the next chapter - and `c:\brahms\Projects\AtmModel\final_source` is the library path).

7. Run the Virtual Machine (simulation engine) on the xml files created by the Composer in the same folder where you saved the .b files. You can do so with the "Run Model" command in the Composer, or, if you want to go "manual", by issuing the following command line in MS-DOS: `bvm -cf c:\Brahms\AgentEnvironment\vm.cfg gov.nasa.arc.brahms.atm.AtmModel.`
8. Parse the resulting history of events .txt file from the Agent Viewer, which you can access from the Composer, and then open the database the parser has just created.

3.5 A NOTE ON DEBUGGING...

While the Brahms Composer provides syntactic verification of your Brahms files, the system does not yet offer full debugging capabilities. Debugging a Brahms model can be a difficult task and some tips will be provided at the end of the tutorial, in section 4.14.2 and chapter 5.

3.6 KNOWN BUGS IN BRAHMS AGENT ENVIRONMENT

A list of known bugs for the current version of the Brahms Agent Environment is available online from <http://www.agentisolutions.com>.

3.7 CONTACTING THE BRAHMS PROJECT TEAM FOR TECHNICAL SUPPORT

If you have questions about this tutorial, you can reach the curator at: acquisti@agentisolutions.com. In case you have problems with the installation or questions and problems with the use of the Brahms components that the tutorial does not address, you can reach Technical Support at:

E-mail: <mailto:support@agentisolutions.com>

www: <http://www.agentisolutions.com/support>

The support team also actively participates in the discussion forums located on the AgentiSolutions website. Support questions or questions about Brahms programming and modeling can be posted in those forums. The Brahms community will try to help you out whenever they can. You can find the discussion forums at: <http://groups.google.com/group/brahms-forum?hl=en>.

3.8 OTHER IMPORTANT DOCUMENTS

- Brahms Language Specification; updated version available online at http://agentisolutions.com/documentation/language/ls_title.htm.
- Brahms Tutorial Lite. A compact version of this tutorial, updated online at http://agentisolutions.com/documentation/tutorial/tt_title.htm.
- Brahms Composer Manual. A guide to the IDE used to build Brahms model and a good companion to this tutorial, updated online at <http://agentisolutions.com/documentation/>.
- Maarten Sierhuis's PhD thesis, from which are taken parts of this document: Modeling and Simulating Work Practice: Brahms, A multiagent modeling and simulation language for work systems analysis and design. The book can be purchased online from AgentISolutions website: www.agentisolutions.com.
- On the theoretical foundations of Brahms: Clancey, W. J., Sachs, P., Sierhuis, M., and van Hoof, R. (1998). "Brahms: Simulating practice for work systems design." *International Journal on Human-Computer Studies*, 49, 831-865.

Additional sources for each chapter are reported in footnotes at the beginning of that chapter. A References chapter is included at the end of this document.

3.9 LATEST CHANGES

The Brahms language is continuously evolving. While we will make an effort to keep this tutorial always updated, in some cases new constructs might be added to the language and old constructs might be removed or changed and not be immediately reported in this document. For the latest information on the Brahms language specifications, you should regularly visit: http://agentisolutions.com/documentation/language/ls_title.htm and verify the "Document History" information. The most recent changes in the language that have not yet been integrated in this tutorial are:

- put and get activities
- create agent activity
- java activities
- unknown values
- maps

- java integration

For more information on these concepts, please read the online Language Specifications.

3.10 DOCUMENT INDEX

You can use the Index at the end of this document to search for items and keywords.

Printed on:

3/31/11 3:08 PM

This is an uncontrolled copy when printed.

Refer to the NX Brahms location for the latest version.

4. ATM SCENARIO¹²

Let the game begin! From this chapter on, you will be driven step by step through a Brahms version of the Atm scenario introduced in the previous sections. You will be shown Brahms concepts and then asked to use them in increasingly realistic and complex representations of an Atm scenario. You will be given lots of examples about what those representations should look like. Your final goal will be to model the following scenario:

Model a day in the life of a college student. Notoriously, college students spend most of their time studying, but get hungrier as the time goes by. When their hunger reaches a certain threshold, students have to move to one of the restaurants around their location. Students choose restaurants according to how much money they are carrying: the richer they are, the more expensive a restaurant they will choose. If a student does not have enough money even for the cheapest restaurant, she will decide to pass first by an Atm of the bank where she has an account.

When she arrives at the Atm, the student inserts her bankcard and tries to remember the PIN associated to her account. The Atm allows its user 3 attempts to digit the correct PIN, before refusing the card altogether. The Atm communicates with the central bank computer to verify the correctness of the information provided by the user. If the bank computer informs to the Atm that the PIN is correct and that the user has enough balance in her account, the Atm will dispense the cash and will print an invoice with the account number and the remaining balance.

Students need to have enough balance in their accounts to take out cash: if they attempt to take out more money than they have, the bank computer will notify the students (through the Atm) of the remaining balance of the account. The student will then need to modify her request accordingly, and only take out the remaining dollars.

THE CAST (AGENTS and OBJECTS)

Students: Kim, Alex

Bank computers: Bank of America, Wells Fargo

Restaurants: Blakes, Raleighs

Studying Places: South Hall, Spraul Hall

Clock: the Campanile

Atms: one Atm for each bank

¹² Sources: Brahms Language Specification TM99-0008 v2.2; Maarten Sierhuis' PhD Thesis; communications with Brahms development team members; Atm tutorial files and comments; Brahms Forum postings.

A few comments. The complete scenarios will be (most likely) quite long and will include several files. This tutorial will drive you through the scenario from start to end. However, while in the beginning all the details will be described and all the code to implement the cases will be provided, as you will progress in the tutorial less and less details will be provided, and more will be left to your skills and your imagination. For example, you will have to think for yourself how to model the reaction of the bank when a wrong pin is inserted, or what happens when too little money is left into the account. For your own verification, however, you have also been provided pieces of codes that represent “snapshots” of the Atm scenario at different stages of its evolution. Remember though that no single solution is the only the right one!

4.1 STRUCTURE OF THE SCENARIO

Each of the following sections will introduce you to a concept of Brahms modeling: the basic compilation unit (section 4.3); the geography (section 4.4); groups, agents, attributes and relations (section 4.5); facts and beliefs (section 4.6); workframes, thoughtframes and activities (sections 4.7 and following); objects and classes (section 4.8); variables (section 4.10); as well as advanced topics like the interaction of many agents, the use of priorities, and the use of random elements (sections from 4.9 on).

Each section will adhere to the same structure: a subsection titled “Introduction” will describe the scope of the section; the “Task” subsection will present you with the modeling goal to be accomplished by the end of the section; the “Description” subsection will discuss, more or less formally, the language concepts of interest to the section.¹³ The “Syntax” subsection will link to the complete syntactical details and rules of the concepts being discussed. Finally, the “Tutorial” subsection will be the area where the “goal” and the “description” meet: it will drive you through the actual coding of the Atm scenario. As a general rule, it will be useful to try to code each section’s scenario on your own, right after reading the “Task” and the “Description,” before reading the “Tutorial” section. Complete model files are also available and linked to from the “Tutorial” sections, as described in **Error! Reference source not found.**

4.2 EXPECTATIONS AND GOALS

The reader should have some level of previous programming experience to get the most from this tutorial. Knowledge of object-oriented languages and rule-based languages is preferable, but not essential.

¹³ The material in these sections is based on Maarten Seirhuis’ PhD Thesis, cfr. 3.8.

This tutorial will provide a self-contained introduction to the language. It will start with simple concepts and examples and builds up in complexity and difficulty. By the end of this tutorial you should be able to model easy to mildly complex scenarios on your own. The tutorial however will *not* cover all of the language features. Once you get going, you should refer to the online language specifications for a complete view of the language.

Experienced programmers might move relatively quickly through the initial sections of the Tutorial (Sections 3 – 8) and spend more time on the final sections, where more complex activities and constructs will be introduced (Sections 9 –11). Even experienced programmers, however, should try to keep in mind that Brahms is different from object-oriented languages: Brahms is an agent-oriented language with elements of rule-based languages, where “activities” are not the same things as “methods,” and “Workframes” are not the same as “`if..then`” constructs of imperative languages.

One final note before kick-off: this tutorial will *not* follow the way you are supposed to build a model in Brahms, because you still do not know about Brahms concepts. As we have described at the end of the previous chapter, once you master Brahms you should start your scenarios from the various ‘model views’ presented in 2.3.2.2. We will come back to this issue in section 4.14. And remember: there is no substitute for practice to learn a new programming language. As mentioned above, while in the beginning all the details will be described and all the code to implement the cases will be provided, as you will progress in the tutorial less and less details will be provided, and more will be left to your judgment. You will have to come out with good solutions to the exercises and challenges proposed in the text. Of course, you will be given examples of code to compare and verify your own solutions.

4.3 LESSON I: GETTING STARTED

4.3.1 INTRODUCTION

This chapter will get you started with your first Brahms file – your first ‘compilation unit’.

4.3.2 TASK

Create a Brahms file that contains a package declaration and an import declaration to be used during the development of the Atm tutorial.

4.3.3 DESCRIPTION: COMPILATION UNIT

A [compilation unit](#) is a file with the extension “.b.” Quite simply, we might say that a compilation unit is each and any of the Brahms files in your Brahms model. Why do we need several files to simulate one Brahms model? Well, that’s up to you as a modeler. In theory, you could write your whole model in one, single, long .b file. This practice however is highly discouraged. Practicality and clarity, among other reasons, suggest using different files for different concepts (for example, a file for each group, a file for each agent, etc.) stored in the same folder (or in folders which are part of the same tree).

Technically, 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
- Import statements that allow types from other packages 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.

The package declaration appears within a compilation unit to indicate the package to which the compilation unit belongs. The Compiler loads a ‘.b’ file when it is referenced in an import declaration. Hence, the first compilation unit that you will write for your Atm model will be a .b file whose only purpose is that of loading all the other .b files that are part of the project.

4.3.4 TUTORIAL

The first step should be the creation of a set of nested directories:

```
c:\Brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\
```

(if you have not created them already, do it now; rest assured that you can create your Atm directory everywhere you want, even though in the examples described in this tutorial and in the instructions on how to use the Brahms Compiler and Virtual Machine batch files it will be assumed that this is in fact the location of the directory you will be using. If you modify it, remember to change accordingly also your library path in the vm.cfg file and the way you call the Virtual machine, as discussed in the previous chapter). If you had created the folders and had unzipped into the `c:\brahms\Projects\` folder the Atm files downloaded from the AgentiSolutions website, we suggest you delete all those files and the AtmModel directory itself – we have to start again from scratch and you have to build your own model now!!

Now, the second step: either from the Composer or from a text editor, create your new model.

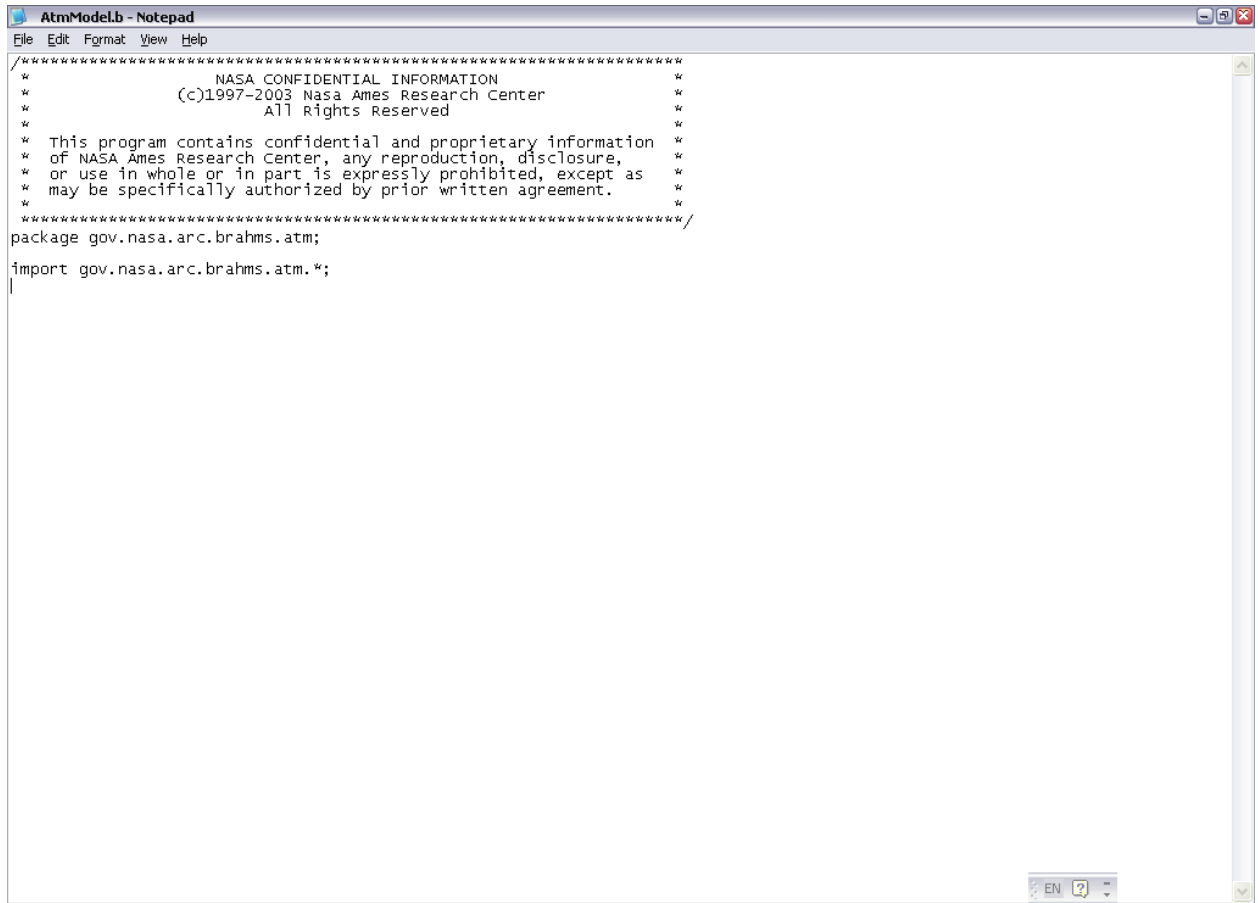
Technical note: depending on which version of the Tutorial files you have downloaded, the files may be installed by the Composer in a slightly different nested directory. Don't worry - for this tutorial we will start again building files from scratch from the `c:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\` directory.

If you use the Composer, you must first start it, then select the “New” tab, and fill in the following information in the respective fields: the ModelName will be `AtmModel`, the package will be `gov.nasa.arc.brahms.atm`, and the location will be `c:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\`. If you now go to the `c:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\` folder, you will notice that the Composer has created the right directories for you and, inside the `atm` folder, a file called `AtmModel.b`.

Before seeing what that file contains, let's see what is the procedure to start a model if you are working from scratch without the Composer – just with a text editor such as Notepad. You would create, again, the `atm` folder and a new document in Notepad, where you will write:

```
package gov.nasa.arc.brahms.atm;  
import gov.nasa.arc.brahms.atm.*;
```

Then, you would save this file as '`AtmModel.b`'. It does not matter that, for the moment, you do not have other files to import – these will come soon! Your file might look somewhat like the following figure:



```
AtmModel.b - Notepad
File Edit Format View Help
/*****
 *          NASA CONFIDENTIAL INFORMATION          *
 *      (c)1997-2003 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.atm;

import gov.nasa.arc.brahms.atm.*;
|
```

Figure 9. AtmModel.b in Notepad

What is the reason for these two lines?

Firstly, the package declaration is used to find Brahms concepts in the file system. Similarly to Java, in Brahms a package is to be mapped to a directory in the file system. The package declaration represents a hierarchical directory structure. The package `gov.nasa.arc.brahms.atm` maps to a directory `gov\nasa\arc\brahms\atm` in the file system. If a group Student were defined in a file named `Student.b` then this file would be located in the `gov\nasa\arc\brahms\atm` directory. The Compiler and Brahms Virtual Machine use the library path to find concepts in a specific package relative to the library path. Thus, 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. In our case, if the library-path is

```
library_path = c:\\brahms\\Projects\\AtmModel\\final_source
```

and you have an import statement like

```
import gov.nasa.arc.brahms.atm.Student;
```

Printed on: This is an uncontrolled copy when printed.

3/31/11 3:08 PM Refer to the NX Brahms location for the latest version.

then the concept Student is expected to be defined as

```
package gov.nasa.arc.brahms.atm;  
group Student { }
```

and is expected to be found in the file

```
c:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\Student.  
b
```

This is why in section 3.3.2 we suggested to add `c:\brahms\Projects\AtmModel\final_source` to the library-path. Note that the file created automatically by the Composer will be exactly the same.

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.

Secondly, the import declaration allows a type declared in another package to be referred to by a simple name that consists of a single identifier. The import declaration makes concepts defined in other compilation units available as one model. In our example, by using the star (*), we are automatically importing all the .b files that will be found by the Compiler in the Atm folder.

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.

4.3.5 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_cun_stx.htm and pages linked from there.

4.4 LESSON II: GEOGRAPHY

4.4.1 INTRODUCTION

This chapter will teach you how to create the geography of a Brahms model.

4.4.2 TASK

Create the geography of the Atm model. The action takes place in Berkeley: there are streets and buildings, like restaurants and bank branches with their Atms. You will model at least two different Atm locations (one for each Bank: Bank of America and Wells Fargo) and two restaurants (Blakes, and Raleighs). You will also have to describe the distances between these locations.

4.4.3 DESCRIPTION

The geography of a Brahms model is described through areas, `area_definitions` and paths. Mind you: this is not a Cartesian geography, it's an abstraction based on concepts (such as areas) and the way those concepts are connected (with paths).

An [area definition](#) is used for defining types of area instances used for representing geographical locations in a model. Area definitions are similar to classes in their use. Examples of area definitions that are already built in the language are 'World', 'Building', and 'City'. Through area definitions, the modeler can create new area types or extends those already existing – for example, by creating the area definition for Restaurant, that `extends` Building.

An [area](#) is an *instance* of an area definition. An example is the area 'Berkeley', instance of City. Areas can be decomposed into sub-areas. For example, a building can be decomposed into one or more floors, or a city into streets, using the *part-of* relation to other areas that is available in each area.

A [path](#) connects two areas and represents a route that can be taken by an agent or object to travel from one area to another. The modeler specifies distance as the time it takes to move from area1 to area2, over the path. The automatic generation of location facts and beliefs for agents and objects moving from one area to another is also implemented. This means that when an object or agent changes location, the simulation engine automatically generates a new location fact about the new location of the agent or object that moved, and also adds the location fact as a belief to the belief-set of the agent or object. At the same time, the simulation engine will add the location belief to all agents in the location. When an agent moves, the simulation engine will also add a location belief for all the other agents and objects in the location to the belief-set of the moving agent. In other words, objects and agents always “know” where they are, and agents will always notice (we use the term “detect”) other agents and objects in its location. However, moving agents and objects do not notice other agents or objects on their path, even though The move activity can be interrupted by a communicate activity in a workframe triggered by a detectable or a belief communicated by another agent (more on this later...). Agents and objects can move through the entire geography.

An agent or object has the functionality of a container artifact and can carry other agents or objects (cf. the *contains* relation in the next section). The geography may be implemented to limit the generation of beliefs to other agents in the immediate environment of the specific agent. When an agent or object moves from one location to another location the simulation engine calculates the shortest route between the two locations, based on the available paths between the areas in the model. It should be noted that when an agent or object moves between two areas that do not have a path between them, the simulation engine assumes that the moving time is zero. However, if there is a route (i.e. a path between the areas) the simulation engine will ignore the zero-time distance without a path (otherwise, it would always take the zero-time distance).

Note that areas can have relations and attributes. An *area_definition* can inherit from more than one area definition, so multiple inheritances are 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. Attributes and relations will be explained in the next sections.

4.4.4 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_adf_stx.htm, http://www.agentisolutions.com/documentation/language/ls_are_stx.htm, http://www.agentisolutions.com/documentation/language/ls_pat_stx.htm, and pages linked from there.

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

4.4.5 TUTORIAL

Create a new file in a text editor with a .b extension and call it AtmGeography.b. Or, if you want to use the Composer, just open your Atm model and click on the “Geography Model” tab at the bottom of the screen to access the Geography graphic interface. Then, click on the menu-button with the letters “are” on it and the icon of a filled green shape. This will create a new “Area_1” concept whose name you will change into “AtmGeography” by simply clicking on the icon of the concept on the screen (Figure 10). Remember to save your model!

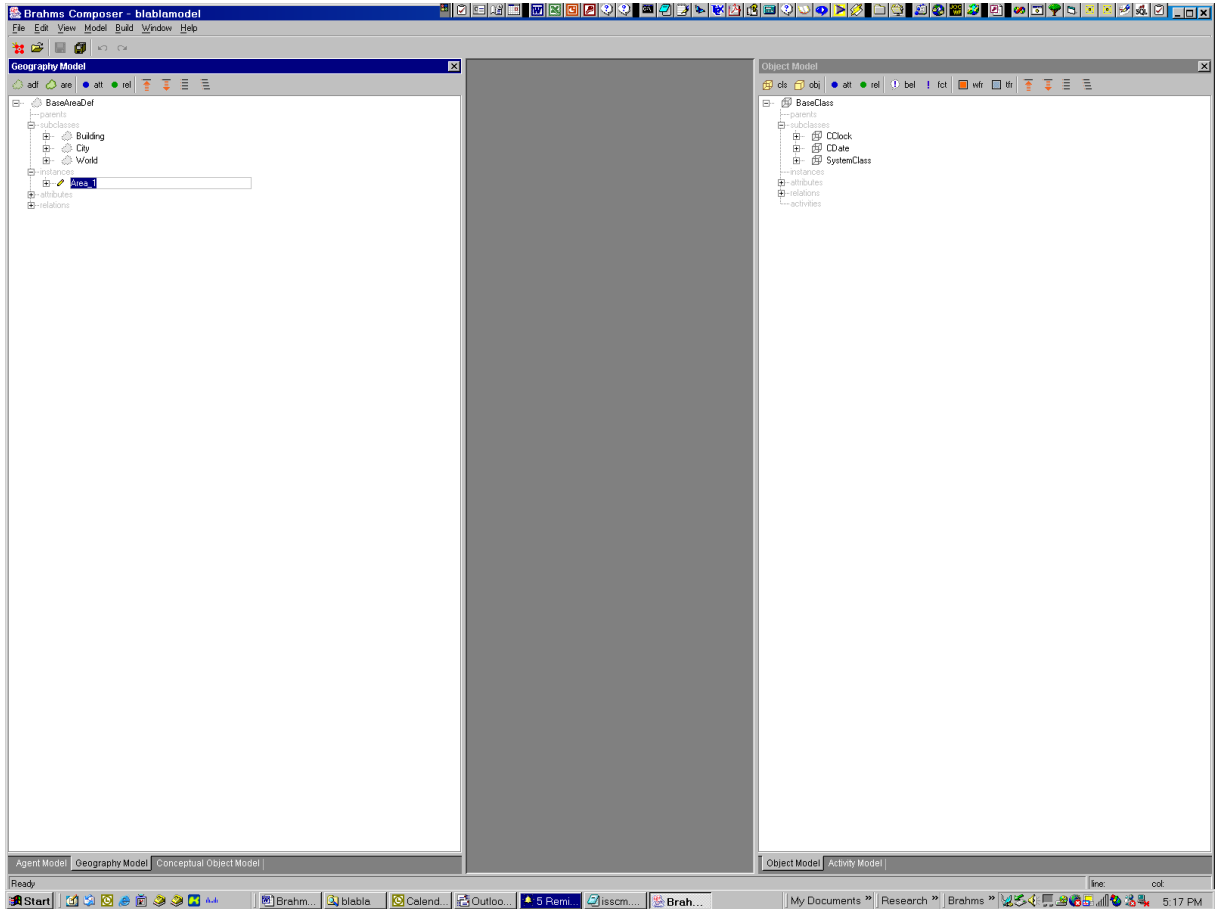


Figure 10 - Creating geography areas in the Composer

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

Important note: we will not keep on presenting both approaches – the text editor and the Composer - for each concept that we will meet in the Tutorial. The Composer manual will tell you how to create concepts via the graphic interface. In this tutorial, instead, we will focus on the language concepts. However, remember that whenever we refer to files being created or edited, this can be done both with an external text editor such as Notepad, or, much more quickly and smoothly, inside the Composer, which offers you advanced features to create and modify concepts and files.

If you are writing your own code, then you will have to insert into this file the following elements: 1) a package declaration (to state that this file belongs to your Atm project); 2) definitions of new areas ([areadef](#): such as world, city, buildings, streets, etc.) and instances of these [areas](#); 3) [paths](#) describing the distance (in terms of the time necessary for movement) from one location to another.

Hence, start writing:

```
package gov.nasa.arc.brahms.atm;
```

to establish that this geography belongs to the AtmModel (the Compiler adds the declaration automatically). Then, you can create the instances of two area definitions already defined by default in the language (World, and City):

```
area AtmGeography instanceof World { }  
area Berkeley instanceof City partof AtmGeography { }
```

What we are saying is that the action will take place in the AtmGeography 'world', and in particular in Berkeley. A complete geography will need other components, such as a University (namely the University of California Berkeley), University Halls, Streets, Restaurants, and Bank Branches. Given that BaseAreaDef and Building are standard area definitions already defined in the language, you can write:

```
areadef University extends BaseAreaDef { }  
areadef UniversityHall extends Building { }  
areadef BankBranch extends Building { }  
areadef Restaurant extends Building { }
```

Note that if you are using the Composer, many of the above code lines will be produced automatically by the Composer as you create, drag and drop concepts, and use the efficient pop-up fields to plug in parameter values.

By using the terms `extends` and `partof` you have just drafted an area type schema for the geography in which the actions of agents and objects of the Atm model will take place. `extends` is used to create a new area definition which inherits characteristics from another area definition. `partof` is used to represent the hierarchy or geographical relations between areas. For example, Wall Street might be `partof` New York, which is `partof` the United States. The next step is to populate this schema with specific instances of the area definitions you have just created, to create a hierarchical geography description of the locations in which agents and objects can be placed. You will need a university, some restaurants, some banks, and some university halls where your student agents will study. Note that comments, in Brahms, can be expressed either by preceding them with `'/'`, or by containing them within:

```
/*  
comment  
*/
```

Hence, if we imagine that Berkeley campus (UCB) is inside the city of Berkeley and that it contains several University Halls, while the banks and the restaurants are in Berkeley but outside the campus, we will write:

```
// inside Berkeley  
area UCB instanceof University partof Berkeley { }  
area SouthHall instanceof UniversityHall partof UCB { }  
area Telegraph_Av_113 instanceof BankBranch partof Berkeley { }  
area SpraulHall instanceof UniversityHall partof UCB { }  
area Bancroft_Av_77 instanceof BankBranch partof Berkeley { }  
area Telegraph_Av_2405 instanceof Restaurant partof Berkeley { }  
area Telegraph_Av_2134 instanceof Restaurant partof Berkeley { }
```

Almost there: The final step consists of connecting together these different areas to allow agents and objects to move between areas. When agents move between areas, you can either state (each time moving between locations makes it necessary) what the distance is between the two areas the agent has to move (expressed as time needed for the move activity); or define this once with the `path` statement in the geography file. We will use this second option here:

```
//paths to and from banks from spraul and south halls  
path StH_to_from_BOA {  
    areal: SouthHall;  
    area2: Telegraph_Av_113;  
    distance: 240;  
}  
path SpH_to_from_BOA {  
    areal: SpraulHall;  
    area2: Telegraph_Av_113;
```

```
        distance: 240;
    }
    path StH_to_from_WF {
        area1: SouthHall;
        area2: Bancroft_Av_77;
        distance: 200;
    }

    path SpH_to_from_WF {
        area1: SpraulHall;
        area2: Bancroft_Av_77;
        distance: 200;
    }

    //paths to and from restaurants from and to spraul and south halls
    path StH_to_from_BB {
        area1: SouthHall;
        area2: Telegraph_Av_2134;
        distance: 360;
    }

    path SpH_to_from_BB {
        area1: SpraulHall;
        area2: Telegraph_Av_2134;
        distance: 280;
    }

    path StH_to_from_RB {
        area1: SouthHall;
        area2: Telegraph_Av_2405;
        distance: 400;
    }

    path SpH_to_from_RB {
        area1: SpraulHall;
        area2: Telegraph_Av_2405;
        distance: 320;
    }

    //paths to and from restaurants and banks
    path BOA_to_from_BB {
        area1: Telegraph_Av_113;
```

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

```
        area2: Telegraph_Av_2134;
        distance: 60;
    }
    path WF_to_from_BB {
        areal: Bancroft_Av_77;
        area2: Telegraph_Av_2134;
        distance: 80;
    }
    path BOA_to_from_RB {
        areal: Telegraph_Av_113;
        area2: Telegraph_Av_2405;
        distance: 80;
    }
    path WF_to_from_RB {
        areal: Bancroft_Av_77;
        area2: Telegraph_Av_2405;
        distance: 100;
    }
}
```

As mentioned above, you can also do all the above in the Composer – in fact with the Composer you will save lots of time and keystrokes. However, given that here we want to learn the details of the language rather than the use of the Composer itself, the above typing exercise will prove useful anyway....

Some comments about this code. Note that if you model paths from a to b and from b to c but *not* from a to c, when you issue a move command from a to c the Compiler will automatically calculate the total travel time for you.

Agents do not notice anything or do other things on their way to a location. If you want something like that to happen, then you have to explicitly code it. For example, if you want to model the possibility that a student can meet another student on the street while on his road from area A to area B, you might split his movement into 2 steps: from A to the street (modeled as an area), and from the street to B. The student agents can now meet each other in the street area. This would definitely be a more advanced Brahms model, and thus we do not expect that you are able to create such a more complex model at this stage¹⁴.

¹⁴ We are currently extending the language allowing for real multi-tasking. This way you can create a move activity in which the agent can notice other agents and do other activities while moving. At this moment the Brahms language does not support this kind of multi-tasking in so-called primitive activities.

As hinted above, there are at least 3 ways to model distances and movements in Brahms: 1) not using *paths* at all, but using the *move* activity instead (see section 4.7) and defining each time how long a certain move activity should take; 2) defining all the paths between places, as done in this section (if you forget to define the path and also the move-time, the Compiler will interpret that as it taking zero time); 3) have travel times that change according to the conditions—for example, if there is a path from a to b requiring 10 seconds that has been coded in a geography file, you might override that within the move action; you might even code the time as a belief of the agent (so that the agent modifies this belief according to factors like weather conditions or transportation means being used). An additional way to move something in a Brahms model is to have an object—say, a motorcycle, that contains another agent or object: the motorcycle will actually do the move activity, and the total travel time will depend on it rather than the speed of the agent. Containment of agents and/or objects is realized with a `contains` relation. We will discuss the `contains` relation in section 4.8.

Now you are ready to try and compile your Atm model for the first time. Follow the steps described in the previous chapter. Remember to choose the “Build Model” menu option in the Composer once your Atm model is open. If everything goes well, some xml code will be produced. If something goes wrong, either in the command line interface or in the Composer some error messages will appear. In particular, the Composer will give you information about possible syntactical and semantic errors in your code, with the exact line and column where they have been found.

Some final notes: as you will soon see in the next sections, an agent needs an initial belief about the location of an object (say, an Atm machine) to move there. Remember that area definitions can have attributes, and agents can have beliefs about them—this might turn useful in your models.

4.5 LESSON III: GROUPS, AGENTS AND ATTRIBUTES

4.5.1 INTRODUCTION

This chapter will teach you how to define and create groups (e.g. Students) and agents that are members of groups. It will also teach you about attributes and relations of Brahms groups and agents.

4.5.2 TASK

Create the group Student and agent Alex from the model schemas you have previously drafted. Alex is a student and will have attributes and relations that are relevant to the Atm scenario, such as; where agents study, feel hungry, and thereafter go and eat with money they obtain from Atms. In this section we will start with a small subset of attributes (for example, the gender, the hunger, and so on) and relations (like hasCard, and hasAccount), which will be expanded on as the Tutorial progresses.

4.5.3 DESCRIPTION

4.5.3.1 AGENTS & GROUPS

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 about the agent 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."

In Brahms we use the notion of *strong agency*: Brahms agents model human behavior, and the Brahms modeling language implements in some way all of the attributes discussed in context of weak and strong agency, i.e. autonomy, social ability, reactivity, pro-activeness, mobility, and bounded rationality.

People and artifacts (both physical and conceptual) are represented as "objects", generally having properties, such as geographical location, which may change over time depending on their interactions. The term "agent" is generally used to refer specifically to an object that represents a person or, more inclusively, that represents an interactive system that has behavior interacting with the world that we want to represent as having the capabilities of awareness, reasoning and a mental state - intentionality.

An [agent](#) is a construct that generally represents a person within a workplace or other setting being modeled. Agents have a name and a location. To specify what an agent does, the modeler defines activities and workframes for the agent. The key properties of agents are group membership, beliefs, workframes, thoughtframes, and location.

Groups and members of groups

In Brahms, [Agents](#) are members of groups. It is possible to model actual individual's behavior—for example, the activity of an agent 'Alex' in the Atm model: the activities will then be defined local to agent 'Alex', and will be agent-specific (i.e. not inherited by other agents). However, most Brahms models will not go into as much detail as to define the activities of individual agents, but rather describe the behavior of abstracted groups of agents in entities called *groups*. In describing the activities of groups of agents, a specific agent will inherit the activities of the group. In this way we can describe the daily activities of the group in a more abstracted way (i.e. non-individual specific), but we can make it specific through the parameters and the attributes of each specific agent.

A *group* can represent one or more agents, either as direct members or as members of subgroups. Typically, a modeler would associate descriptions of activities with groups, so that a group represents a collection of agents that perform similar work. A group may have only one member and roles may be highly differentiated. Depending on the purpose of the model, agents in a model may represent particular people, types of people, or pastiches. The modeler may define groups to represent anything, such as "service technicians", "people who like sushi", or "people who wear spacesuits". Each agent and group can be a member of any number of groups, providing that no cyclic membership results.

Groups and agents are the most central elements in a Brahms model. An agent represents an individual, whereas a group represents a group of individuals playing a particular role in an organization. The simulation engine schedules the constrained activities of agents, not for groups. However, being a member of a group, the group's constrained activities are also scheduled as they pertain to the individual agent. A Brahms model is always about the activities of individual agents in a work process. Agents in Brahms are socially situated in the context of work, the organization, and its culture. However, the Brahms language is a multi-purpose AOL, which means that there is nothing inherently in the Brahms language that constraints the programmer to use agents for other purposes.

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. An agent can be a member of one or more groups. When an agent is a member of a group the agent will 'inherit' (or 'instantiate') 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 by agents, 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 an attribute defined for the agent has precedence over an attribute with the same name defined in one of the groups of which the agent is a member.

Elements of agents and groups

Brahms agents and groups may have the following elements.

Name: The name of an agent is its unique identifier. This element is *not* optional. Normally we give agents fictitious names to identify specific individuals in an organization without identifying them.

Display: The display name of an agent is an *optional* textual description of the agent's name. The display name can have spaces. It is not used as the unique identifier for the agent.

Group-membership: An agent can be a member of one or more groups. This element is optional. When an agent is a member of a group the agent will inherit all elements from the group(s) it is a member of. An agent can be seen as a member of a group. In case the same constructs are encountered in the inheritance path always the most specific construct will be used. For example, 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.

Cost and time: The cost per unit ("Cost/unit"), and the unit time for which the cost is entered ("Unit (seconds)"). This element is optional. For example, if the cost attribute is 10.0 and the time attribute equals 3600 seconds it means that the cost of an agent is 10 BB's (Brahms Bucks) per hour. Using these attributes the simulation engine can calculate cost statistics of a work process, based on a calculation of the summation of an agent's activity time.

Location: An agent can have an initial location within the geography (see the previous section, 4.4). This element is optional.

Attributes: Represent a property of an agent or object in the world. Attributes can have values. Currently only single-valued attributes are allowed. The value of an attribute is specified through facts and/or beliefs (we will describe attributes in more detail later in this section). This element is optional.

Relations: Represent a relation between two concepts. A concept can be either an agent or an object. 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. Relations are specified through facts and/or beliefs (we will describe relations in more detail in section 4.8). This element is optional.

Initial-beliefs: A belief is a first-order predicate statement about the world. This element is optional. Beliefs are always local to an agent, i.e. only the agent can access its beliefs, and no other agent can. This allows us to represent how a specific agent ‘views’ the state of the world. Agents act based on their beliefs. Beliefs are the ‘triggers’ of agent’s actions. We will discuss beliefs in section 4.6. *Initial beliefs* define the initial state for an agent. Initial beliefs are turned into actual beliefs for the agent when the model is initialized at simulation start time.

Initial-facts: Facts represent the state of the world. A fact is a first-order predicate statement about the world. This element is optional. 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. Initial facts define the initial state of the world. Initial facts are turned into facts in the world when the model is initialized for a simulation run. There is a fundamental difference between the “ownership” of a belief and a fact. A belief is “owned” by a specific agent during the execution of the model. No other entity in the model can access that belief without some interaction with the agent (direct or indirect). However, although initial-facts are defined with an agent or object, at execution time a fact is not “owned” by that agent or object. A fact is global, and can be acted on (in the case of objects) or detected (in the case of agents). We will discuss facts in section 4.6.

Activities: In this element the activities an agent can be engaged in are defined. This element is optional. Activities in Brahms take a certain amount of time, either derived or defined. There are a number of types of activities that are defined for the Brahms language. Activities defined are executed by workframes. We will describe activities in section 4.7

Workframes: In this element the activity rules, called “workframes”, are defined. This element is optional. Workframes describe the constraints of executing activities. Workframes are situation-action rules. We will study workframes in section 4.7.

Thoughtframes: In this element the agent’s inference rules, called “thoughtframes”, are defined. This element is optional. Thoughtframes are inherently different from workframes, as they do not execute any activities. We will discuss thoughtframes in section 4.9

4.5.3.2 ATTRIBUTES

[Attributes](#) represent a property of a group, agent, object class or object. Attributes have values. In Brahms we currently only allow single-valued attributes. The value of an attribute is defined through facts and/or beliefs.

Attribute scope

Attributes are always defined within a group, agent, conceptual-class, conceptual-object, class, object definition, area or area 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*.¹⁵

Private attributes:

Private attributes are scoped down to only the concept for which it is defined. A 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 is done through the creation of individual *beliefs* and *facts* for agents and objects, rather than simply assignment operators like '=' or ':='. We will discuss this issue in much more detail in the next sections.

4.5.4 SYNTAX

Syntax details are available at:

¹⁵ Note that attribute and relation scopes are currently *not yet* implemented in the language. This implies that attribute/relation scope definitions will be ignored by the Compiler, and all attributes/relation will be treated (for the time being) as public. This is why in the code examples presented in the text all attributes and relations are treated as 'public'.

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_grp_stx.htm, http://www.agentisolutions.com/documentation/language/ls_agt_stx.htm, http://www.agentisolutions.com/documentation/language/ls_att_stx.htm, and pages linked from there.

4.5.5 TUTORIAL

With this section, you will start populating your virtual world with some actors. Start creating a file `Student.b` (or use the Composer, go to the “Agent” tab, and create a new agent by clicking on the menu-button with the letters “agt” close to the symbol of one hand). If you are writing your own code, then you must write the following (do not forget the package declaration; as noted above, the Compiler adds the declaration automatically and lets you insert the parameter values discussed below – such as attributes, etc. - through efficient pop-up fields):


```
package gov.nasa.arc.brahms.atm;
group Student {
    attributes:

    relations:

    initial_beliefs:

    initial_facts:

    activities:

    workframes:

    thoughtframes:
}
```

It's ok that for the moment we have left most items blank. You have to start with something, right? 'Group Student' is the declaration that you are going to use to describe a new [group](#) with that name. You are not declaring such group as member of any other group, which means that the Group Student will not inherit any previous attribute, or beliefs, etc, from other groups (apart from the default BaseGroup every Brahms group is based upon). On the other side, you could create more complex group hierarchies and memberships than the one we use in this tutorial, where students are simply members of the Student group. For example, you might model the group 'BankAccountHolder' and then make agents belong to both the Student and the BankAccountHolder group. Or you might have groups of groups – for example, a group EconomicStudent part of the group Student. These are modeling issues that you will have to face after you have mastered the basics of the Brahms language. For the moment, however, be careful about simpler issues such as the ordering of the components inside the Group Student definition: even if currently you do not know much about things like workframes or initial-beliefs, you must pay attention to write the *declarations* in the correct order (which is the order presented above) to avoid errors during compilation.

The group 'Student' is representative of all students in Berkeley, but in particular it must be suited to perform the tasks we want to model in the Atm scenario. A good way to achieve this goal is to start giving it proper [attributes](#) and relations. Let's start adding these lines to the Student group file:

```
attributes:
    public boolean male;
    public double howHungry;
```

```
public double preferredCashOut;
```

In Brahms, attributes can be of type boolean, double, int, symbol and string, but also other user-defined type (groups, classes, areas) and Brahms built-in meta-type definitions (such as Agent, Group, Class, etc.). 'Symbol' is just any value you want to give to an attribute without referring to it as a string. This has some advantages: it is easier to write down values for attributes that way (for example, rather than having a boolean type attribute 'male', we could have a symbol type attribute 'gender', which can have the values 'female' or 'male'). A symbol type can be seen as an enumerated type in languages such as Pascal, and C, except that we do *not* have to declare the possible values. Any value is possible.

In this tutorial, we start giving students some simple distinguishing characteristics: whether their gender is female or male, how hungry they are, and how much money they normally take out from their accounts when they go to an Atm. As you can see, for the moment we are not specifying any of these values: the student can be either female or male, can be very hungry or not hungry at all (let's assume that the higher the howHungry attribute is, the hungrier we consider the student), and can decide to take whatever cash he or she wants out of the bank. So, where do we specify these values?

The answer can be given on two levels. First, [attributes](#) (as well as relations we will study in section 4.8) values are specified through *facts* and *beliefs* that we will model in the next section. We can specify such facts and beliefs right inside the Group Student body (did you notice there were already the labels for the initial-facts and initial-beliefs sections in the source code for the group Student?). Defining initial beliefs and/or facts in the group is appropriate only when they are common to the entire group. When this is not the case, we better specify these values inside the body of the [Agent](#) that is member of that group.

Now, create a new file and call it: `Alex_Agent.b`. Make sure that the name of the file is the same as the name of the concept that is being defined in the file; in this case `Alex_Agent`. Then add the following in the newly created file:

```
package gov.nasa.arc.brahms.atm;
agent Alex_Agent memberof Student {
    location: SouthHall;
    initial_beliefs:
        (current.howHungry = 15.00);
        (current.male = true);
        (current.preferredCashOut = 8.0);
    initial_facts:
        (current.male = true);
        (current.preferredCashOut = 8.0);
}
```

Alex_Agent is the first agent in the Atm model. It is a member of the group Student, which means that every activity that will be described for the Student group, Alex_Agent will be able to perform. The structure of an 'Agent' file can be exactly the same as that of a Group: we could have, again, workframes, thoughtframes and so on. The reason we rarely do so is because we almost never model specific activities, workframes and thoughtframes at the agent level: we prefer to model activities as activities of a group, performed by particular agents. However, there is nothing wrong with defining these elements at the agent level. In this case, we are only giving a particular location to our Agent: this means that when the simulation starts, Alex_Agent will be in that initial location (e.g. South Hall), while other agents might start elsewhere. Note that in this scenario we will model all the activities that agents perform (studying, eating, getting money from the Atm) within the Student group. That is: agents, such as students, will study, eat, get money from the Atm, etc. Other strategies are of course available: you might create other groups—for example, Bank Account Owners—and model activities for that group—such as going to the Atm; then, you would make the agents of your model be also members of that group. What solution is best? That is a modeling decision. Just remember the inheritance rules: in case the same constructs are encountered in the inheritance path, always the most specific construct will be used. This means that an element defined for a group the agent is directly member of, has precedence over an element with the same name defined in one of the groups of which the former group is a sub-group.

Hold on! We will not yet run the simulation. A little more patience and you will arrive to that. As you have seen, even though this section was about agents, groups and attributes, we could not help making reference to other concepts such as *beliefs* and *facts*. All of these concepts are interrelated in Brahms, because agents are the most important concepts. Agents act on the basis of facts and beliefs in a world made of objects, areas, other agents, activities, workframes, and thoughtframes . Hence, facts and beliefs is what we are going to explore in the very next section, in order to understand what the statements like '`current.preferredCashOut = 8.0`' in the initial-beliefs and initial-facts section actually mean.

4.6 LESSON IV: FACTS AND BELIEFS

4.6.1 INTRODUCTION

This chapter will teach you about facts and beliefs in Brahms.

4.6.2 TASK

The balance of a bank account; the price of a lunch at the restaurant; the time of day, etc. There exist many ‘facts’ in a Brahms model—and at least as many ‘beliefs’ for each agent! In fact, everything that happens in a Brahms model is related to either a fact, or a belief, or both. Hence, now start giving your agent(s) initial facts and beliefs about their environment that they will need in order to act in their world.

4.6.3 DESCRIPTION

The state of the world in Brahms is stored as propositions called “facts”. A fact is meant to represent some physical state of the world or an attribute of some object or agent. Facts are global: with the appropriate *detectable* any agent can detect a fact (we will discuss this later in section 4.7). Agents and objects can have “beliefs.” Beliefs are propositions (like facts) that represent the internal “mental” state of an agent or object. Beliefs are always local to the agent or object.

4.6.3.1 BELIEFS

A [belief](#) represents a particular object knowing about something, typically an agent. A belief always has the form:

(<object or agent>.<attributename> <operator> <value>)

OR

(<object or agent> <relation-name> <object or agent> <is true | is false>)

Beliefs are always local to an agent or object; that is, only the agent or object itself can examine, or search (i.e. reason with) its own beliefs, and no other agent or object can do so. A belief held by an agent may differ from the corresponding fact or a belief that another agent has about the same fact. Beliefs can be declared as initial-beliefs at the object class, object, group, or agent level, or an agent or object can create beliefs during the running of a Brahms simulation. A belief can be thought of as an object-attribute-value triplet. There are four ways an agent or object’s beliefs can change:

1. An agent or object can create or change a belief before or after performing an activity or during some reasoning.
2. An agent or object can detect a fact in the world, which, once detected, becomes a belief.
3. Another agent or object communicates its belief(s) to another agent or object.
4. The modeler adds initial-beliefs at the class, object, group, or agent level.

The representation of beliefs together with reasoning implements a conventional first-order predicate logic of beliefs. The modeler has available the full range of representation of, and reasoning on, beliefs conventionally found in rule-based systems such as EMYCIN and other agent languages. However, because the logic is first-order, agents are not modeled as having second-order beliefs (beliefs about other agents' beliefs).

Agents and objects in Brahms have beliefs represented as first-order propositions. For instance, suppose agent *A1* believes that he is writing his dissertation, and that it will be finished on time. *A1* would then have the belief set:

{ BEL(Is-Writing (A1, Dissertation)), BEL(Will-Finish-On-Time(A1, Dissertation)) }

An agent will always start out with an *initial-belief* set that is defined at the agent's local-level, and the groups of which the agent is a member. Initial-beliefs are assigned in the initialization phase of a simulation. These initial-beliefs define the initial state for the agent. An agent without an initial state could be seen as initially 'dumb', or an agent that has not experienced anything yet. As the simulation time moves forward agents will infer, detect and receive new beliefs, either based on their actions and communications in the world or deducing new beliefs, using an inference rule.

4.6.3.2 FACTS

[Facts](#), in Brahms, are factual states of the world. They represent, what we call, a 'birds-eye view' of the world. Facts are global to the world, meaning that they can be 'seen' by every agent and object in the world.

Therefore, in Brahms there is a difference between *facts* that actually hold in the world, and beliefs of an agent or object. This means that we can represent the facts in the world separate from the belief-state of agents about that world. For instance, although the fact is that the color of my car is red, I can believe that the color of my car is green, because I might be colorblind. In representing the context of the agent as facts in the world, we are able to have multiple agents react on the same facts in different ways, dependent on their beliefs about these facts. In Brahms, it is not necessary that *any* agent has *any* belief about a fact. Specifically, facts are independent from the knowledge of agents (see Figure 11).

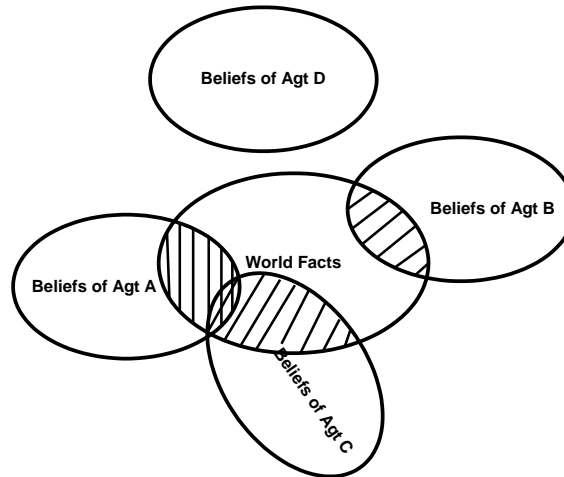


Figure 11. Beliefs and facts Venn diagram

4.6.4 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_bel_stx.htm, http://www.agentisolutions.com/documentation/language/ls_fct_stx.htm, and pages linked from there.

4.6.5 TUTORIAL

Let's go back to the code we were working on in the Agents and Groups section:

```
package gov.nasa.arc.brahms.atm;
agent Alex_Agent memberof Student {
    location: SouthHall;
    initial_beliefs:
        (current.howHungry = 15.00);
        (current.male = true);
        (current.preferredCashOut = 8.0);

    initial_facts:
```

```
(current.male = true);  
(current.howHungry = 15.0);  
(current.preferredCashOut = 8.0);  
}
```

A [belief](#) is a first-order proposition that an agent or object believes to be true.

At the beginning of the simulation, you can give your agents initial beliefs about himself and other agents, objects, concepts in the world. During the simulation, the agent can get new beliefs – through his own workframes and thoughtframes as well as by communication. Similarly, facts are state of the world that you can assign at the beginning of the simulation as ‘initial facts’ and then modify during the simulation in a variety of ways. Importantly, facts are ‘general’ – they exist in the world and they can be detected as well as modified by other agents or objects; but 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 can represent information stored in/on the object. Agents act based on their beliefs, whereas objects do *not*. Beliefs are the ‘triggers’ of agent’s actions (cf. section 4.7). Hence, when we will study about ‘conclusions’ in workframes and thoughtframes (see sections 4.7 and 4.9) we will see that an agent can ‘conclude’ beliefs only for himself, even though he might ‘conclude’ facts about another agent’s attribute.

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. In this case, we are giving agent Alex_Agent some initial beliefs about himself. The use of ‘current’ in the above code has the very same scope of the term ‘this’ in C++ or Java. It always refers to the agent or object executing that particular construct, or, in other words, it always refers to the element inside whose body declaration the ‘current’ term is being used. Note, furthermore, that attributes can be referred to very much like in Java in the construct: object.attribute. Older version of the Brahms language also admitted the construct: ‘the attribute of object’ but it is preferable to use the newer and more Java-like syntax.

Similarly, [facts](#) represent the state of the world. A fact is a first-order proposition 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 the belief. Objects on the other hand, only react to facts. 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. In the example above we are stating for example that the attribute `male` of the agent is a fact in the world that other agents can detect, and potentially act upon.

Not always things are so straightforward. Is the hungriness level of Alex_Agent only a belief or an actual fact? One might argue for both options. The hungriness level of a person is based on some specific fact, i.e. it is something which exists in the world - the status of the agent's endocrine system, which can be detected with appropriate measurements and tests and which might have an effect on its own even if the agent does not detect it or as different beliefs about it (think for example of an agent so deeply involved in an activity that he forgets about eating, while his body is instead increasingly in need for food). On the other side, what would be the use of making the hungriness level a fact in the world of this scenario? Note in fact that facts in the world can be detected by other agents or objects: so, do we want such an internal fact as the hungriness state of an agent to be (potentially) public knowledge? Of course, the other agents and other objects in the model do know automatically know about all the facts in the world: they only see or act upon depending on how the modeler decides to model his scenario (for example, another agent would need a 'detectable' - more on this in section 4.13 - to get information about a fact 'hungriness level'). On the other side, you do *not* need any fact to model the belief of an agent, but you do need facts to make objects react (objects do *not* react on beliefs). This dichotomy involves some philosophical discussion, that we leave aside for now, but it suffices to say that it leaves open several approaches to the modeling of attributes such as the physical state of a human agent (here, his 'hungriness level', which could have been modeled just as a belief only) as well as of complex artifacts such as Banks and Atms in our model (should they be objects or agents)? In the code attached to this Tutorial, we have chosen to model banks and Atm's as objects, but you should always remember that more than one approach is possible in Brahms, and a few might be equally correct (we will discuss these issues in particular when we will discuss detectables).

On the other side, an agent needs to have a belief (even if an 'uncertain' one: see next section for an explanation of what that means in Brahms), in order to use it. For example, workframes are triggered when preconditions are met (cf. 4.7). For a precondition to be met, the agent must have at least a belief about the existence of the particular concept considered in the precondition.¹⁶ An agent does *not* automatically know even about its own attributes. For example, in the Atm scenario, the hungriness level is an attribute of Students. Even after modeling its value for Alex_Agent as a fact, we still must set up a belief for the agent in order for Alex to know and act after it.

¹⁶ Note, however, that the agent does not need a prior belief about a concept in order to receive a communication about it.

So, when should we use beliefs and when do we use facts? Be careful: facts in the world can be known by everybody in the world with a detectable to detect that fact: another agent can know your account balance if you model it as a fact (and, of course, if you model that other agent's detectable of your bank account). A rule of thumb is the following: when you are modeling flows of data, you might preferably use beliefs; if you are dealing with things happening and changing in the physical world, then you might instead use facts. It is up to you as a modeler to find the right combination between realism and practicality, specificity and generality. There will be several cases in the Atm scenario (as well as in all the other scenarios you will model in Brahms), where more than one option is available to you. Take cash and Atm machines: in the coming pages we will be modeling cash as an object that belongs to the student, with an attribute "balance" that represents the total amount of cash the student is carrying. Other ways of modeling this are also possible. Similarly, the Atm will be modeled as an object (which implies that it will react only to facts), but it could also have been modeled as an agent. The choice is the modeler's. The rule of thumb is to try to represent things as close to your world experience as possible. Thus, one reasonable good rule is: Humans are modeled as agents and everything else is modeled as objects, most importantly all objects in the world.

4.7 LESSON V: WORKFRAMES AND PRIMITIVE ACTIVITIES

4.7.1 INTRODUCTION

This chapter will teach you how to use activities and frames in Brahms models.

4.7.2 TASK

Finally, some action inside the model! Alex_Agent is hungry and needs food, therefore he goes to the restaurant, where his 'hungriness level' is automatically decreased.

4.7.3 DESCRIPTION

4.7.3.1 WORKFRAMES

Recall the conceptual approach to modeling in Brahms that we have described in chapter 2:

```
GROUPS are composed of
  AGENTS having
    BELIEFS and doing
      ACTIVITIES executed by
        WORKFRAMES defined by
          PRECONDITIONS, matching agent's beliefs
          PRIMITIVE ACTIVITIES
          COMPOSITE ACTIVITIES, decomposing the activity
          DETECTABLES, including INTERRUPT, IMPASSES
          CONSEQUENCES, creating new beliefs and/or
facts
```

An [activity](#) is an abstraction of real-life actions that help accomplish a task. A model of an agent's activities describes what the agent actually does over time (i.e. its actions) based on decision-points that are described based on the causal relationship between the decision to perform an action and the past and present state of its beliefs. In Brahms we represent activities to take a certain amount of time.

However, an agent cannot always apply all its available activities, given the agent's cognitive state. Each activity is therefore associated with a *conditional statement* or *constraint*, representing a condition/activity pair, most of the time referred to as a *rule*. If the conditions of a rule are believed, then the associated activities are performed. In Brahms, such rules are called [workframes](#). Workframes are situated-action rules.

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

A workframe defines an activity (or activities) that an agent or object may perform. Workframes have conditions, called *preconditions*, that constrain when to carry out the activity. A workframe precondition tests a belief held by the agent executing the workframe. A workframe can also contain a detectable. *Detectables* describe circumstances (in the form of *fact-conditions* about the world) an agent might observe while executing the workframe. Detectable could, for instance, create an impasse to completing the activity (we will discuss detectables in more detail later in section 4.13). A typical workframe would be defined as follows, inside the body of an agent:

```
workframes:
    workframe wf_moveToRestaurant {
        repeat: true;
        variables:
        when
            [...]
        do
            [...]
```

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

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 in the world) 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 (facts).

As mentioned in the previous section, 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 and the beliefs do not trigger any workframes. Workframes for objects are only triggered by facts in the world.

Workframes can also be associated with objects. In this case workframes satisfy their preconditions with *facts* rather than with beliefs. Workframes for objects are inherited from object classes as workframes for agents are inherited from groups.

On the other side, having two or more agents with different workframes, performing the same activity, can represent individual differences. Individual differences can also be modeled by giving different agents the same workframes but different beliefs about the world. Activities are constrained on their activation by the preconditions that are associated with the workframe it is in. For example, activities may have preferential start times, as expressed in the preconditions, which may refer to the time in hours, minutes, seconds, day of the year, and/or day of the week.

Repeat

A workframe can be performed one or more times depending on the value of the 'repeat' attribute. 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 different ways. You can have the simulation engine determine the priority of the workframe (in that case the priority will 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); or you can set the priority of the workframe manually by setting a priority value with the priority attribute.

This priority value will cause the simulation engine to use this value instead of the deduced priority value.

4.7.3.2 ACTIVITIES

As mentioned earlier, an *activity* is an abstraction of real-life actions that help accomplish a task. A model of an agent's activities describes what the agent actually does over time (i.e. its actions) based on decision-points that are described based on the causal relationship between the decision to perform an action and the past and present state of its beliefs. In Brahms we represent activities to take a certain amount of time.

There are several types of activities. The simpler ones are primitive activities. 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. In the next sections we will study other activities, such as communicate, create object, as well as composite activities (that is, activities composed by other activities). In this section we will just consider one particular form of activity – the move activity – because it will come immediately useful for our Atm model.

Move activity

The primitive *move* activities trigger an agent or object to move to a location if not yet located there. For this activity type, the modeler defines the goal-location, such as the name of an *area* in the geography model, or a variable referring to an area. In moving, an agent or object may act as a container for another agent or object that is carried along. For example, a car-object may carry an agent, and then move to a new location. To simulate this, the modeler links the carrier and the carried with the built-in *contains* relation, before the move activity is executed (about relations see section 4.8). This is done with a consequence that asserts the relation, and then negates the relation with another consequence when the trip is completed, and the carrier “drops” the carried object or agent in the new location.

When a primitive move activity is executed, and the goal-location is different from the agent's or object's current location, the agent or object will start moving to the goal location. The simulation engine finds a path between the locations and gets or computes the distance. It is possible, however, to define the duration of the activity and thus avoid the need to define a geography model with travel paths. The engine calculates the duration of the trip and uses it to set the duration of the primitive move activity. When the agent or object reaches the new location, a new fact and belief are created stating that it is there. The agents currently at the new location detect the agent or object and will therefore get a belief about its location. A newly arrived agent will also detect the other agents and objects in the new location. The agent or object then continues with the workframe.

Brahms can handle interruptions that cause the location of an agent to change. Work that has to be done at a specific location may be interrupted and the agent may then move to another location to do work of a higher priority. When the higher-priority work is completed, before the agent resumes the interrupted work, the agent returns to the location where the agent has to do the work.

Declaration and reference

All activities have to be declared in the activities section of either a group, an agent, a class, an object, or a composite-activity. A typical declaration could be as follow (see more in the Tutorial sub section below):

```
activities:
move moveToRestaurant() {
  [Body of the activity]
}
```

The declared activities can then be referenced in the workframes defined for the group, agent, class or object, as we will soon see in the tutorial section:

```
workframes:
  workframe wf_moveToRestaurant {
    repeat: true;
    variables:
    when
      [...]
    do
      { [activity]
        [...]
      }
  }
```

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

Printed on: This is an uncontrolled copy when printed.

3/31/11 3:08 PM Refer to the NX Brahms location for the latest version.

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.

As mentioned before, primitive activities are atomic actions, and a small number of primitive activities are defined to have built-in semantics that is implemented in the Brahms engine. These predefined primitive activities exist to communicate beliefs, create runtime objects, and travel to a location.

4.7.3.3 PRECONDITIONS

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.

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:

```
carl.color = ?
```

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

```
carl.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 $O_1 \text{ rel}$ 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 $O_1 \text{ rel} ?$, 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 $O_1A_1 \text{ rel}$ 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 $O_2 \text{ rel}$ 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 unexpected:

```
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. We will discuss variables in section 4.10.

4.7.3.4 CONSEQUENCES

Consequences are statements that are inside a workframe's body. They can be situated before or after activities. *Consequences* are facts or beliefs or both that may be asserted when a workframe is executed. They exist so a modeler may model the results of the activities in a workframe. A consequence is formally like a condition and defines the facts or beliefs that will be created or changed, when executed. The property *fact-certainty* is the probability that the fact will be changed or created; the default value is 100%. The property *belief-certainty* is the probability (with also a default value of 100%) that the belief will be changed or created, conditional on the fact being true. That is, if the fact-certainty and the belief-certainty are each 50%, then 1 in 2 times the fact will be created and 1 in 4 times the belief will be created. If the fact-certainty is zero, then no fact will be created but the belief-certainty determines how often a belief is created.

A consequence is a logical statement for concluding/asserting new beliefs for an agent or object and/or facts in the world.

Fact certainty

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

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.

4.7.4 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_wfr_stx.htm, http://www.agentisolutions.com/documentation/language/ls_pac_stx.htm, http://www.agentisolutions.com/documentation/language/ls_mov_stx.htm, and pages linked from there.

4.7.5 TUTORIAL

So: the first action we will make our Alex_Agent complete in our Atm scenario will simply be going to one of the two Restaurants in town and eating there. This will be a very simplified version of what happens in real life. We do not have yet a Restaurant object - but you might remember that we have already defined a couple of restaurant locations (that extend the `building` area definition). One of such location is `Telegraph_Av_2405`, the street address of the restaurant. Let's say that we know that every student will go to that specific restaurant – not only Alex – to keep things simple for the moment. And let's assume that, just by being there, the student will feel automatically less hungry (basically, we will bypass modelling the 'eating' step). So, we might write something like this: inside the Student body declaration, under the `workframes:` tag, let's define a workframe called 'moveToRestaurant':

```
workframes:
  workframe wf_moveToRestaurant {
    repeat: true;
    variables:
    when
      (knownval(current.howHungry > 2.00) and
       knownval(current.location != Telegraph_Av_2405))
    do
      {moveToRestaurant()};
      conclude((current.howHungry = current.howHungry -
                5.00), bc:100, fc:100);
    }
  }
```

The first line of the workframe simply declares the unique name of the workframe. We do not bother – for the moment – with the variables declaration. Repeat is set to true so that this workframe might be repeated, if needed. The repeat attribute for workframes allows workframes with the exact same variable binding to be executed repeatedly. When a workframe is instantiated with a specific binding that workframe instantiation (wfi) is executed. When the wfi has been completed the engine will determine if the bindings are still valid (if the preconditions still hold true for the same beliefs). If that is the case and if repeat is set to true, the engine will make a new wfi with the exact same bindings available. If repeat is set to false then that specific workframe will never be executed again with that specific set of bindings.¹⁷

¹⁷ The only exception to this rule is with workframes specified within a composite activity (that we will consider in section 4.11). While a composite activity is active the engine keeps track of the executed bindings for workframes, but when the composite activity ends, all of these bindings are reset. This is only relevant in case repeat is set to false.

The `when:` tag is very important. It states under which conditions the workframe has to be triggered. When the [conditions](#) stated are met (precisely those values, cause we are using the `knownval` statement) the [conclusions](#) stated after the 'do' tag will be fired, in the order that they are written. In this case, we need that the current student/agent being considered be hungrier than '2.00' (whatever that means; and that he/she is not already at the restaurant. The conclusion will make him less hungry by a constant (equal to 5) after he has been to the restaurant. (Aside note: you can of course alter this and other values in this scenario – how much money the agents has or gets from the bank, how expensive the restaurants are, etc. In fact, you are encouraged to do so and see how your simulation will change!).

Note that `knownval` checks for a belief that the agent has about that concept, *and* also checks to see whether that belief is true; hence `kknownval (...)` is by default a check for 'is it true?'. Note, moreover, that preconditions are always checked from the point of view of the agent/group/object or class from inside which they are executed. A `when` condition about a belief that the agent does not even have will simply be skipped. Hence, beliefs for agents are absolutely crucial in their workframes (and thoughtframes). The Compiler will throw no error (because no error is there to be found), but all the consequences after the `do` tag will be ignored. Note that in order to make an agent do something when it does not have a belief about something (i.e. not even an uncertain belief), you can write: `unknown(current.attribute)`. On the other side, the situation where an agent has an 'uncertain' belief about something (for example, an agent who has forgotten how hungry she is, but not about the fact that she has an 'hungriness' attribute), might be represented with the following format: `(current.howHungry != anynumber)`. A `known` precondition applied to an agent with such belief and referred to the attributed `howHungry` will evaluate to true (and, of course, the `knownval` precondition will evaluate to false).

Note how the `not` operator works when applied to beliefs about attributes and relations:

```
not(current contains object)
```

evaluates to false if the agent has a belief that she contains the object. The same thing would happen by evaluating a belief under a different syntactical form: `knownval(current contains object is false)`. On the other side, if the agent does not have a belief, `not(current contains object)` will evaluate to true simply because there is no belief; but `knownval(current contains object is false)` will evaluate to false for the same reason. With relations, most of the times `not` can be safely used rather than the `is false` statement.

Let's go back to the tutorial. If the conditions are met, the agent will first try to perform the [activity](#) `moveToRestaurant()`; then, the agent will process some conclusions about beliefs and facts of the world (in particular, about the fact/belief that he/she is less hungry than before). Note that activities must always be called from a workframe. Of course, you can re-use the same activity and make it be called by several workframes with different parameters. Note, however, that `moveToRestaurant()` is different from `wf_moveToRestaurant`. The latter is a workframe; the former is an activity, written similarly to a method in Java, where inside the parenthesis parameters can be passed to make the activity specific (in this case we are not passing any parameter, but we still need to use the parenthesis).

So, workframes check the facts and beliefs in the world, and when their conditions are met, they trigger activities and conclusions. But where are the activities explicitly written? Who tells the agent what `moveToRestaurant` really means? We need to add some more code for this purpose (aside: activities can be interrupted when conditions are met that trigger new activities/workframes; however, when we use *composite* activities things become much trickier: cf. 4.11). We will add this code right after the `activities` tag, outside the workframe:

```
activities:
    move moveToRestaurant() {
        location: Telegraph_Av_2405;
    }
```

This is a simple move activity. We did not have to specify its duration because the geography of this scenario has already been described in the `AtmGeography` file where we have declared a path for the movement we are considering now. We could have overridden those timings by imposing, for example,

```
move moveToRestaurant() {
    random: true;
    min_duration: 50;
    max_duration: 150;
    location: Telegraph_Av_2405;
}
```

Now, try to compile this code. You can do it from the `Compile Model` command inside the `Composer`, or you can do it manually as explained in Chapter 3.¹⁸ Remember in this case that you must use the `AtmModel.b` file, which imports all the other files in the `Atm` project/folder.

¹⁸ The command should look like:

If everything worked out fine, you should now be able to see a message (in one of the Composer's windows or in the command line interface) telling you that the model has been successfully built. You might even want to open one of the xml files that have been produced in the Atm folder. For example, there will be an Alex_Agent.xml file that will look more or less like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE AGENT SYSTEM "file:///C:\Program Files\Brahms\AgentEnvironment
1.0\DTD\agent.dtd">
<!-- Generated at Sun Apr 15 18:23:02 PDT 2001 -->

<AGENT name="projects.Atm.Alex_Agent" display="Alex_Agent"
location="projects.Atm.SouthHall">
  <MEMBEROF ref="projects.Atm.Student" />
  <BELIEFS>
    <BELIEF>
      <OAV lObjRef="current" lObjType="Current"
attRef="projects.Atm.Student.howHungry" evalOp="eq" value="15.0"
valueType="double"/>
    </BELIEF>
    <BELIEF>
      <OAV lObjRef="current" lObjType="Current"
attRef="projects.Atm.Student.male" evalOp="eq" value="true"
valueType="boolean"/>
    </BELIEF>
    <BELIEF>
      <OAV lObjRef="current" lObjType="Current"
attRef="projects.Atm.Student.preferredCashOut" evalOp="eq" value="8.0"
valueType="double"/>
    </BELIEF>
  </BELIEFS>
  <FACTS>
    <FACT>
      <OAV lObjRef="current" lObjType="Current"
attRef="projects.Atm.Student.howHungry" evalOp="eq" value="15.0"
valueType="double"/>
    </FACT>
    <FACT>
```

```
C:\Program Files\Brahms\AgentEnvironment\bc.bat
c:\brahms\Projects\AtmModel\final_source -src
c:\brahms\Projects\AtmModel\final_source\gov\nasa\arc\brahms\atm\AtmModel.b
```

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

```

        <OAV lObjRef="current" lObjType="Current"
attRef="projects.Atm.Student.male" evalOp="eq" value="true"
valueType="boolean"/>
    </FACT>
</FACT>
        <OAV lObjRef="current" lObjType="Current"
attRef="projects.Atm.Student.preferredCashOut" evalOp="eq" value="8.0"
valueType="double"/>
    </FACT>
</FACTS>
</AGENT>

```

Now, you are ready for the big step! Try running your simulation for the first time, the way it was discussed in Chapter 3.

If everything goes fine, you will see something like:

```

java
Auto
info - Loaded concept 'projects.atm.BOA_to_from_StH'
info - Loaded concept 'projects.atm.SpH_to_from_BOA'
info - Loaded concept 'projects.atm.BOA_to_from_RB'
info - Loaded concept 'projects.atm.BB_to_from_WF'
info - Loaded concept 'projects.atm.anylocation'
info - Loaded concept 'brahms.base.City'
info - Loaded concept 'projects.atm.SpH_to_from_RB'
info - Loaded concept 'projects.atm.Telegraph_Av_2405'
info - Loaded model 'projects.atm.AtmModel'
info - Starting virtual machine
info - Starting scheduler
info - Starting engine for 'projects.atm.Alex_Agent'
info - Stopping virtual machine
info - Stopping scheduler
info - Stopping engine for 'projects.atm.Alex_Agent'
info - Stopped event notifier
info - Stopping FileAgentViewerService 'FileAgentViewerSvc_1'
info - Stopping TestExternalService 'TestExternalSvc_1'
info - Stopping logger
Virtual Machine's running time: 0:0:18.900
Model Loader's running time: 0:0:8.520
Model Initializer's running time: 0:0:1.810
Model Simulator's running time: 0:0:2.310
Model Exporter's running time: 0:0:0.0
Press any key to continue_

```

Figure 12. The Simulation Engine (Virtual machine)

This means that the Virtual Machine has loaded all concepts, started the engine, and run successfully the simulation. If the simulation never halts, or never even starts because a java exception is thrown, there must be again an error in the code. Go back and check it (we will discuss errors in detail over the next sections).

On the other side, let's assume that you have made it till now. You might like to know that a text file has been produced and saved (by default), in the Brahms/AgentEnvironment/Databases directory, with a unique name that includes the date and time of the simulation run (for example, it will look something like `AtmModel_20010415_134559.txt`). Go and look for that file. It will be mostly incomprehensible, now, but for future reference it is good to know it is there. It is the history of your simulation that must be passed to the Agent Viewer for parsing, in order to be transformed into a MySQL database file that the Agent Viewer can then display as a 2-dimensional graph.

But let's go with order. The text file will look something like this:

```
precondition|PRE1|knownval(current.howHungry > 2.0)
precondition|PRE2|knownval(current.location != projects.Atm.Telegraph_Av_2405)
consequence|CON1|conclude((current.howHungry = current.howHungry - 5.0));
workframe|WFR1|wf_moveToRestaurant|PRE1,PRE2|MOV1,MOVE,CON1,CONSEQUENCE-MESSAGE
group|GRP2|Student||AGT1|WFR1
attribute|ATT1|howHungry
attribute|ATT3|preferredCashOut
attribute|ATT2|male
[...]
belief-he|BC0|belief: (projects.Atm.Alex_Agent.howHungry =
15.0)|NEW|COMPLETED|0|0|-1|none|HL2|BC0
belief-context|BC0|OBJ-ATT-
PAIR|AGT1|AGENT|ATT1|EQ|VALUE|NONE|NONE|NONE|15.0|belief:
(projects.Atm.Alex_Agent.howHungry = 15.0)|AGT1|INITIAL
belief-he|BC1|belief: (projects.Atm.Alex_Agent.male = true)|NEW|COMPLETED|0|0|-
1|none|HL2|BC1
belief-context|BC1|OBJ-ATT-
PAIR|AGT1|AGENT|ATT2|EQ|VALUE|NONE|NONE|NONE|true|belief:
(projects.Atm.Alex_Agent.male = true)|AGT1|INITIAL
belief-he|BC2|belief: (projects.Atm.Alex_Agent.preferredCashOut =
8.0)|NEW|COMPLETED|0|0|-1|none|HL2|BC2
```

If you can't read it, the Agent Viewer can! Open the application Agent Viewer, and click on the top left menu button, called 'Parse New File' (or just use the window menu). Look for the history file when the selection window opens, and once you find it, select it. The Agent Viewer will use an existing database sample file as a template to create the new database file with the history of the simulation (do not delete or remove or change the database file called `brahms.mdb` in the Database folder!!). It will create a new database file, ask you to save it (with a name such as `AtmModel_20010415_134559.mdb`), and then it will actually start parsing the file. If the parsing completes without sending out error messages, click on the second leftmost menu button on the top, to *open* the newly created file. Agent Viewer will by default try to open the most recently created database file, so you will simply have to click on yes in the selection window.

Does the file load? You will see a screen like this:

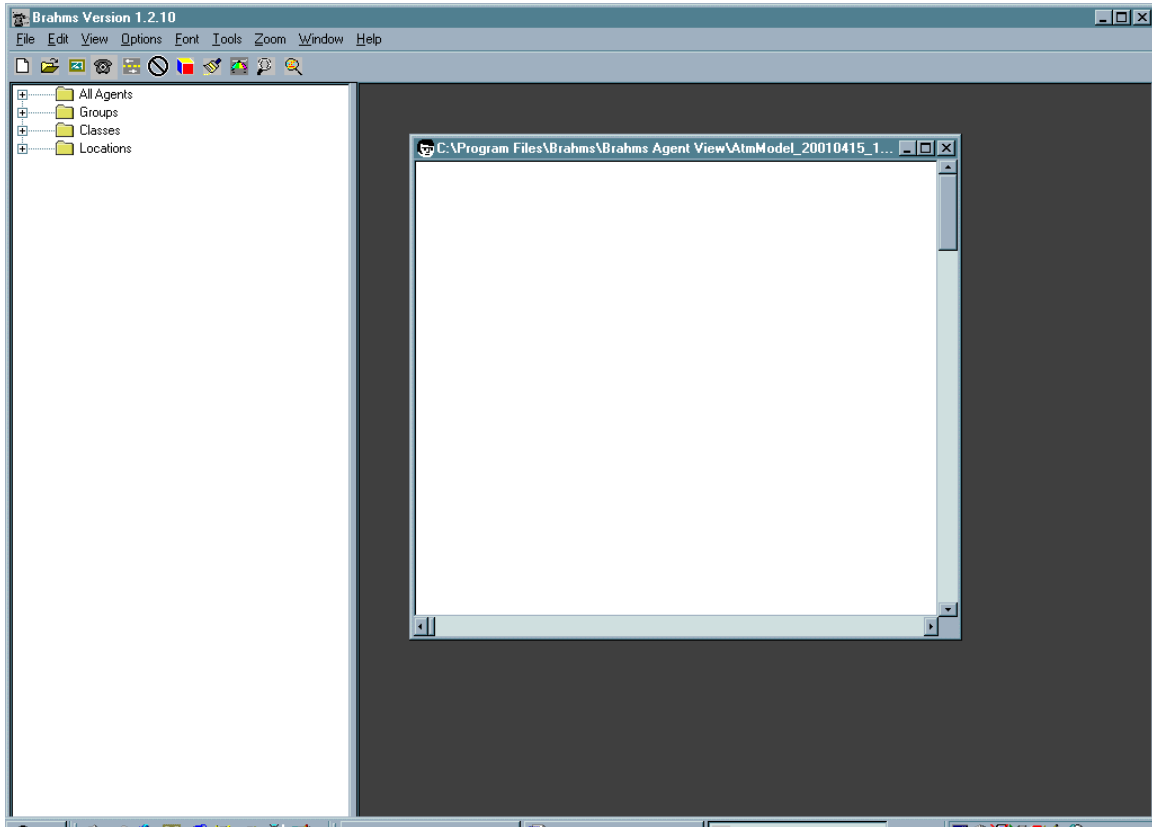


Figure 13. The Agent Viewer and the Atm Scenario

The elements on the left are the components of your simulation. Click on the agent folder and then once on Alex_Agent. Then zoom in or out in the picture (right menu buttons on the top) until you see something like this:

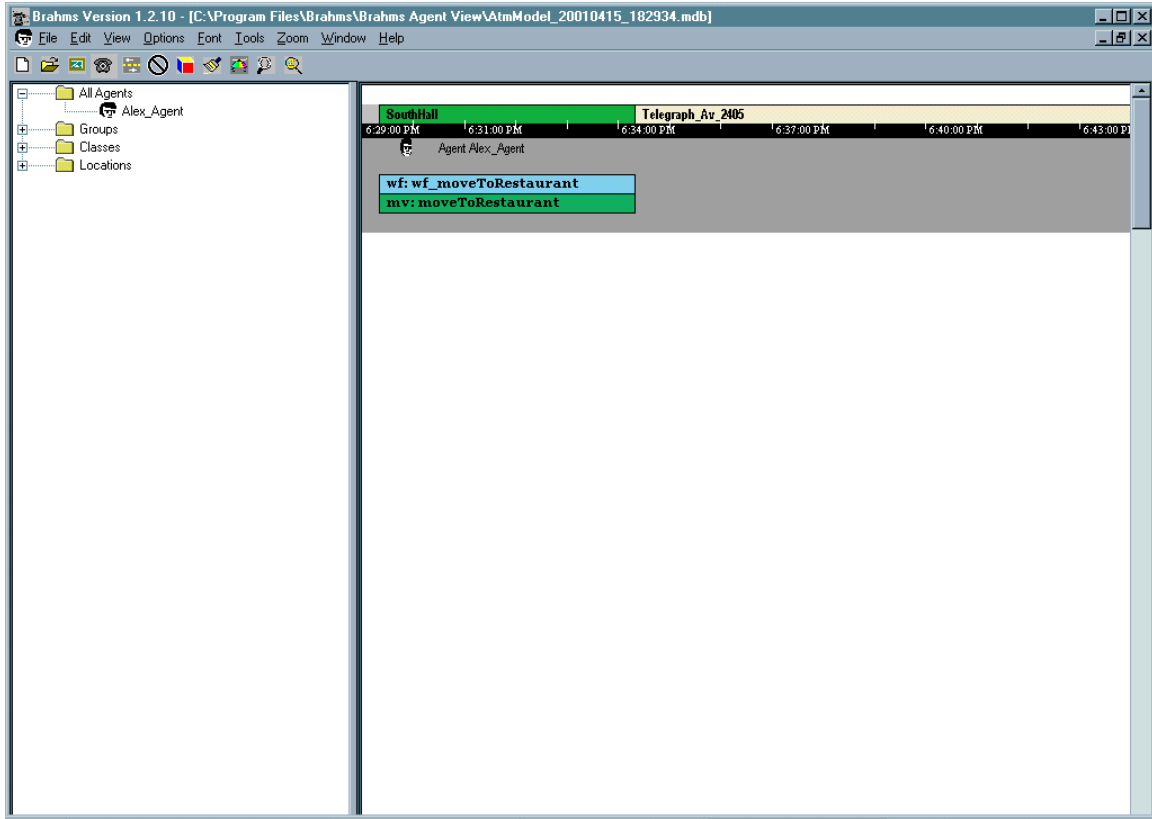


Figure 14. The First Activity in the Agent Viewer

Congratulations!! You have just completed and verified your first Brahms simulation. Alex_Agent starts, as he should, in South Hall, and then moves to Telegraph Avenue when he is hungry (and he is very hungry already at the beginning of the simulation). Play around with the components of the Agent Viewer and of your model. Take some time to check the various menus. You will see that the Agent Viewer can show things like communications, thoughtframes (cfr. section 4.9) and much more. For example, click once with the left mouse button on the workframe wf_moveToRestaurant. You will access the Explanation Facility:

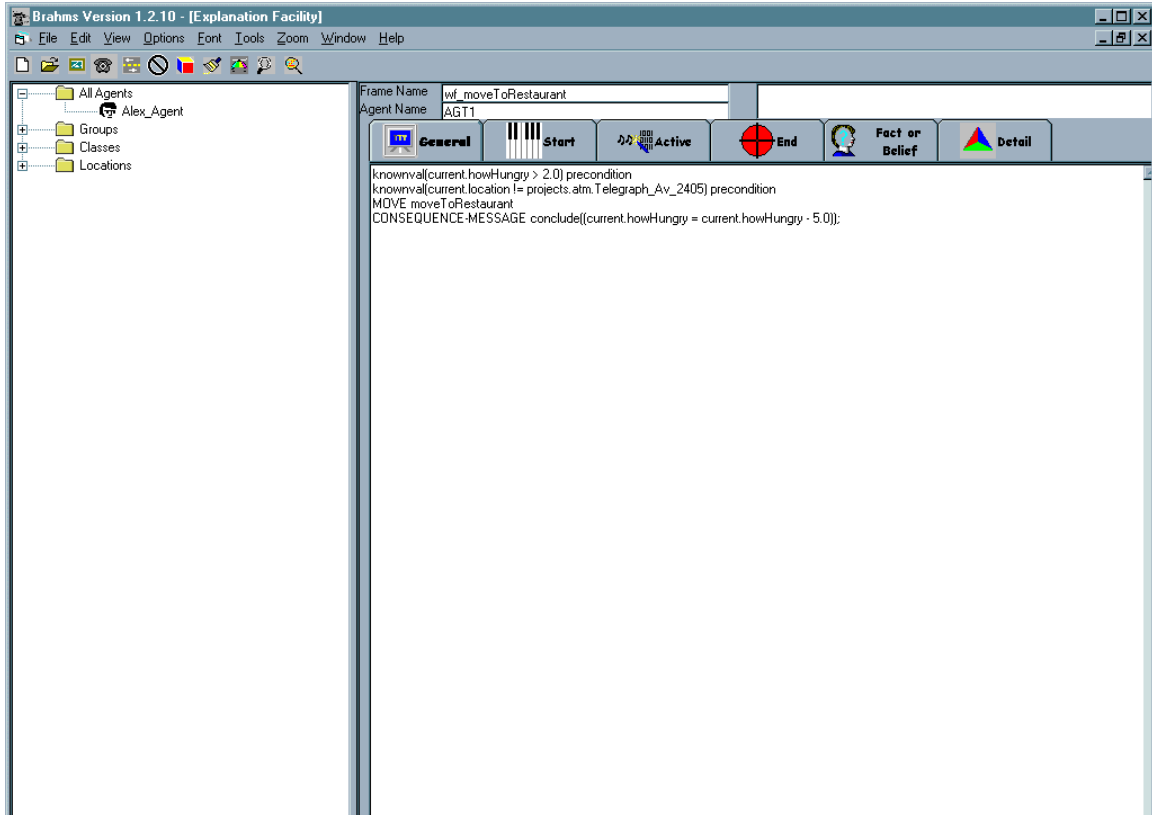


Figure 15. The Explanation Facility in the Agent Viewer

From the explanation facility you can get a picture of (almost) anything that is going on in the simulation: why workframes or activities are triggered ('General' tag), with some detail ('Start' tag); what happens when (for example) the workframe is triggered ('Active' and 'End' tags); and all of the beliefs and facts that take place or change in the world of your simulation ('Fact or Belief' tag). The explanation facility will be a powerful tool to debug and verify your simulation. Only in very extreme cases you will need to use anything else (for example, the raw text file with you simulation's history).

Ok, before with go on with something else, this is a summary of the steps necessary to use the Agent Viewer:

1. Open the application Composer and switch to the Agent Viewer view.
2. Click on the menu, File -> Agent Viewer -> New Database... .
3. Look for the history file when the selection window opens, and once you find it, select it and press Open. Wait while the Agent Viewer parses the file.

4. When parsing is complete, the database should automatically be opened and its results displayed, if not click on the menu File -> Agent Viewer -> Open Database... to *open* the newly created database.

A few comments about activities and workframes.

First, preconditions are not compulsory – we can make an agent always do something by eliminating the when condition and setting repeat to true.

Second, activities are not really simultaneous: if an agent walks, she cannot study: her time is devoted to one only activity. It is possible, however, to model an activity like: ‘studying while walking’; or to write a composite activity (cf. 4.11) and let more internal workframes be triggered at the same moment by playing around with priorities, preconditions and activity times. There are also special activities called “composite activities”, but we will deal with those later in the Tutorial (cf. 4.11). For the moment, it might suffice to know the following: if you have two workframes with preconditions that are both satisfied by the beliefs of an agent, Brahms will create two workframe instantiations. Both will be marked as available, but the agent will only work on one of them. The workframe being worked on depends on the priority of the workframe, the highest priority workframe is selected. If both frames have the same priority the frame that is first in the list of available frames will be selected. Now, even if Brahms currently does not support true multi-tasking, it does have a subsumption architecture that can create some form of multi-tasking although not in the sense of operating systems by using composite activities. For example, if I were to have a workframe A with a composite activity doWork and in that composite activity doWork I have a workframe B:

```
composite_activity doWork() {
  workframes:
    workframe B {
      when(knownval(inbox.numOrders > 0))
      do {
        getOrderFromInbox();
      }
    }
    workframe A {
      when(knownval(agt.available = true))
      do {
        doWork();
      }
    }
}
```

Then, if the agent believes that it is available and that the number of orders in the inbox is larger than 0, firstly workframe A is made available and active for the agent to work on, and as soon as the agent starts doWork it will also have workframe B available and will work on that instantiation as well.

Third, you might be interested in knowing whether there are `if...then...else` statements in Brahms. The quick answer is: no. The explanation is that you can do exactly the same by writing, for example, two or three thoughtframes. Declarative structures such as `if...then...else` are not welcome in Brahms, because they are built to be checked in specific orders, while Brahms works on rules, and all the preconditions of all rules are always being checked. This is an important characteristic of rule-based programming.

Fourth, note that any workframe in a composite activity has access to the activities defined within that same composite activity and to the activities defined at the same level as the composite activity and the activities higher in the activity hierarchy above the composite activity.

Sixth, Brahms does not yet support OR statements in workframes. In order to accomplish the same results of OR statements, you would have to duplicate the body of your workframe into a new workframe and use the OR'd preconditions.

```
when(  
  knownval(car1.color = red) or  
  knownval(car1.color = blue)) {  
do {  
  someActivity();  
}
```

would have to be rewritten as two workframes with the bodies:

```
wf1:  
  
  when(knownval(car1.color = red))  
  do {  
    someActivity();  
  }  
  
wf2:  
  
  when(knownval(car1.color = blue))  
  do {  
    someActivity();  
  }
```

Lastly, no nested expressions in the preconditions are allowed, and only simple ones in the conclusions are.

But let's go back to our simulation. Why does the agent not do anything else after going to the restaurant? Well, first of all, this statement is not entirely correct: if you check the beliefs and the facts of the world from the 'Fact or Belief' tag, you will see that your agent is now less hungry than before (`howHungry = 10.00`).

After this, the agent has nothing to do: one of the two preconditions for triggering the `moveToRestaurant` activity is not met – the agent is already at the Restaurant! So, let's create some more movement. Let's say that whenever the agent goes to the restaurant, after eating he moves back to South Hall to study. Let's add an activity 'goToSouthHall':

```
move moveToSouthHall() {
```

```
        location: SouthHall;  
    }  
}
```

Remember, we do not have to specify the path and the duration of this move action because we have done so already in the geography file. Then, modify the workframe wf_moveToRestaurant as follows:

```
workframe wf_moveToRestaurant {  
    repeat: true;  
    variables:  
    when  
        (knownval(current.howHungry > 2.00) and  
         knownval(current.location != Telegraph_Av_2405))  
    do {  
        moveToRestaurant();  
        conclude((current.howHungry = current.howHungry - 5.00),  
                bc:100, fc:100);  
        moveToSouthHall();  
    }  
}
```

and compile/run/parse your simulation as before to see what happens:

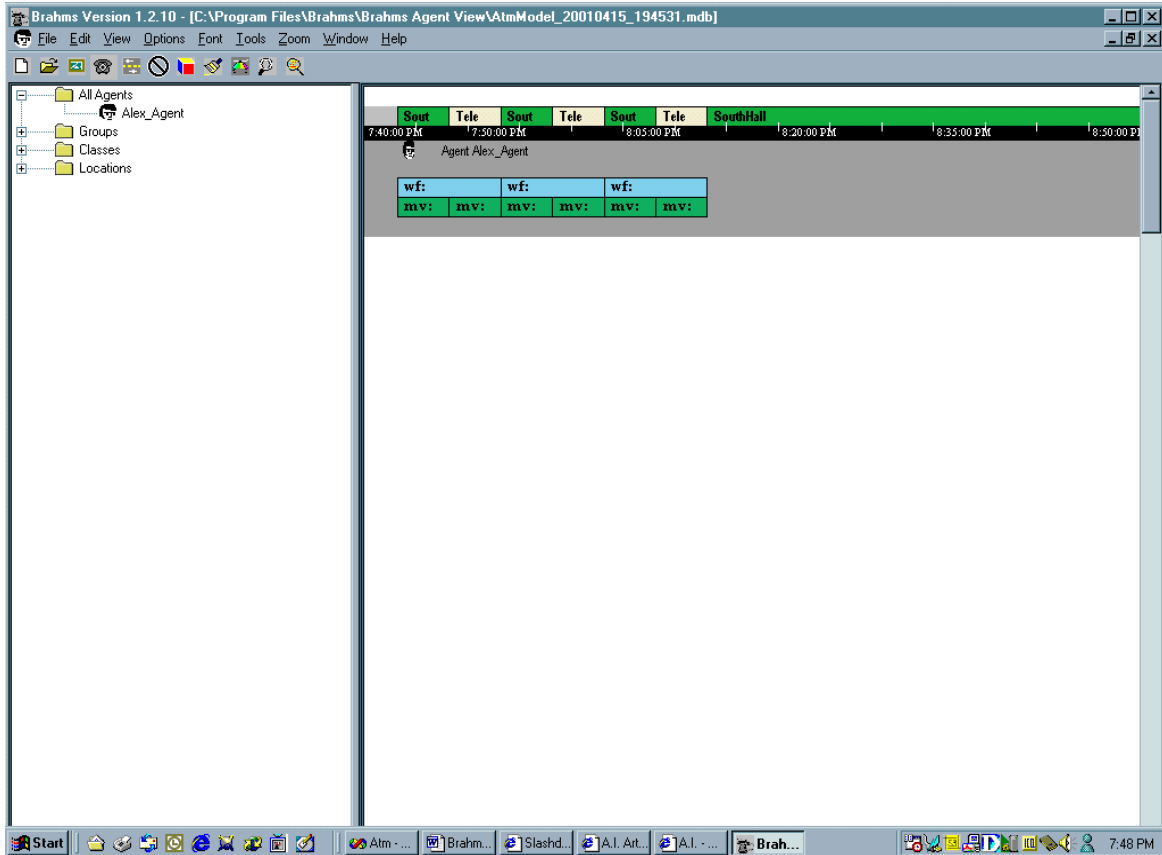


Figure 16. More Complex Activities

As you can see, now the agents move back and forth South Hall and the Restaurant. Note that order *is* important. If you have a conclude statement after a call to a move action (or any other action), the conclusion is drawn *after* such action has been completed. The order that the Compiler follows is left to right, top to bottom. The same rule is applied to different workframes inside the body of – say – an agent. Keep this simple rule in mind when unexpected things happen: it could well be a problem of priorities between workframes (we will discuss priorities in section 4.13), an issue with the order of the workframes and the preconditions that must be satisfied to trigger them, or just a matter of the order you are triggering activities and conclusions within each workframe. For example, processing beliefs could make some of the available frames unavailable and could make unavailable frames become available (more on this in section 4.11).

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

Note, moreover, that if the Agent Viewer crashes when opening a database it has parsed correctly, it might be because no activity is really running. Furthermore, agents need a location and a performed activity to be shown in the Agent Viewer; and workframes that only triggers conclusions but do not trigger activities will not show up in the Agent Viewer.

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

4.8 LESSON VI: CLASSES, OBJECTS AND RELATIONS

4.8.1 INTRODUCTION

This chapter will teach you how to create objects as instances of classes in a Brahms model, and will discuss relations between objects and agents.

4.8.2 TASK

Create the objects that are part of the simulation: the Atm(s), the campanile (to signal the flow of time), the bank, the cash, the bank account, the restaurant diner, etc., together with their various attributes and their relations. Then, give the simulation a little more 'movement': Alex_Agent, as before, is hungry and needs food; therefore he goes to the restaurant, where his 'hungriness level' is automatically decreased. The diner is an actual object with a specific location and some attributes (let's imagine all restaurants in the simulation have fixed-price menus, and one such attribute is the cost of the restaurant's fixed menu). In addition, now Alex needs money to pay for the restaurant: if he does not have enough money, Alex will have to go to the Atm machine. While in that location, he will automatically get our more cash. This will happen only once (it will be replicated and extended in the next lessons...)

4.8.3 DESCRIPTION

4.8.3.1 OBJECTS

An [object](#) 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. In Brahms there is a difference between animate—intentional—objects (which we refer to as agents) and inanimate—unintentional—objects (which we refer to as *objects*). In all other agent-languages there is only one type of object, namely an intentional agent. In Brahms, agents are intentional.

However, we also want to be able to describe artifacts in the real world as action-oriented systems, but unintentional at the same time. We describe such an artifact as an *object*. An example of an object in Brahms is a fax machine. If we want to describe the behavior of a fax machine, we could argue that we could describe a fax machine as an intentional agent. However, in the real world we would never ascribe intention to the actions of a fax machine. A fax machine mainly reacts to facts in the world; such as a person pushing the start button on the fax machine that makes the fax machine start faxing the document. Since in Brahms we are interested in describing the world with its animate and inanimate objects, we want the capability to make a difference between an intentional object (an agent) like a human and an unintentional object (an object) like a fax machine. There might be occasions when the *intentional stance* is appropriate for objects. When this is the case, we might decide to represent a machine as an agent. For example, in the Atm scenario the Atm machines and Bank computers might be modeled as agents (there will be differences regarding detectables and what the objects act upon, and we will analyze these differences soon).

To summarize, an object in Brahms is a construct that generally represents an artifact. The key properties of objects are facts, workframes, and activities, which together represent the state and causal behaviors of objects. Some objects may have internal states, such as information in a computer, that are modeled as beliefs. Other artifact states —such as the fact that a phone is off hook— are facts about the world.

On the other side, 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.

4.8.3.2 CLASSES

[Classes](#) in Brahms represent an abstraction of one or more object instances. The concept of a 'class' in Brahms is similar to the concept of a template or class in object-oriented programming. It defines the activities and workframes, initial-facts and initial-beliefs for instances of that class (objects). Brahms allows for multiple inheritances for objects (note that objects currently do not inherit the cost, time-unit and resource values – they must be specified in the object's body). Classes are used to define inanimate artifacts, such as phones, faxes, computer systems, pieces of paper, etc.

In a model, a hierarchy of classes can be built by defining the class inheritance. A class can inherit from more than 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.

Note that a conceptual object class defines a type of conceptual objects used in a model.

4.8.3.3 ELEMENTS OF OBJECTS AND CLASSES

A Brahms object has all of the elements that an agent has, plus two additional elements; *conceptual object membership* and *resource*. Furthermore, instead of having a group membership relation with groups, an object can have *class-inheritance* relationships with classes.

A Brahms object has the following extra elements:

Class-inheritance: An object can be an instance of one or more classes. In case constructs with the same name are encountered in the inheritance path, always the most specific construct will be used. For example, a workframe defined for the object has precedence over a workframe with the same name defined in one of the classes of which the object is an instance.

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 and workflow.

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).

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.

4.8.3.4 RELATIONS

[Relations](#) are used to represent the connections 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.

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`. The distinction is exactly the same as that for attributes – hence we do not repeat it again here.¹⁹

4.8.4 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_cls_stx.htm, http://www.agentisolutions.com/documentation/language/ls_obj_stx.htm, http://www.agentisolutions.com/documentation/language/ls_rel_stx.htm, and pages linked from there.

4.8.5 TUTORIAL

Let's start creating now the objects that are important in our Atm scenario. We might forget some, and we will add more on the way.

We certainly need banks (or, at least, bank central computers) that keep information about bank accounts and communicate with Atms when agents visit them for cash. Hence, we also need bankcards, to be used at Atms, and we need a formalism to model cash itself. We will also need diners where the agents eat – such diners might be simply modeled as locations (remember that locations can have attributes, such as the cost of the diner and its name), or we might model diners as actual entities (for example objects) at a *specific* location (as you have already understood, there are several ways to model any scenario and you always have to find the one which is the more appropriate, the one that balances complexity with realism).

So, let's start creating some new classes! We quite certainly need Atms. Thus, let's create a new file `Atm.b`. As usual, remember to define the package at its beginning:

```
package gov.nasa.arc.brahms.atm;
class Atm {
    display: "Atm";
    cost: 0.0;
```

¹⁹ As mentioned in a previous note, however, both relation and attribute scopes are currently *not yet* implemented in the language. This implies that attribute/relation scope definitions will be ignored by the Compiler, and all attributes/relation will be treated (for the time being) as public. This is why in the code examples presented in the text all attributes and relations are treated as 'public'.

```
resource: false;
attributes:
    public int currentAccountCode;
    public int currentAccountPin;
    public boolean checkedAccountCode;
    public boolean checkedAccountPin;
relations:
    public Bank ownedbyBank;

initial_facts:
    (current.checkedAccountCode = false);
    (current.checkedAccountPin = false);
}
```

Let's comment a little the code above. Apart from some differences that you should be able to easily spot, the structure looks very similar to that of a group. We have already coded some initial facts inside the class definition (rather than an object, specific instance of that class) because we simply want all instances of Atms to start that same way.

Note that [objects](#) do *not* need a location to exist and act in the world. On the other side, what does it mean that objects react on facts only? Objects can have beliefs, and can even have thoughtframes where they conclude new beliefs. However, they act on facts only, in the sense that the preconditions of an object's workframe must be satisfied by facts rather than beliefs. In fact, objects act on fact *regardless* of the beliefs they have (or do not have)!

In addition, in the code above we have also used for the first time a relation: ownedbyBank. This relation links a specific Atm to a specific Bank (which Atm and which Bank is specified at the object level). So, let's now also define a Bank class:

```
package gov.nasa.arc.brahms.atm;
class Bank {
    display: "Bank";
    cost: 0.0;
    resource: false;

    attributes:
        public string name;
        public int receivedAccountPin;
        public int receivedAccountCode;
```

```
        relations:

        initial_facts:

        activities:
        workframes:
    }
```

Of course, as the tutorial proceeds, we will add many other attributes, as well as activities and workframes. For the moment, we can note that we do not have to model an inverse relation Bank owns Atm inside the Bank class – one relation will be enough to make our model work, as we will see soon when we will be dealing with variables (section 4.10). On the other side, we should start filling in the details of these classes and populate the model with their instances. Let's imagine that there are two Banks in our model of Berkeley: Bank of America and Wells Fargo. Each has one Atm in town. So, we can write for the Atms:

```
package gov.nasa.arc.brahms.atm;
object Boa_At看 instanceof Atm {

    location: Telegraph_Av_113;

    initial_facts:
        (current ownedbyBank Boa_Bank);
}
```

and

```
package gov.nasa.arc.brahms.atm;
object WF_At看 instanceof Atm {

    location: Bancroft_Av_77;

    initial_facts:
        (current ownedbyBank WF_Bank);
}
```

and, for the Banks:

```
package gov.nasa.arc.brahms.atm;
object Boa_Bank instanceof Bank {
    display: "Boa_Bank";
}
```

```
        initial_facts:
            (current.name = BankofAmerica);
    }
```

and

```
package gov.nasa.arc.brahms.atm;
object WF_Bank instanceof Bank {
    display: "WF_Bank";

    initial_facts:
        (current.name = WellsFargo);
}
```

We then need to model the Accounts of the agents and their BankCards. To keep things very simple, a BankCard will keep stored its pin and its account code, that we can model as attributes. We can also model Cash as an object whose attribute 'balance' gives the amount of the cash currently carried by the agent. Finally, a bank Account will have some minimal attributes such as its code, its balance, and some [relations](#), such as what bank it has been opened with, or what agent it is owned by (which could be also modeled from inside the Student declaration). Armed with this information, we can create the following files: `Account.b`,

```
package gov.nasa.arc.brahms.atm;
class Account {
    display: "Account";
    cost: 0.0;
    resource: true;

    attributes:
        public double balance;
        public string typeof;
        public int code;
        public int pin;

    relations:
        public Bank openedWithBank;

    activities:
}
```

then `Cash.b`:


```
package gov.nasa.arc.brahms.atm;
class Cash {

    display: "Cash";
    cost: 0.0;
    resource: true;

    attributes:
        public double amount;

    activities:
}

```

and finally `BankCard.b`:

```
package gov.nasa.arc.brahms.atm;
class BankCard {

    display: "BankCard";
    cost: 0.0;
    resource: true;

    attributes:
        public int code;

    relations:
        public Account accesses;

    activities:
}

```

In a similar fashion, we start drawing the net of relations that link students to other objects in the Atm universe. The first we can think about are Banks, bank Accounts, and BankCards. Hence we write inside the body of the Student group:

```
relations:
    public Account hasAccount;
    public Cash hasCash;
    public BankCard hasBankCard;

```

Finally, we must create the additional files for the specific instances of these classes that belong to Alex_Agent: create therefore a file `Alex_Account.b`,

```
package gov.nasa.arc.brahms.atm;
object Alex_Account instanceof Account {
    display: "Alex_Account";

    initial_facts:
        (current.balance = 100.00);
        (current.typeof = checking);
        (current.code = 1212);
        (current.pin = 1111);
        (current.openedWithBank Boa_Bank);
}
```

then a file `Alex_Cash.b`:

```
package gov.nasa.arc.brahms.atm;
object Alex_Cash instanceof Cash {
    initial_facts:
        (current.amount = 8.00);
}
```

and finally a file `Alex_BankCard.b`:

```
package gov.nasa.arc.brahms.atm;
object Alex_BankCard instanceof BankCard {
    initial_facts:
        (Alex_BankCard.code = 1212);
        (current.accesses Alex_Account);
}
```

Of course, we must also modify the `Alex_Agent` file to reflect the new information. We must update the beliefs (and the facts, where necessary) as follows:

```
initial_beliefs:
    (current.howHungry = 15.00);
    (current.male = true);
    (current.preferredCashOut = 8.0);
    (current.contains Alex_Cash);
    (current.contains Alex_BankCard);
    (Alex_Account.balance = 100.00);
    (Alex_Account.code = 1212 );
```

```
(Alex_Account.pin = 1111);  
(Alex_Account.openedWithBank Boa_Bank);  
(Alex_Cash.amount = 13.00);  
(current.hasCash Alex_Cash);  
(current.hasBankCard Alex_BankCard);
```

We can spend a few words here about how to express conditions (preconditions and conclusions) about relations. The `is true` or `is false` statements are only used for relations. `(current.hasBankCard bkc is false)` is the right syntax to check whether the relation (fact of belief thereof) is false. But if we wanted to check whether this relation was actually true, we could just write: `current.hasBankCard bkc`. In other words, the `is true` form is not needed when checking or concluding conditions. One might ask what is the difference between `knownval((current.attribute = x))` and `knownval(current.attribute = x) is true`. Note that we are using the 'contains' relation: it is a relation built-in the language that is very useful when we want to move objects and agents together with what they are carrying.

The difference is that we can apply `is true` to relations (because, otherwise, it would be impossible to create a false belief; in other words, an agent always believes its own beliefs), but we do not use it for attributes. Therefore, when you want to express conclusions about relations, if the conclusion is true, you can simply conclude `(whoever.hasCard whatever)`; if false, you might instead conclude `(whoever.hasCard whatever is false)`. Finally, in preconditions, `knownval(whoever.hasCard whatever is false)` is equivalent, but possibly slightly preferred, to: `not(whoever.hasCard whatever)`

Let's go back to the model. With regards the diner, we leave it as an exercise for the reader to create a Diner class, that would have attributes such as the cost of the fixed menu. Two restaurants should be created: Raleigh and Blakes. Raleigh will be located in `Telegraph_Av_2405`, and its attribute 'foodcost' (that represents the cost of the fixed menu) will be set equal to 4.0. Blakes will be located in `Telegraph_Av_2134` and its 'cost' will be set to 6.0 dollars. (Note that from Section 4.9 we will start presenting the code of the Atm tutorial for your verification. If you need, you can start already browsing those files from [here](#)).

Are you done with the restaurant files? Ok, now we are ready to play a little more with the activities and workframes of our scenario. Let's say that we want our agents to move to the Atm of the bank where they opened their account whenever the cash they are carrying goes below a certain level. Let's say that this level is 10, a little above the cost of a lunch at any of the two restaurants. We have modeled only Alex_Agent and his objects, for the moment, so we will write something like this inside the Student group (`workframes: tag`):

```
workframe wf_moveToLocationForCash {  
  repeat: true;  
  variables:
```

```
when(knownval(Alex_Cash.amount < 10.00))
do {
    moveToLocationForCash(Telegraph_Av_113);
    conclude((Alex_Cash.amount = Alex_Cash.amount
+ current.preferredCashOut), bc:100, fc:100);
    moveToLocationForCash(SouthHall);
}
}
```

(we will see later - but you might as well imagine that already - that this is not a good approach to a general solution of the issue: can you guess why?). Of course, we also need to pass some new initial beliefs to the agent:

```
initial_beliefs:
    (Blakes_Diner.location = Telegraph_Av_2134);
    (Raleigh_Diner.location = Telegraph_Av_2405);
    (Boa_Atm.location = Telegraph_Av_113);
```

Now we better create a new move activity, more generic than the ones we have used before:

```
move moveToLocationForCash(Building loc) {
    location: loc;
}
```

where, of course, the location can be specified inside the workframe, as we have done in the workframe MoveToLocationForCash.

We will also need to give the agent an idea of what the cost of each diner/restaurant is, starting with Raleigh; so, modify the agent's initial beliefs:

```
(Raleigh_Diner.foodcost = 4.0);
    (Blakes_Diner.foodcost = 4.0);
```

and then modify also the moveToRestaurant workframe...

```
workframe wf_moveToRestaurant {
    repeat: true;
    variables:
    when
        (knownval(current.howHungry > 2.00) and
        knownval(current.location != Telegraph_Av_2405))
    do
        {moveToRestaurant()};
        conclude((current.howHungry = current.howHungry - 5.00),
bc:100, fc:100);
```

```

        conclude((Alex_Cash.amount = Alex_Cash.amount -
                Raleigh_Diner.foodcost), bc:100, fc:100);

        moveToSouthHall();
    }
}

```

Save everything and execute the usual steps to compile and parse your model. You should get something like this:

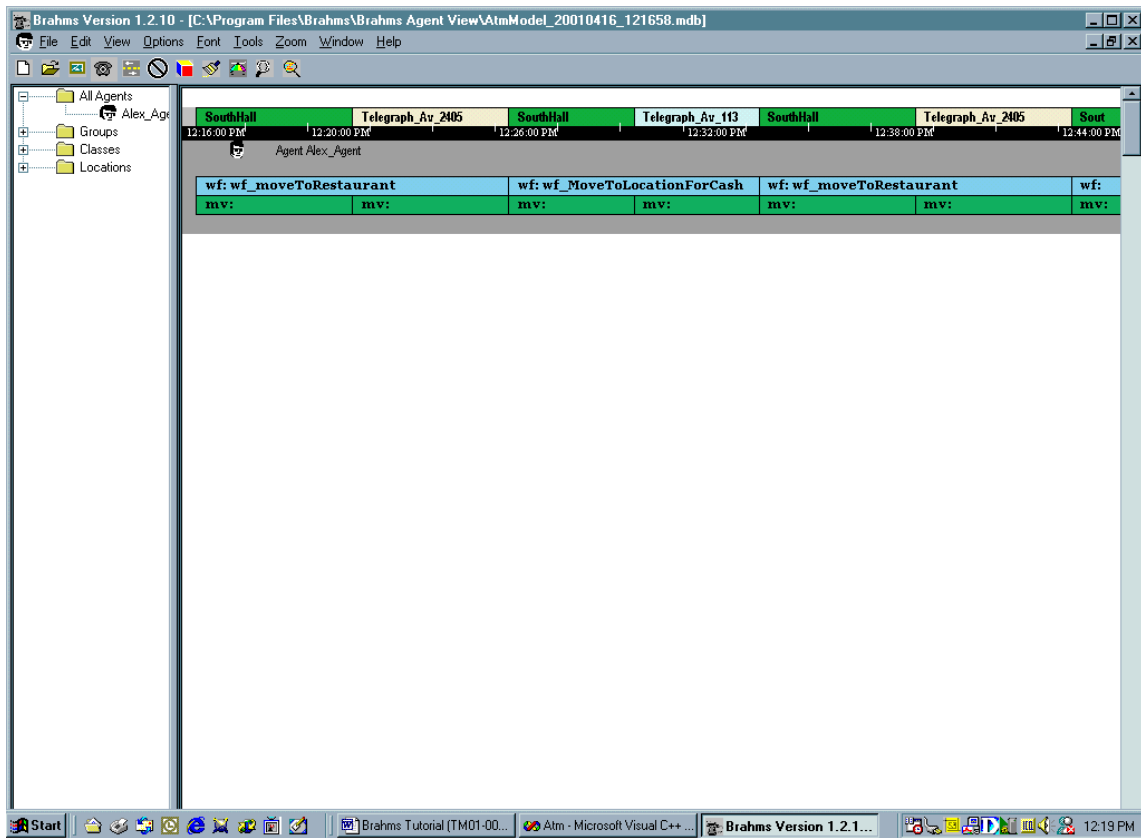


Figure 17. The Agent Viewer and Other Activities

Our agent checks his location and compares it with the location of a restaurant; if the two locations are the same, the agent realizes that he is already in the restaurant. Note that we have not given the agent a belief about the location of the diner *object*. The reason the agent is still able to derive this information lies in the fact that whenever an agent reaches a location, she can see the other objects in that location (otherwise she would have no idea of where a diner, is unless she is explicitly given a belief about that).

4.9 LESSON VII: THOUGHTFRAMES AND OTHER ACTIVITIES

4.9.1 INTRODUCTION

This chapter will teach you how to use thoughtframes and other primitive activities (such as communication activities) in Brahms models.

4.9.2 TASK

Now, try to connect the variation of an agent's hungriness to an object – the Campanile – that at regular intervals broadcasts a signal to all objects and agents in the scenario. The Campanile is actually signaling the flow of time. In this section you will have to use communication activities (such as the broadcast activity executed by the Campanile) and thoughtframes: when agents perceive the Campanile signal, their hungriness levels increase; moreover, depending on how much cash they have left, they will decide which restaurant to move to; finally, they will decide whether they need to go to the Atm even before going to the restaurant and spending their money. For the moment, you will not use variables – the choice of the different restaurants, for example, will be hard-coded in the activities that make the agent actually go to a specific restaurant.

4.9.3 DESCRIPTION

4.9.3.1 THOUGHTFRAMES

Thoughtframes define deductions, mostly referred to as *production rules*. Thoughtframes are similar to workframes, but are taken to be *inferences* an agent (or object) makes without executing any activities. Thoughtframes have the same preconditions and consequences as workframes. Thoughtframes have no activities, consume no time, and cannot be interrupted. Once the preconditions of a thoughtframe match the beliefs of the agent or object, its consequences are automatically executed, similar to forward-chaining rules. An important point is that the preconditions in thoughtframes for object *always* only match with the beliefs of the object. Another important point is that the consequences in a thoughtframe can only create new beliefs for the agent or object, and *cannot* create new facts in the world.

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.

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.

4.9.3.2 CREATE-OBJECT ACTIVITY

Primitive [create-object](#) activities allow the modeler to create new objects at runtime or to make copies of existing objects dynamically. The modeler can specify when the actual creation or copying takes place during the execution of the activity, by setting the when-value to either *start* or *end*. Create-object activities can be used, for example, to model a fax machine creating a new instance of a fax elsewhere, or a customer creating an order. In addition, in a create-object activity, an object can automatically be connected to a conceptual object or placed at a location.

4.9.3.3 COMMUNICATION ACTIVITY

The predefined primitive [communication](#) activities transfer beliefs to/from one agent to one or several other agents, or to/from an (information carrier) object.

An agent can give (send) and request (receive) beliefs. One can think of the agent-to-agent and agent-from-agent communication primitives as modeling a simple conversation where agent A asks B to tell him anything B knows about subjects X (From B), and likewise tells B anything that A knows about subjects Y (To A). In either case, beliefs must be specified in so-called transfer-definitions. In the first case, it specifies what beliefs will be transferred to the "To" agent or object. In the second case, it matches these beliefs against the beliefs of the "From" agent or object. Only the agent or object's beliefs that match the specified beliefs in the transfer-definition are transmitted.

A belief specified in a primitive communication activity is deemed to match another belief under the same conditions that a workframe *known-type* precondition is deemed to match a belief. The specified beliefs are transmitted only if they are actually held by the agent or object. In other words, an agent or object has to have the belief before it can communicate (i.e. tell) the belief to another agent or object. The transmitted beliefs overwrite any beliefs the recipient might have about the same object-attribute or object-relation.

Beliefs transferred to or from an object, model information stored in or on the object. For example, a modeler can use a communication activity to model the reading of information from, or the writing of information to, a fax, paper, bulletin-board, or a computer system. If transmitted beliefs contain variables that remain unbound in the recipient-initiator's workframe, then those variables are bound from matching beliefs supplied by the sender-responder.

4.9.3.4 BROADCAST ACTIVITY

Primitive [broadcast](#) activities work like communication activities. Here, however, the acting agent is broadcasting the matching beliefs to *all* other agents in the same location as the acting agent. One can think of the broadcast activity as modeling an agent shouting information to other agents.

When an agent broadcasts, the agent transmits beliefs to all other agents in the *same* geographical area (location) if the agent has a location, or to all other agents if the agent has no location. If an object broadcasts, the object most likely transmits a belief about itself (e.g., a phone ringing), which will be received by the agents in the same location if the object has a location, or by all agents if the object does not have a location.

4.9.3.5 JAVA ACTIVITY

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.

4.9.4 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_tfr_stx.htm, http://www.agentisolutions.com/documentation/language/ls_bac_stx.htm, http://www.agentisolutions.com/documentation/language/ls_jac_stx.htm, and pages linked from there.

4.9.5 TUTORIAL

Up till now, the agents of the Atm world have been acting following basic needs, with no sense of time. Now we will create a new class of objects – clocks – and a specific instance (the Campanile) to signal the passage of time to all the participants to the simulation. The Campanile will [broadcast](#) a signal at regular times. This signal will be received by every agent, that might or might not modify her behavior accordingly.

Let's first consider a new MyClock.b class:

```
package gov.nasa.arc.brahms.atm;
class MyClock {
  attributes:
    public int time;
  activities:
    primitive_activity asTimeGoesBy() {
      random: false;
      max_duration: 3599;
    }
    broadcast announceTime() {
      random: false;
      max_duration: 1;
      about:
        send(current.time = current.time);
        when: end;
    }
  workframes:
    workframe wf_asTimeGoesBy {
      repeat: true;
      when(knowval(current.time < 20))
      do {
        aTimeGoesBy();
        conclude((current.time = current.time + 1),
          announceTime());
      }
    }
}
```

This clock works in a very simple way: it spends 3599 time units (or seconds) waiting, and then [‘signal’](#) to the world (in one second, or unit of time) the fact that one hour has passed. It does so for 20 hours (the usual working day of a Berkeley student – sleep is not an option!). Then, it stops doing anything.

Now consider a specific instance of this class: let’s call it `Campanile_Clock.b`.

```
package gov.nasa.arc.brahms.atm;
object Campanile_Clock instanceof MyClock {
//    location: SouthHall;
//    no location has been added, so that the Campanile can broadcast to
//    all the agents, wherever they are.
    initial_beliefs:
        (current.time = 1);
    initial_facts:
        (current.time = 1);
}
```

No location has been given to the Campanile, so that when it broadcasts, all the agents and objects in the simulation, wherever they are, they will receive the signal.

Ok. Now we have to make our Students do ‘something’ in response to these signals. Let’s go back to the `Student.b` file. Firstly, we add a new attribute:

```
public int perceivedtime;
```

and we also give the Student (and all of them, rather than `Alex_Agent` alone) an initial_belief about such `perceivedtime`:

```
(current.perceivedtime = 1);
(Campanile_Clock.time = 1);
```

(we need to give a belief about the Campanile only to ‘bootstrap’ the simulation and in particular the activity of studying for the agent. Note that it is not necessary in itself to give any belief about the campanile’s time to the agent, since the belief will be broadcast and will enter the set of the agent’s beliefs regardless of whether the agent had a prior belief or not). Then, we create a new [thoughtframe](#) where we put these new concepts into action. Thoughtframes can be used to model reasoning, problem-solving (for example inside some composite activity – cf. 4.11), mental states... Here we interpret hungriness as a (at least partially) mental state, and under the thoughtframe section of the Student body declaration therefore we write:

```
thoughtframes:
    thoughtframe tf_feelHungry {
        repeat: true;
        when(knownval(Campanile_Clock.time >
current.perceivedtime))
```

```

        do {
            conclude((current.perceivedtime =
Campanile_Clock.time), bc: 100);
            conclude((current.howHungry = current.howHungry +
3.00), bc:100);
        }
    }

```

Note that thoughtframes act on beliefs and conclude only beliefs, and facts are ignored.

Then, we also add a new activity (Study) and another activity (Eat), so that we split into smaller units the activity of going to the restaurant for food. Hence, let's write:

```

workframe wf_study {
    repeat: true;
    when(knownval(Campanile_Clock.time < 20) and
        knownval(current.howHungry < 21) and
        knownval(current.location = SouthHall))
    do {
        Study();
    }
}

```

This workframes is supposed to make the Student study, until he is very hungry. Modify the goToRestaurantForFood workframe accordingly, i.e. set the value above which the student will feel the urge to go to the diner as 20.00. You also need to write a new primitive activity:

```

primitive_activity Study() {
    max_duration: 1500;
}

```

Then, let's modify the wf_moveToRestaurant workframe so to split it in 3 steps: 1) when the agent is hungry, she moves to the restaurant; 2) at the restaurants, she eats; 3) she goes back to study.

We will probably need just primitive activities: a generic moveToLocation(Building loc) activity, whose parameter is used to make it more specific; and a primitive Eat() activity. Let's then write (in the Student.b file):

```

activities:
    move moveToLocation(Building loc) {
        location: loc;
    }
    primitive_activity eat() {

```

```
max_duration: 400;
```

```
}
```

which will be triggered by the following new workframes:

```
workframe wf_moveToRestaurant {
    repeat: true;
    when(knownval(current.howHungry > 20.00) and
        knownval(current.location != Telegraph_Av_2405))
    do {
        moveToLocation(Telegraph_Av_2405);
    }
} // wf_moveToRestaurant

workframe wf_eat {
    repeat: true;
    when(knownval(current.howHungry > 20.00) and
        knownval(current.location = Telegraph_Av_2405))
    do {
        eat();
        conclude((current.howHungry = current.howHungry -
            5.00), bc:100, fc:100);
        conclude((Alex_Cash.amount = Alex_Cash.amount -
            Raleigh_Diner.foodcost), bc:100, fc:100);
        conclude((current.readyToLeaveRestaurant = true),
bc:100);
    }
} // wf_eat

workframe wf_backToStudy {
    repeat: true;
    when(knownval(current.readyToLeaveRestaurant = true) and
        knownval(current.location = Telegraph_Av_2405))
    do {
        moveToLocation(SouthHall);
        conclude((current.readyToLeaveRestaurant = false),
bc:100);
    }
} // wf_backToStudy
```

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

You should have noted the presence of a new Student attribute: `readyToLeaveRestaurant`. There are always several ways of dealing with a situation in Brahms. In this case, the action of eating triggers several consequences, one of those being a feeling of 'satisfaction' that urges the agent to leave the restaurant having satisfied his needs, and is modeled through the abovementioned attribute. If you decide to follow this approach, then you must also remember to declare the attribute in the agent's definition and give her an initial belief:

```
(current.readyToLeaveRestaurant = false);
```

Note that now you can also correct a non pleasant aspect of the `wf_MoveToLocationForCash` workframe – the fact that it called the activity `moveToLocationForCash` twice – also when the agent was actually going back to study. Use the new activity and correct the workframe, now letting the student go back to a University Hall after getting the money and before spending the cash at some restaurant (the more realistic case is the one where the students goes straight to the restaurant after getting the money: we will model this in section 4.13):

```
workframe wf_moveToLocationForCash {
    repeat: true;
    variables:
    when(knownval(Alex_Cash.amount < 10.00))
    do {
        moveToLocation(Telegraph_Av_113);
        conclude((Alex_Cash.amount = Alex_Cash.amount +
            current.preferredCashOut), bc:100, fc:100);
        moveToLocation(SouthHall);
    }
}
```

As you can see, your model is getting step by step more general and re-usable. You can verify and compare your code with the code we have prepared - [here](#). In the next section we will study how to make it even more general and adaptable.

4.10 LESSON VIII: VARIABLES

4.10.1 INTRODUCTION

This chapter will teach you how to use variables in Brahms models.

4.10.2 TASK

Until now we have been using groups almost as if they were specific agents: referencing specific objects such as `Alex_BankCard`, and sacrificing generality to simplicity. The task in this lesson is to make the activities we have already created more general by using variables: for example, the cash that the Student uses will not be `Alex_Cash`, but a 'generic' cash that is bound as `Alex` only at run time. This way the same construct can be used for any other agent. The same reasoning must hold for restaurants, banks, bank cards, and so on, given that there is more than one restaurant, more than one bank, etc.

4.10.3 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 in a frame make the frame a template for activities (workframe) or reasoning (thoughtframe) that agents and objects may perform. Variables may have quantifiers, as we will describe below. Brahms supports three quantifiers for variables: *foreach*, *forone*, and *collectall*. Variables can be used in preconditions, consequences, detectables, and as parameters for activities. The quantifier affects the way a variable is bound to a specific instance of the defined type (group or object class) of the variable.

4.10.3.1 FOR-EACH

A for-each variable is bound to only one instance, but for each instance that can be bound to the variable, a separate *workframe instantiation* is created. Consider, for example, a precondition and workframe indicating:

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

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

```
when (knownval(order is-assigned-to Allen))
do {
    work-on(order);
}
}
```

If three Orders are assigned to agent Allen and agent Allen has beliefs for all three of the orders matching the precondition, Brahms creates three workframe instantiations (wfi's) for agent Allen, and in each wfi the for-each variable is bound to one of the three orders. This means that Allen works on all three the orders, *one order at a time*. The *order* in which Allen works on the three orders is undefined.

4.10.3.2 COLLECT-ALL

A collect-all variable can be bound to more than one instance. The variable is bound to all matching belief-instances, and *only one wfi is created*. Consider the previous example with a different variable declaration:

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

In this situation the simulation engine creates *one wfi* and binds the collectall variable to a *list* of all three orders. This means that Allen *works on all three orders at the same time*, cutting the actual activity duration in three.

4.10.3.3 FOR-ONE

A for-one variable can be *bound to only one* belief-instance, and *only one wfi* is created. A for-one variable binds to the first belief-instance found and ignores other possible matches. As far as the modeler is concerned, the selection is random, meaning it in the case of multiple matches it is undefined which order is selected. In the previous example workframe, the variable declaration would look like:

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

In this situation, one wfi gets created, and only one of the three orders gets bound. This means that Allen randomly *works on just one of the orders*, cutting the actual activity duration in three as in the collectall case.

4.10.3.4 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. unassigned variable is unbound (that is, it does not get a value) when a frame instantiation is created; an unassigned variable gets a value through a communicated belief or object creation activity, which binds the variable to a newly created object.

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.

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.

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.

4.10.4 SYNTAX

F Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_var_stx.htm, and pages linked from there.

4.10.5 TUTORIAL

Let's go back to the code in Student.b. Let's check again how the student – *any* student – decides whether he or she needs to get more cash from the Atm:

```
workframe wf_moveToLocationForCash {  
    repeat: true;  
    variables:  
    when(knownval (Alex_Cash.amount < 10.00))  
    do {  
        moveToLocation (Telegraph_Av_113);  
        conclude ( (Alex_Cash.amount = Alex_Cash.amount +  
current.preferredCashOut), bc:100, fc:100);  
    }  
}
```

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

```
        moveToLocation(SouthHall);
    }
}
```

so, every student will check the amount of cash of Alex, and every student will go for cash to the Atm of the bank where Alex has his account! This of course is wrong. [Variables](#) can correct this situation.

The way to solve this impasse is through matching, using variables in the precondition of the workframe in the Student group. The same reasoning applies to relations. Say that we have a relation in the student group: public Bankcard hasCard. To match a specific student with a specific card, we need in the file of the agent (for example, Alex_Agent) who is memberof Student, something like: initial_facts (or/and beliefs):

```
(current hasCard Alex_BankCard)
```

then, we would bind the agent to the proper card from inside the Student file with a code like:

```
workframe matchBankcardExampleInGroupStudent {
    repeat:false;
    variables:
    forone(BankCard) bc; // let's assume the student has only
    // one bankcard
    when (knownval(current hascard bc)) // here the engine matches and
    // binds the var. bc

    do {
        doSomeActivityThatLowersBankCardBalanceWithThreeDollars();
        conclude((bc.balance = bc.balance - 3.00), bc:100, fc:100);
    }
}
```

Variables are powerful in rule-based programming. When you want to use an agent or object or value from an agent's belief within a workframe or thoughtframe, you most likely will have to use a variable in the precondition. You also use a variable to make the matching of the rule more general (note, in fact, that in the example above the precondition would match for every student and the bankcard that belongs to the student. It would even be possible to have the student have multiple bankcards - i.e. multiple beliefs with the relation; by using the `forone` variable - instead of a `foreach` - the engine would simply match to one of the beliefs).

When you use variables in workframes, the first thing the engine does it to checks all the variables of that kind; the second thing consists of binding the variables to the specific item we are considering in the preconditions. By giving it a name inside the workframe, we become able to use them in methods being called, conclusions, etc. In particular, their being used as parameters in activities is one of the most useful features in the language. Note that, in fact, you can write something like:

```
knownval(p = current.something)
```

which will effectively create a binding and pass the `current.something` value to `p`, that could be used as a parameter in a method/activity (importantly, you cannot pass a form like `object.attribute` as a parameter for any action). For example, in the `communicatePin` activity we can create a variable `p` for the pin and pass that as the parameter.

Similarly, we can use variables to specify the cost of the restaurants in lines such as;

```
conclude((current.howHungry = howHungry - amount_of_food_at_restaurant),  
bc; 100);
```

where the variable that determines by how much the hunger of the agent has decrease after eating might be decided and bound at run-time depending on which restaurant the agent is in at that moment.

Consider also the following two lines:

```
knownval(current hasAccount bka) and  
knownval(bka.pin = p) and
```

The first line binds a specific Account `bka` to the current agent; then, the second line will bind the pin to that specific Account. In other words, when we use complex relations (e.g., a son of b son of c son of...) we can bind them together by respecting their logical order. Furthermore, the equality sign in the lines above will also give to `p` the value of `bka.pin`.

Binding and beliefs are really crucial in workframes and thoughtframes. To use variables, you have to bind them with the preconditions (there are exceptions to this rule, and we will discuss them later in this section). But to evaluate the preconditions, your agents need beliefs. If your code is not working as you expect, try checking first if these crucial steps (beliefs and binding) have been coded correctly!

Be careful about using variables with locations: note that,

```
forone(Cash) cs;  
// forone(location) lc; this one would not work  
forone(Building) bd; // this line will work  
  
when(knownval(current hasCash cs) and  
knownval(cs.amount < 16.00) and
```

```
knownval(current.howHungry > 20.00)  
knownval(current.location = dn.location)
```

The first line will not work, the second will. A very general way to deal with locations and geography concepts is to refer to them as AreaDef, which is a 'meta type' (cf. http://www.agentisolutions.com/documentation/language/ls_att_sem.htm). In the code above for example you could write:

```
forone(AreaDef) bd; // this line will work
```

and this line will work regardless of whether bd is a building, a city, a world, etc...

Note that the above examples are assigned variables. Pre- and post- assigned variables are both assigned variables (consider the example given earlier, `<myagent>.car = <mycar>`, where `myagent` is pre-assigned and `mycar` is post-assigned; a further example: in `current.mytime = time`, `time` is post-assigned). If we were using relations (that, as we have discussed, can have multiple values) in combination with `foreach` statements, we could produce multiple instantiations of the same workframe. With the `collectall`, instead, we would have one only workframe instance that would process all the (eg the variable contains instantiation of different attributes/relations). Unassigned variables instead are not inserted (nor bound) in any preconditions – rather, they are used and get their context in a create-object activity (unassigned variable will be bound to the destination object), or in communications to bind the variable to a context based on what is communicated. For example, in a communication activity where we want to transmit a boolean `whatever` that can be either false or true, we can write:

```
Communication_activity [...]  
Send (current.whatever = current.whatever)
```

or also:

```
Send (current.whatever = anyvalue)
```

And both versions will work.

So, let's try to apply this new tools to the Atm case. We will have to modify the workframe `moveToLocationForCash` as follows:

```
workframe wf_moveToLocationForCash {  
  repeat: true;  
  
  variables:  
    forone(Cash) cs;  
    forone(Atm) at;  
    forone(Bank) bk;  
    forone(Building) bd;  
    forone(Account) ac;
```

```

when(knownval(current hasCash cs) and
     knownval(cs.amount < 10.00) and
     knownval(current hasAccount ac) and
     knownval(ac openedWithBank bk) and
     knownval(at ownedbyBank bk) and
     not(current.location = at.location) and
     knownval(at.location = bd ) and
     knownval(current.readyToLeaveRestaurant = false))
do {
  moveToLocation(bd);
  conclude((cs.amount = cs.amount + current.preferredCashOut),
bc:100, fc:100);
  moveToLocation(SouthHall);
}
}

```

(here we are still making the student first go back to a University Hall after she got her money, before spending the cash at some restaurant, because it is simpler. But soon – section 4.13 - you will be asked to model this more realistically by having the student go straight to the restaurant after getting the cash). Remember to add Alex’s initial beliefs:

```

(current hasAccount Alex_Account);
(Alex_Account openedWithBank Boa_Bank);
(Boa_Atm ownedbyBank Boa_Bank);

```

and also modify the Study workframe as follows:

```

workframe wf_study {
  repeat: true;
  variables:
    forone(Cash) cs;

    when(knownval(Campanile_Clock.time < 20) and
         knownval(current hasCash cs) and
         knownval(current.howHungry < 21) and
         knownval(current.location = SouthHall) and
         knownval(cs.amount >10.00))
    do {
      study();
    }
}

```

}

There are some interesting issues in this construct. We have an example of the successive binding we mentioned earlier, where variables are bound one after the other, sequentially. We can also see how those variables can be used as parameters in the activities that are triggered from inside the workframe, as well as in the conclusions. Also, note that as the activities become more complex, so become their interactions. Take the preconditions in the `goToLocationForCash`: try to see what happens if we remove the precondition:

```
knownval(current.readyToLeaveRestaurant = false))
```

The reader should now try to apply these concepts to the diner case. Eating at the restaurant is structured into the various steps above, that can be now made more general by passing parameters and using variables. You should have preconditions in the workframe `wf_move` that test in which location-restaurant your agent is. When modeling the restaurants, you will have to consider the fact the restaurants have different prices, and the agent might chose the restaurant each time after having considered how money she is carrying. Such chosen restaurant can be modeled either as a relation or as an attribute. What would be better? This is another open question of modeling. One factor to consider might be, for example, the following: while attributes can only hold (currently) single values, relations can be multiple.

After you attempt to complete this scenario, you can find comparison code up to this section [here](#).

4.11 LESSON IX: COMPOSITE ACTIVITIES

4.11.1 INTRODUCTION

This chapter will teach you how to use composite activities in Brahms models.

4.11.2 TASK

This is an important Lesson: you need to organize the various activities that you wrote earlier in the tutorial, into composite activity, and then increase the number of activities to make the model more realistic. In particular: the activity 'goToAtm', will be a composite activity that will comprehend several different steps (moveGoAtm, insertCard, getMoney, leaveAtm); goToRestaurant, similarly, will be composed of 3 activities: moveToRestaurant, Eat, leaveRestaurant. You will try to use the same 'moveToLocation' primitive activity for all the various cases: you will see that you can call it from different workframes, and by passing the appropriate parameters, make it do different things.

4.11.3 DESCRIPTION

The activities in a workframe are one or more primitive activities, one or more composite activities or both. A [composite activity](#) includes one or more workframes, any of which may trigger other composite activities, each with its own workframes (Figure 18). Other than a few predefined atomic activities that have semantics, activities are differentiated solely by the modeler's description and use of them (see the sections on primitive activities and composite activities).

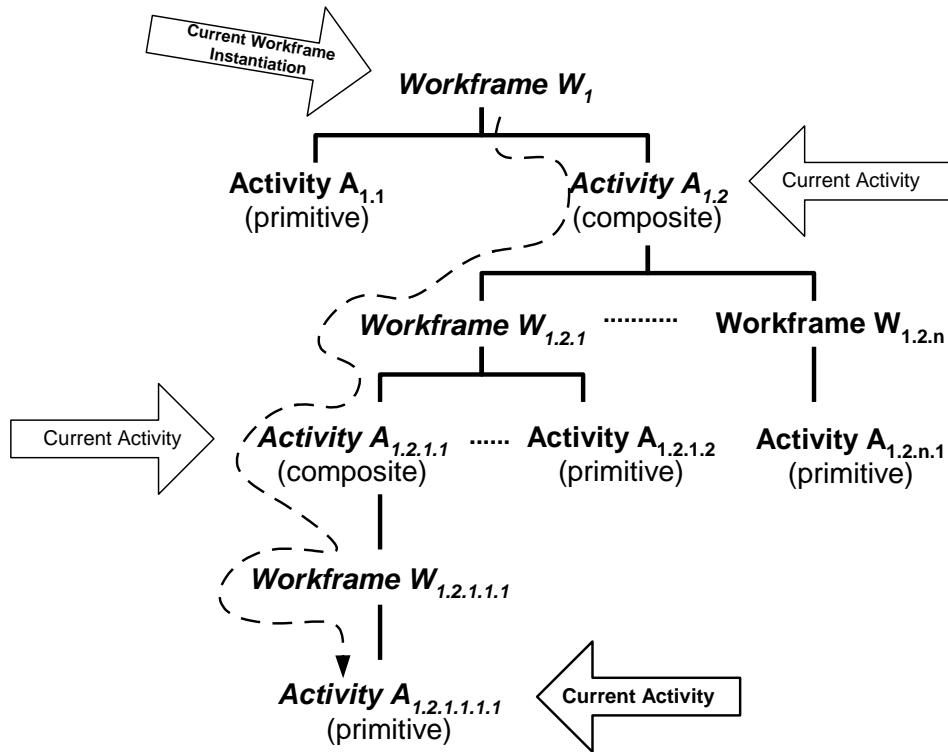


Figure 18. Workframe-Activity Hierarchy

4.11.3.1 PRIMITIVE ACTIVITIES

Primitive activities take time, which may be specified by the modeler as a definite quantity or a random quantity within a range. However, because workframes can be interrupted and never resumed, when an activity will finish cannot be predicted from its start time. Primitive activities are atomic behaviors that are not decomposed. Whether something is modeled as a primitive activity is a decision made by the modeler. A primitive activity also has a *priority* that is used for determining the priority of workframes.

4.11.3.2 COMPOSITE ACTIVITIES

A composite activity expresses an activity that may require several workframes to be accomplished. Since activities are called within the do-part of a workframe, each is performed at a certain time within that workframe. The body of a workframe has a top-down, left-to-right execution sequence. Preference or relative priority of workframes can be modeled by grouping them into ordered composite activities. The workframes within a composite activity, however, can be performed in any order, depending on when their preconditions are satisfied. In this fashion, workframes can explicitly control executions of composite activities, whereas execution of workframes depends not on their order but on the satisfiability of their preconditions and the priorities of their activities (see Figure 18).

A composite activity can terminate in the following four ways. First, a composite activity terminates whenever the workframe in which it is executed terminates, due to a workframe detectable of type complete or aborts. Second, a composite activity terminates whenever a detectable of type complete or abort is detected within the composite activity. Third, a composite activity terminates immediately whenever an end condition declared within the composite activity is activated. And fourth, a composite activity terminates when the modeler has defined it to be ended "when there is no more work available" and no more workframes in the composite activity are available or being worked on. During the execution of a composite activity, the engine continuously checks whether the agent has received a belief that matches any end-conditions.

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.

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.

4.11.4 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_att_cac.htm, and pages linked from there.

4.11.5 TUTORIAL

We will now try to add some complexity to the Atm scenario by modeling subactivities with more details and representing them inside a composite activity. We will do this both for the activities related to getting money out of an Atm, and also for those related to eating at a restaurant.

We will start with the Atm case and we will use a composite activity that we will call: useAtm. First, create a new workframe:

```
workframe wf_useAtm {
    repeat: true;

    when(knownval(current hasCash cs) and
        knownval(cs.amount < 10.00) and
        knownval(current hasAccount ac) and
        knownval(ac openedWithBank bk) and
        knownval(at ownedbyBank bk) and
        not(current.location = at.location) and
        knownval(at.location = bd ) and
        knownval(current.readyToLeaveRestaurant = false))
    do {
        useAtm();
    }
}
```

Create new attributes that will be used for this activity:

```
public Bank chosenBank;
public boolean needCash;
public boolean receivedCash;
public boolean pinCommunicated;
public boolean readyToLeaveAtm;
```

and give them the proper values in the agent body (all the booleans are false; for Alex_Agent, the chosen bank is Boa_Atm). Now: useAtm is a composite activity. Its workframe is defined as all others, after the `workframes: tag`. And the activity is placed among other activities. The novelty is that the composite activity itself has, inside its body, other activities and other workframes.

Here, your goal is to model a simple composite activity. The student goes to the Atm, inserts the BankCard, communicates the pin, and then (we assume, for the moment, that the pin is correct) gets the card back and the preferred amount of cash. This is very simplistic: there is not real interaction with the Atm, there are no errors with the Pin...This is ok! This composite activity will be probably already enough to keep you working for a little while – we will model more complex interactions in the next section.

A sketch of the code would be the following. Look at that code only after having experimented yourself, and remember: this is only one of the many ways you could use to reach your goal.

You will start defining a composite activity inside the `activities:` declaration:

```
composite_activity useAtm() {
```

then, inside this composite activity, you will define the sub-activities and the related workframes. In other words, the composite activity will work as if it were scaling down the usual structure of an agent/object body:

```
activities:
```

```
primitive_activity insertBankCard() {
    max_duration: 45;
}

communicate communicatePIN(Atm at3, Account bka) {
    max_duration: 20;
    with: at3;
    about:
        send(bka.pin = p);
    when: end;
}

primitive_activity getCash() {
    max_duration: 50;
}
```

Finally, as mentioned, you will add the workframes (yes, still inside the composite activity body):

```
workframes:
```

```
workframe wf_moveToLocationForCash {
    repeat: true;
    variables:
```

```
        forone(Cash) cs;
        forone(Atm) at;
        forone(Bank) bk;
        forone(Building) bd;
    when(knownval(current hasCash cs) and
        knownval(current.chosenBank = bk) and
        knownval(at ownedbyBank bk) and
        knownval(cs.amount < 10.00) and
        not(current.location = at.location) and
        knownval(at.location = bd ))
    do {
        moveToLocation(bd);
    }
}

workframe wf_insertBankCard {
    repeat: true;
    variables:
        forone(BankCard) bkc2;
        forone(Atm) at2;
        forone(Bank) bk2;
        forone(Building) bd2;
    when(knownval(current hasBankCard bkc2) and
        knownval(current.chosenBank = bk2) and
        knownval(at2 ownedbyBank bk2) and
        knownval(current.receivedCash = false) and
        knownval(current.location = at2.location) and
        knownval(current.pinCommunicated = false) and
        knownval(current contains bkc2))
    do {
        insertBankCard();
        conclude((current contains bkc2 is false), bc:100,
fc:100);
        conclude((at2 contains bkc2), bc:100, fc:100);
    }
}

workframe wf_communicatePIN {
```

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

```
repeat: true;
variables:
    forone(Account) bka;
    forone(BankCard) bkc3;
    forone(Atm) at3;
    forone(Bank) ba3;
    forone(Building) bd3;
when(knownval(current hasBankCard bkc3) and
    not(current contains bkc3) and
    knownval(current hasAccount bka) and
    knownval(current.chosenBank = ba3) and
    knownval(at3 ownedbyBank ba3) and
    knownval(current.pinCommunicated = false) and
    knownval(current.location = at3.location) and
    knownval(at3 contains bkc3))
do {
    communicatePIN(at3, bka);
    conclude((current.pinCommunicated = true), bc:100);
    conclude((current contains bkc3), bc:100, fc:100);
    conclude((at3 contains bkc3 is false), bc:100,
fc:100);
}
}

workframe wf_getCash {
    repeat: true;
    variables:
        forone(BankCard) bkc4;
        forone(Cash) cs4;
        forone(Atm) at4;
        forone(Bank) bk4;
    when(knownval(current hasBankCard bkc4) and
        knownval(current hasCash cs4) and
        knownval(current contains bkc4) and
        knownval(current.chosenBank = bk4) and
        knownval(at4 ownedbyBank bk4) and
        knownval(current.pinCommunicated = true))
```

```
        do {
            getCash();
            conclude((cs4.amount = cs4.amount +
                current.preferredCashOut), bc:100, fc:100);
            conclude((current.pinCommunicated = false), bc:100);
            conclude((current.receivedCash = true), bc:100);
            conclude((current.readyToLeaveAtm = true), bc:100);
        }
    }

    workframe wf_BackToStudy {
        repeat: true;
        variables:
            forone(Atm) at5;
        when(knownval(current.readyToLeaveAtm = true) and
            knownval(current.location = at5.location))
        do {
            MoveToLocation(SouthHall);
            conclude((current.needCash = false), bc:100, fc:100);
            conclude((current.readyToLeaveAtm = false), bc:100);
            conclude((current.receivedCash = false), bc:100);
        }
    }
}
```

You should now apply the same strategy to the case of the restaurant, by making the activities of going, eating and coming back from a generic restaurant a composite activity. The choice of the restaurant should be modeled as a thoughtframe. Moreover, this choice will have to be coordinated with a thoughtframe where the agent decides whether she needs cash or not before going to the Atm (hint: you can use a new needCash attribute to trigger the useAtm composite activity). Be very precise when using composite activities. Like normal activities, they can be interrupted and resumed. However, if your code is not written properly a composite activity might create a loop or an impasse that halts your simulation (see more on this in the section on debugging, 4.14.2).

Comparison code is offered [here](#).

4.12 LESSON X: MULTI-AGENT, RANDOMNESS, AND COMPLEX INTERACTIONS

4.12.1 INTRODUCTION

This chapter will teach you how to use multiple agents and random elements in Brahms models. It will also increase the complexity of the Atm scenario by explicitly modeling the interaction between the Atm and the agent.

4.12.2 TASK

There are three goals in this Lesson. The first goal is to include another agent in the model – Kim. She is member of the student group, like Alex, but she has different tastes, habits, time schedules. For example, Kim and Alex feel hungriness in different ways. The second goal is to randomize some of the elements of the simulation: for example, when an agent hears the campanile signal, she might or might not get hungrier; in front of the Atm, she might or might not remember the proper pin; and so on. The third and more complex step is that of modeling the interaction between the Atm and the agent. This description will still be rough – no interaction with the bank, no checking of the balance, etc. – but will add some more realism to the simulation. One note: this and the next Lesson will be more challenging than the previous ones – you will be expected to reorganize the concepts you have been learning and reassemble the code of your scenario.

4.12.3 DESCRIPTION

We do not need a new formal description of the concepts related to the use of multiple agents, because there is nothing new here: we simply need to ‘throw’ other agents into the simulation. Similarly, the basic concepts related to randomness have been already discussed: agents conclude beliefs or facts with certainty (bc and/or fc values set to 100) or uncertainty. The interesting thing here is to see the long-term effects of making these changes. We will discuss them in the tutorial subsection below.

However, there is more to say about how the simulation engine decides to ‘direct’ the staging of your simulation. Since the more activities we model, the more complex the simulation becomes, we need to be very clear about the steps the engine follows to decide what to do at every specific step of the model.

In what follows, therefore, we discuss the model of execution for a Brahms model. The model of execution defines how a Brahms model is executed, and thus describes a

simulation of a model of work practice. As a multiagent system, a Brahms model consists of a number of independent agents and objects that operate independently, but interact with each other. Wooldridge (Wooldridge, 1992) describes two possible execution models for multiagent systems, *synchronous execution*, and *interleaved execution*. In both cases the execution of a multiagent system is defined by a state σ_t of the system at time t , and a state σ_{t+1} of the system at time $t+1$ caused by a *state-transition* τ_t at time t . Keeping track of the state changes of the system over time the *history* of an executing system can be considered a sequence of state and state-transitions.

Synchronous execution

In a synchronous execution system each agent and object has an initial state defined as its *initial belief set*, closed under its deduction rules. This amounts to an initial state of the system as a collection of initial belief sets for each agent, each set closed under the agent's own deduction rules.

Agents are able to change their state by performing a *move*. A move is defined as a tuple of actions. A *transition* is a collection of moves, specifically, one for each agent. In other words, a move is a state-transition for an individual agent, whereas a transition is the global state-transition for the whole system (i.e. all the agents and objects and facts). In Brahms we use the *synchronous execution model*. The reason for this is simply the fact that we are simulating and not running in real-time. In our simulation model our agents and objects need to be synchronized according to a unique global clock.

A *world* is the situation-specific model (SSM) of the simulation, at a specific moment in the execution of the system (Clancey, 1992). A *state* is defined as just the belief-set of an individual agent, at a specific moment in the execution of the system. And, a *situation-specific model* for an agent is defined by all the existing global facts, and all the agent's beliefs at the moment of inquiry, as well as the current-, available-, and interrupted workframe and thoughtframe instantiations, and the current activities.

A *state-transition* occurs when an agent, performing a work- or thoughtframe, executes a consequence that creates a new belief or fact. A state-transition can also occur when an agent receives a new belief as a result of a) a communication, b) the detection of a fact, c) a move to a new location, or d) receiving time and date beliefs from the simulation engine. During the state-transition the simulation engine determines the effects of the transition on the agent's internal state, which can result in more transitions.

Frame execution

An order of testing and execution must be imposed in any simulation tool on conditions and operations that in principle apply or occur simultaneously. The following paragraphs describe the order in which the parts of a workframe are evaluated and executed in an implementation of Brahms.

For each agent the preconditions are the first things checked in a frame (workframes and thoughtframes). They are checked in the order in which they are declared within the frame. When all of its preconditions match (i.e., are satisfied), a frame becomes available. When a frame becomes available *frame instantiations* are created for each set of variable bindings from the precondition matching. If a frame has multiple variables that can be bound, there will be a frame instantiation created for each valid combination of variable-bindings. Each frame instantiation is executed in sequence (i.e. one after another). There can only be one frame instantiation executed at a time (in one clock-tick). The order of the sequence is undetermined.

After the preconditions match and a *workframe*²⁰ is selected it will start to work (one frame instantiation for each set of valid variable-bindings). The working time will be specified in the workframe; or, if the workframe contains any composite activities, the working time will be the cumulative time of the executed composite activities. At any time during this working time, a variety of things may happen. Consequences may be asserted, facts may be detected, and communications may occur, depending on their ordering in the do-part of the workframe. If the do-part includes one or more move activities, the agent will go to the specified locations as the moves are executed.

Within a detectable, the modeler can specify when the agent or object can detect a fact. When a workframe contains a composite activity, the modeler *must* specify the time to be "whenever", because the engine cannot calculate the total working time for the frame in advance.

When multiple detectables are declared within a *workframe*, they are checked in the order in which they are declared. When two detectables are specified to be executed at the same time, and the first states that the frame should be interrupted and the second states that the frame should be aborted, the frame will be interrupted.

The do-part of a frame is ordered, and the simulation engine evaluates the do-part components in the order in which they appear from top-to-bottom and left-to-right. The do-part may include activities and composite activities for workframes, and consequences that will be asserted as beliefs and/or facts for workframes and thoughtframes.

Frame states and transitions

As described above, frames are stateless and serve as declarative definitions, whereas *frame instantiations* are dynamically created, associated with a particular agent or object, have state, and have a related context.

The possible *states* of a frame instantiation are set forth in the following table.

²⁰ The next parts are limited to workframes, because thoughtframes do not take time, and do not have activities and detectables in them.

Table 1. Frame instantiation states

not-available	No instantiation exists for a given (frame, agent or object, context) set. Either the preconditions of the frame have no matches, or previously active instantiations have all completed and been reset with no matches. This is more or less the start-state of every frame instantiation.
available	The preconditions of the frame have been satisfied for some context and agent or object, but the frame instantiation has not yet been started by the agent or object.
working	The agent or object is performing this frame instantiation for the current clock-tick.
interrupted	<p>The workframe instantiation has already had at least one clock-tick worked on it, but the agent or object is performing some other workframe instantiation during the current clock-tick.</p> <p>Thoughtframe instantiations cannot be in this state.</p>
interrupted-with-impasse	<p>A detectable has caused the agent or object to have an impasse with the workframe instantiation. The workframe instantiation cannot continue until the condition causing the impasse is resolved.</p> <p>Thoughtframe instantiations cannot be in this state.</p>
done	The agent or object has completed all the activities in the frame instantiation. If the reset-when-done attribute of the associated frame is false, then the frame instantiation will exist in the done state. Otherwise, the preconditions will be evaluated and the frame instantiation will become either available or not-available (i.e., deleted).

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

Given these possible frame states, there are a number of different allowable state-transitions for frame instantiations. These are shown in Figure 19.

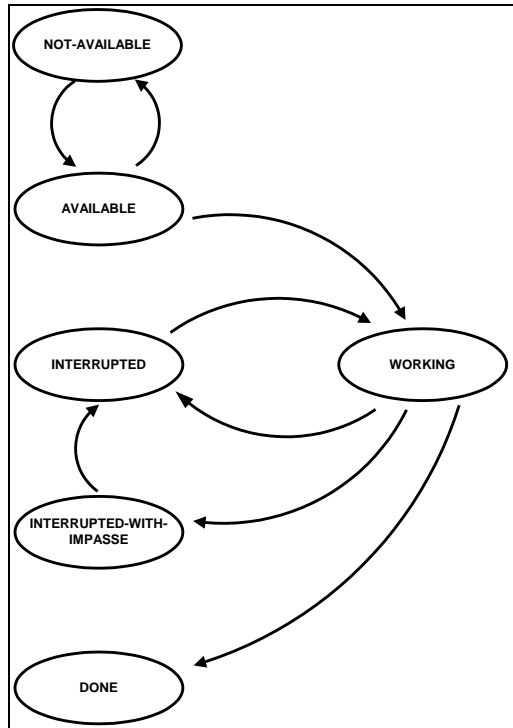


Figure 19. State-transition diagram for frame instantiations

The allowable state transitions, shown in Figure 19, are listed in **Error! Reference source not found.**, with their causes and implications.

Table 2. Frame state-transitions

<p>not-available ⇒ available</p>	<p>When the preconditions of a frame are satisfied for a particular agent or object and context, then a frame instantiation is created and put in the state <i>available</i>. This frame instantiation can then be worked on by the agent or object.</p>
<p>available ⇒ not-available</p>	<p>If an available frame instantiation has not been started, and the preconditions (which were previously satisfied) become unsatisfied, then the frame instantiation is deleted, and thus becomes <i>not-available</i>.</p>

<p>available working ⇒</p>	<p>If the frame instantiation becomes the current-work of an agent or object, then the frame instantiation has state <i>working</i>.</p>
<p>working interrupted ⇒</p>	<p>Whenever a different workframe instantiation becomes the current-work of an agent or object, the (previous) working frame instantiation becomes <i>interrupted</i>. Note that the agent can choose from the union of the sets (available, working, interrupted) for the current-work for the next clock-tick. This resolution mechanism works on the basis of priorities of the workframe instantiations.</p>
<p>working interrupted-with-impasse ⇒</p>	<p>This state change happens when the following conditions are all met: (1) an agent detects a fact, (2) the current working workframe instantiation contains a detectable that references that fact, (3) that detectable is satisfied, and (4) the impasse-type attribute of the detectable is impasse. The agent cannot continue working on the workframe instantiation until the impasse is resolved, i.e. the detectable condition becomes false due to a change in the beliefs of the agent or object. The workframe instantiation is set to <i>interrupted-with-impasse state</i>.</p>
<p>working ⇒ done</p>	<p>When all activities of the frame instantiation are completed after the current clock-tick, then the frame instantiation becomes <i>done</i>, iff the reset-when-done attribute of the associated frame is false. Otherwise, the transitions <i>working⇒available</i>, depending on whether the preconditions are still being satisfied. However, this is accomplished by creating a new frame instantiation. When <i>done</i>, the frame instantiation is deleted.</p>
<p>interrupted ⇒</p>	<p>When the agent or object picks an interrupted workframe instantiation to become the current-work for the next clock-tick, the frame instantiation becomes</p>

working	<i>working again.</i>
interrupted-with-impasse interrupted ⇒	<p>When a belief of an agent or object causes the detectable that caused an impasse to be no longer satisfied, then the impasse is removed, i.e. the belief causes the detectable-condition to no longer match the current beliefs of the agent or object. The frame instantiation can be worked on once again, so the state is changed from interrupted-with-impasse to <i>interrupted</i>, after which, in the next clock-tick the frame instantiation could transition to a working state.</p>

To decide for each agent what to work on next, the simulation engine executes a number of steps. At each clock tick, the simulation engine determines which workframe should be selected to work on next. This selection is based on the *priorities* of available, current and interrupted workframe instantiations. A *current workframe instantiation* is selected in preference to interrupted or available workframe instantiations of equal priority, so that an agent tends to continue doing what it was doing.

The selected workframe is then executed, leading to the agent detect things in the world (through detectables) and possibly begin a subactivity. When a workframe instantiation is interrupted, it is reexamined on subsequent clock-ticks to see whether it should be considered for selection. When a composite activity is terminated, because its end-condition is satisfied, the workframe instantiations *below* it are also terminated. When an activity is interrupted, Brahms saves the workframe/activity-hierarchy so the context can be reestablished after an interruption.

The questions remain; 1) How does a workframe get selected to become instantiated, and 2) When multiple workframes are instantiated, how does the engine determine the priority of a workframe instantiation?

The answer to the first question is that at every clock-tick the simulation engine checks if any of the preconditions of the agent's frames are satisfied (i.e. match with beliefs in the agent's belief-set)²¹. When all preconditions in a frame match, the frame is instantiated and each frame instantiation is set to the *available state*. At that moment, the engine includes the frame instantiation in its decision to determine what frame instantiation to work on next.

The answer to the second question, from above, tells us how this is done. Each workframe instantiation has a *priority*. The priority of each workframe instantiation is set based on the priorities of the *primitive activities* in the workframe. The priority of a workframe is the priority of its *highest priority* primitive activity.

Thus, all in all, the *emergent behavior* of agents during a simulation depends on two independent things. First, it depends on when preconditions of frames match on the belief-set of the agent. Of course, the belief-set of an agent depends on many factors during a simulation, such as detection of facts, moving to locations, communication with others, etc. This means that the behavior of an agent is first and foremost dependent on the behavior of other agents and objects in its environment, as well as the state of the environment itself. Secondly, the behavior of an agent depends on which frames are instantiated together at any moment in time. This is because each instantiated frame has a specific priority, and it will depend on the priority of the other frame instantiations whether a frame instantiation is picked as the next work to be done.

multi-tasking agents

In a Brahms simulation, an agent may engage in multiple activities at any given time, but only one workframe is active at any one time. At each clock-tick, the simulation engine determines which workframe should be selected, based on the priorities of available, current and interrupted work (see previous section). Current work is selected in preference to interrupted or available work of equal priority, so that an agent tends to continue doing what it was doing. The selected workframe is then executed, leading the agent to act in the world and possibly begin a subactivity. When a workframe is interrupted, it is reexamined on subsequent clock-ticks to see whether it should be considered for selection. When a composite activity is terminated because its end condition is satisfied, the workframes below it are also terminated. When an activity is interrupted, Brahms saves the line of activities and workframes so context can be reestablished after an interruption.

²¹ The speed at which this is done is heavily dependent on the implementation. In the old G2 engine this was done using a loop-structure, where for every agent and object, all preconditions for all frames were checked at every clock-tick. In the new Java engine, we have invented a multi-agent version of a Rete-like algorithm. We call this a Reasoning State Network (RSN). This allows the engine to only check the those preconditions in frames that can potentially match with beliefs that just changed in the agents' belief-set.

An important consequence and benefit of this combined modeling and programming paradigm is that all of the workframes of a model are simultaneously competing and active, and the selection of a workframe to execute is made *without* reference to a stack or tree of workframe execution history. This paradigm is a major difference from most other goal-oriented problem-solving systems, such as Soar (Laird et al., 1987).

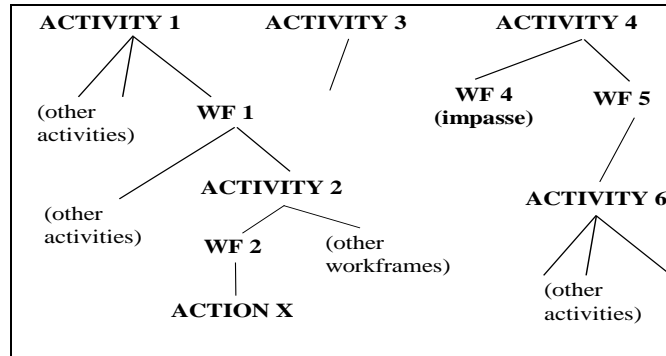


Figure 20. Multi-tasking in Brahms

An illustration of this is given in Figure 20. An agent (not shown) in a running model may have multiple competing general activities in process: Activities 1, 3, and 4. Activity1 has led the agent (through workframe WF1) to begin a subactivity, Activity2, which has led (through workframe WF2) to a primitive activity ActionX. When Activity2 is complete, WF1 will lead the agent to do other activities. Meanwhile, other workframes are competing for attention in Activity1. Activity2 similarly has a competing workframes. Priority or preference rankings led this agent to follow the path to ActionX, but interruptions or reevaluations may occur at any time. Activity3 has a workframe that is potentially active, but the agent is not doing anything with respect to this activity at this time. The agent is doing Activity4, but reached an impasse in workframe WF4 and has begun an alternative approach (or step) in workframe WF5. This produced a subactivity, Activity6, which has several potentially active workframes, all having less priority at this time than WF2.

With this activity-base paradigm, we can simulate the reactive situated behavior of humans. People are always working on many different activities, but our context forces us to be *active* in only one. However, at any moment we can change focus and start working on another competing activity, while queuing others.

4.12.4 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_wfr_des.htm, and pages linked from there.

4.12.5 TUTORIAL

The sketch of the coding will be the following.

For what refers to multiagent, you can create a new agent – Kim_Agent.b – and give her the same attributes (with different values) that we have been giving to Alex_Agent.

For the randomization, you simply need to change things here and there. For example, the agents will only detect with probably 50% that the Campanile has signaled a new hour and will get hungrier only 50% of times (the two distributions are independent in the example below):

```
conclude((current.perceivedtime = Campanile_Clock.time), bc: 50);  
conclude((current.howHungry = current.howHungry + 3.00), bc: 50);
```

Recall that the property belief-certainty is the probability (with also a default value of 100%) that the belief will be changed or created, conditional on the fact being true. That is, if the fact-certainty and the belief-certainty are each 50%, then 1 in 2 times the fact will be created and 1 in 4 times the belief will be created. If the fact-certainty is zero, then no fact will be created but the belief-certainty determines how often a belief is created.

If you have not done so already, you should create a Wells Fargo Bank (WF_Bank) and its Atm (WF_Atm). Then, you should write down accordingly the relations between Kim, her account and the bank/Atm. Do not forget Kim's initial beliefs about these relations! Thereafter, you must start modeling the Bank and the Atm themselves and the interaction between the student and the Atm. Most of the activities needed for the student have already been coded. You will probably need some changes and a new 'wait' activity for the moments when the student is waiting for replies from the Atm. You should also add the stochastic possibility that the student makes error when trying to remember or digit the pin. It will be likely that you will have to revise and update older parts of your model to achieve these goals.

For the Atm, you will probably need activities to: get the account associated to the bankcard; get the pin; pass the pin and the account to the bank; receive back the authorization (or lack thereof) from the bank, dispense or not dispense the cash. Similarly, the Bank will need activities to make it receive Pins and account numbers, verify them with the information that it is stored in the bank computers, and authorize or not authorize the payment. For the moment, just one possibility of error will suffice, and no question to the student about how much money she wants to take out (we can assume it is a fixed amount). We will complete these details in the next section, where we will also present the complete files of the scenario.

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

NOTICE: Not for use or disclosure outside of NASA Ames Research Center except under written agreement.
© 2001-2011 NASA Ames Research Center. All Rights Reserved.

4.13 LESSON XI: DETECTABLES, PRIORITIES AND THE COMPLETE SCENARIO

4.13.1 INTRODUCTION

This chapter will tell you about detectables and priorities in Brahms models, and will bring you to the completion of the Atm scenario.

4.13.2 TASK

Use detectables to interrupt, abort or then continue activities. The Atm machine will be waiting for inputs until keys are pressed on its pad or replies are received by the Bank computers. The student will be waiting for replies from the Atm. Use priorities: workframes that might be triggered under the same conditions will follow the traditional top/down, left/right order. To modify that order, you can use priorities.

Then, use these concepts to complete your Atm scenario! A scenario will be considered complete when it will reproduce the following:

Model a day in the life of a student. Students spend most of their time studying, but get hungrier as the time goes by (signaled by a clock). When they are particularly hungry (the threshold level varying with the student), they decide to move to one of the restaurants in town. Students choose the restaurant according to how much money they are carrying. If a student does not have enough money even for the cheapest restaurant, she will decide to pass first by the Atm of the bank where she has her account.

When she arrives at the Atm, the student inserts her bankcard and tries to remember the PIN associated to her account. The Atm allows its users 3 attempts to digit the correct PIN, before refusing the card altogether. The Atm communicates with the central bank computer to verify the correctness of the information provided by the user. If the bank computer communicates to the Atm that the PIN is correct and that the user has enough balance in her account, the Atm will dispense the cash and will print an invoice with the account number and the remaining balance.

Students need to have enough balance in their accounts to take out cash: if they attempt to take out more money than they have, the bank computer will notify the students (through the Atm) of the remaining amount of dollars in the account. The student will modify her approach accordingly, and take out just exactly the remaining dollars.

THE CAST (AGENTS and OBJECTS)

Students: Kim, Alex

Bank computers: Bank of America, Wells Fargo

Restaurants: Blakes, Raleighs

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

Studying Places: South Hall, Spraul Hall
Clock: the Campanile
Atms: one Atm for each bank

4.13.3 DESCRIPTION

4.13.3.1 DETECTABLES

A detectable is a mechanism by which, whenever a particular fact occurs in the world, an agent or object may notice it. The *noticing of the fact* may cause the agent or object to stop or to finish the workframe.

Two things occur in a detectable. First, the agent or object detects the fact and the fact becomes a belief of the agent or object. Second, *only* in the case of an agent, the beliefs of the agent are matched with the *condition* used in the detectable, and if there is a match the then-part of the detectable is executed, which may *abort* or *interrupt* the workframe. For objects, in the second step the facts in the world are matched with the detectable condition and if there is a match, the then-part is executed. These two steps are independent: Whether or not the fact is present in the world, the condition in the second step is tested. For example, if "the color of the telephone-1 is blue" is a fact, and a workframe contains the following detectable condition, "the color of the telephone-1 is red", in the first step an agent will obtain the belief "the color of telephone-1 is blue". In the second step, "red" would be compared with "blue" and the condition will *fail*, so the then-part of the detectable would *not* be executed.

The *action* or *then-part* of a detectable defines the *detectable type* and is one of five keywords: continue, abort, complete, impasse, and end-activity. *Continue* is the default: the agent or object detects conditions, but the workframe proceeds unaffected. With *abort*, a condition causes the agent or object to stop executing the workframe. With *complete*, a condition allows the agent or object to only perform the remaining consequences of the workframe, without doing the rest of the workframe's activities. With *impasse*, the condition prevents the continuation of the workframe *until* the condition is removed. In this case, the workframe goes into the *interrupted-with-impasse* state. *End-activity* is only meaningful when the detectable is in a composite activity: It does not detect facts, but causes the activity to be terminated immediately, based on matching the beliefs of the agent or object to the detectable condition. This allows an agent or object to abort working on composite-activities.

It is worth emphasizing that the detectable mechanism is operative for all workframes on the execution path of the agent or object's workframe-activity hierarchy (see Figure 18). This even holds for workframes that are in an interrupted or impasse state, so that a "whenever" detectable in any of those frames can detect a fact at any time.

Detectables can also be used to model *impasses*. A common example of an impasse is the case of inaccurate or missing information. Workframes may be written to handle impasses. For example, if a supervisor is ready for a technician but does not know the technician's telephone number, another workframe may lead the supervisor to look up the number.

With a detectable, an agent may notice *passive observables*, as when someone shouts, a fax machine beeps, or an agent is present vying for attention. Passive observables fall into two general classes: *sounds* and *visual states*. Objects that cause a sound—fax or phone—create the fact that represent the sound, which can then be detected. Sounds may persist over many simulation clock-ticks. Propagation into the surrounding space will recur as long as the object is making a sound. Propagation may be affected by geography.

When

For each detectable needs to 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:

For the detectable needs to be specified exactly when the detectable needs to be checked by specifying the time relative to the workframe instantiation's start the detectable needs to check the fact set and belief set.

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.

4.13.3.2 PRIORITIES

Activities can be assigned a priority. For example:

```
Move moveToLocation() {  
    [...]  
    priority: 2;  
}
```

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.

There are some interesting things to say about priorities and frames. Let us start with thoughtframes. When thoughtframes have no priority, the engine consider them as having priority 0. If more the one thoughtframe was available at the same time then it was up to the engine to decide which one wouldbe fired first, as a modeler you would have no control over that. The engine maintains a list of available thoughtframes ordered by priority. Whichever thoughtframe becomes available first will be fired first. Now that we have priorities you can control the execution sequence of thoughtframes if more then one thoughtframeis available at the same time. Whenever a thoughtframe becomes available the engine will retrieve the priority of the thoughtframe,if no priority is set, 0 wil be used. The engine will then add the thoughtframe to the list of available thoughtframes and keep the list sorted, highest priority first. Theoretically all available thoughtframes are fired at the same time. Since no 'at the same time' exists in the engine the engine will work through the list of available thoughtframes starting with the highest priority thoughtframe. It fires it and processes any concluded beliefs. Processing these beliefs could make some of the available thoughtframes unavailable and could make unavailable thoughtframes become available. The newly available thoughtframes are added to the list of available thoughtframes sorted on priority. The engine will keep getting the highest priority available thoughtframe and process it until all available thoughtframes are processed.

Once all thoughtframes are processed the engine will start checking the workframes again. The priority of workframes can be set in one of two ways. One is to set the priority on the workframe directly using the 'priority' attribute. The other method is what we had in the past is to have the engine determine the priority of the workframe by checking the priority of the activities. The engine will first check if the priority is set directly on the workframe, if sothat priority will be used. If no priority is set the workframe will get the priorities of the activities and use the priority of the activity with the highest priority in the workframe. If no activities are used in a workframe and no priority is set for the workframe then the priority 0 is used. Available workframes are also placed in a list sorted by priority. We also have the interrupted workframes list sorted by priority. The work selection algorithm determines what workframeis to be worked on next. The work selection algorithm retrieves the highest priority available workframe, the highest priority interrupted workframe and the workframe currently active (if one is active). Out of those three workframes the workframe with the highest priority will be selected to work on. If all three have the same priority, then the current work is selected. If both the interrupted work and available work have the same higher priority then the current work then the interrupted work will be selected (one tends to continue with the work that was started before starting new work). If the availablworkframe has the highest priority of all three then that workframe will be selected to work on.

4.13.4 SYNTAX

Syntax details are available at:

<http://www.agentisolutions.com/documentation/documentation.htm>

In particular, the concepts presented in this section are also discussed at http://www.agentisolutions.com/documentation/language/ls_att_det.htm, and pages linked from there.

4.13.5 TUTORIAL

Since you have done it till here, you should be able to complete this Lesson and the Atm scenario by yourself. A few hints will be provided about [detectables](#) and priorities and some new activities. More detailed information will have to be extracted from the online language specifications. You can of course check the provided code for comparison. However, try to find alternative and interesting ways to model the interaction between the student, the bank, and the Atm. Furthermore, now you should have the student go straight to the restaurant after getting cash out of the Atm.

Let us assume that you have modeled a `wait` workframe for the student, when he is waiting the reply from the Atm to know whether it will receive cash or not. You might have written something like the following:

```
workframe wf_waitAtmReply {
    repeat: true;
    variables:
        [...]
    detectables:
        detectable AtmRepliesYes {
            when(whenever)
            detect((at4.cashCanBeDispensed = true), dc:100)
            then abort;
        }
        detectable AtmRepliesNo {
            when(whenever)
            detect((at4.cashCanBeDispensed = false), dc:100)
            then abort;
        }
    when( [...] )
    do {waitAtmReply();
    }
}
```

This 'wait', even if the `waitAtmReply` is allowed to last for hours, will halt when an answer from the Atm is received. This is an example of how detectables can be used.

Be careful about detectables and objects. When an object detects a fact, it also creates a belief and then triggers the conclusions based on that belief. However, the detectable can also be triggered by a belief.

Note that objects react on facts and not beliefs. This raises an issue regarding when objects should react to things that are communicated to them. It might happen that a fact takes place and is communicated between objects – but the object receiving the communication has already ‘known’ about this change in the facts of the world (this will probably happen when you model the communications between the bank and the Atm). Using beliefs here is no solution, cause objects do not react on beliefs. This is one of those examples where different approaches are possible, depending very much on the interpretation you give to the model and its way of representing reality. You could decide, for example, to transform the objects into agents given their high degree of complexity and interaction with other agents. Or, you might want to adapt the code to take into consideration these situations (as it has been done in the files we provide as a ‘solution’ to the scenario). Or you might use dataframes, a new language construct recently added to the language (for more on this, as usual refer to the online language specifications).

For what relates to priorities, a possible way to use them is by inserting a parameter – say, `pri` – in the activity being considered, for example:

```
moveToLocation(Building loc, int pri) {  
    Location: loc;  
    Priority: pri;  
}
```

so that you can modify those priorities as the simulation goes on, when you call the activities from a workframe.

However, be careful when using priorities to control the sequence of activities. The sequence of activities is emergent in work practice and depends on the belief state of an agent. Priorities can be used in situations where one activity clearly has a higher priority than another activity, for example picking up a phone when it rings might get a higher priority than reading a document. In other case, priorities might be less appropriate. Priorities are probably best used when there exist some activities that can be performed at the same time, and you do not want to tie those workframes together artificially.

Figure 21 - A screenshot from the complete Atm scenario

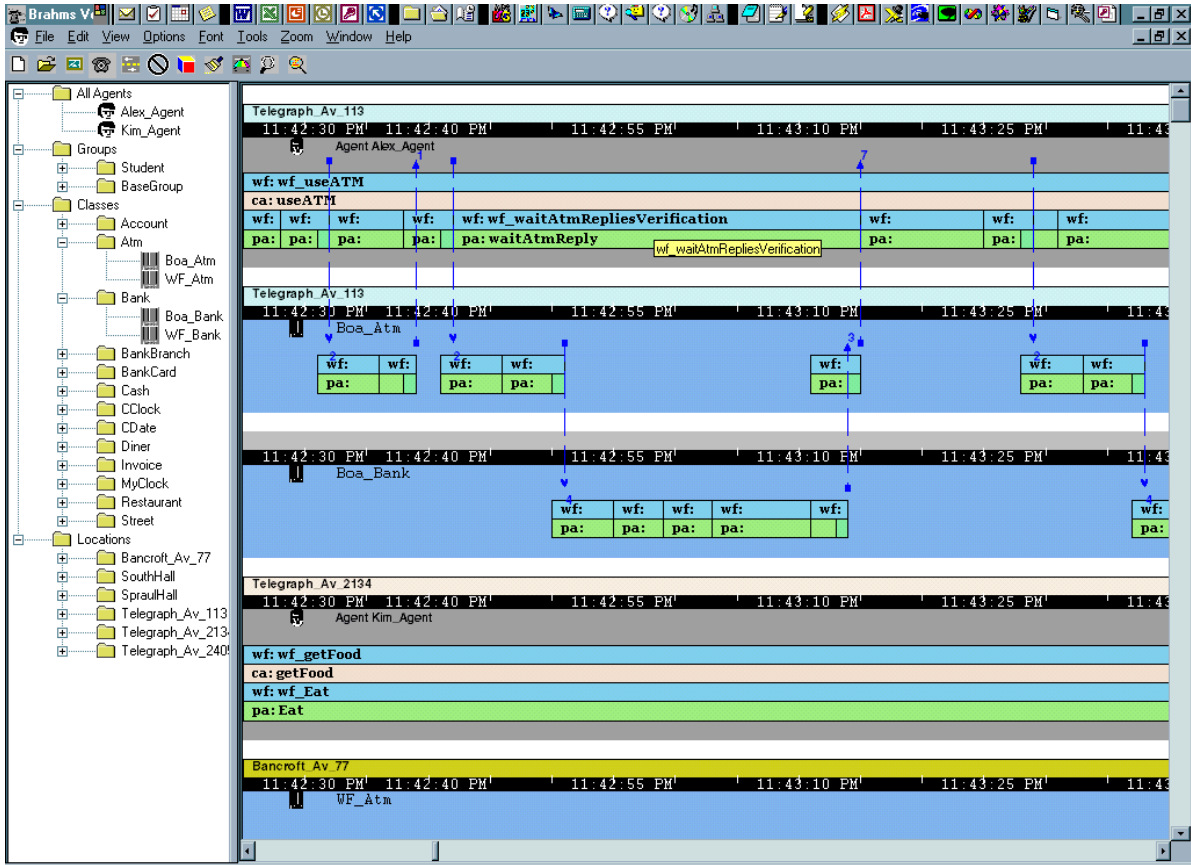


Figure 22 - A zoom in of the complete Atm scenario

You can also check and compare the final code for this scenario [here](#). Recall that the one provided is just one of the many ways you could code your model, and enjoy the completion of the Atm tutorial!

4.14 CONCLUDING ISSUES

4.14.1 HOW TO BUILD YOUR NEXT MODEL

The steps we have followed in this tutorial were appropriate to learn progressively more concepts of the Brahms language, but are not the best ones to build a new model when you have already mastery of those concepts. The Brahms design team suggests these steps for modelers who already know about Brahms structures and conventions:

1. Write a scenario of what the Brahms model is to model and define the objectives.
2. Go through the scenario and make a list of all concepts in the scenario, concepts like agents (people), artifacts (objects), areas (locations) and conceptual objects.
3. Go through the scenario and find all attributes that say something about the agents or objects in the scenario and list them with the appropriate agent or object.
4. Do the same for relations. Find relationship between agent, objects, areas, and conceptual objects, name them and assign them to one of the concepts you listed in 2.
5. Go through the scenario and make a list of all the activities performed by the agents and objects and place them with the agent or object for which the activity was defined in the scenario.
6. Next, make a classification hierarchy for all the concepts with their attributes, relations and activities. Like in object-oriented programming verify whether certain attributes should be more generic and should be listed higher in the hierarchy for re-use.

7. Once you have your hierarchy defined, start thinking about the conditions under which the activities are performed by again going back to the scenario. For each activity and for each agent or object define the conditions under which the activity will be performed. Also specify how long the activity should take and what states can be concluded when the activity is performed. Also think about the priorities of the activities. All this information will be used as the basis for defining your workframes. When defining you conditions you have access to the four precondition modifier (known, knownval, unknown, and not). Think also about making workframes generic by using variables. Use variables also to define how an agent or object will work on/with one or more variable bindings. As a final thought for activities think about whether the agent or object should be able to detect something in the world, i.e. see something happening while working on the activity (red light blinking on the phone) and define the exact state that is to be detected.
8. For measurement purposes you should associate resources with activities. The resource usage can in turn be used to do some statistical analysis to get answers to your measurement questions.
9. Once you have defined the workframes for each agent and object you can simulate your model and see what happens. Let the work process emerge and you might find interesting observations. Before you see the process emerge you most likely have to fine-tune your model. You might find that certain activities are not being performed due to conditions not being correct, activities not being performed because priorities are not set correctly, etc. This will be your debugging cycle. You will go back and forth between your Brahms model design, implementation and simulation results.

4.14.2 DEBUGGING TIPS

The Agent Viewer and its ExplanationFacility are the best places to start debugging your simulation (assuming you have one and that the Brahms Composer did not throw out errors when compiling the Brahms files).

In some cases, the Brahms Composer will work fine but the simulation run will never end. This might be due to the fact that the simulation is truly not supposed to end. But it might also be a loop or a composite activity with an 'hole' inside (a missing step where no activity inside the composite activity is not triggered). In these cases, you might halt the simulation manually if you have previously an (undocumented) flag `-ui` in the `bvm` bat file. This flag will let you halt the simulation in a smooth way, so that you can then use the Agent Viewer to see what was going on. Remember, however, that in some cases it might prove useful to go through the text file containing the simulation history itself, because it can be telling about what is going wrong.

You can also set your VM to have debug information as output. In the `vm.cfg` you modify the 'information' line by adding the 'debug'. Furthermore, you might also check the `Logs` directory in `AgentEnvironment` and open the log file. You might also open `eventinformation.txt`, and the event history `.txt` file itself that is going to be parsed by the Agent Viewer.

Some common problems are workframes continuously repeated, workframes not triggered (in which case you must find out which preconditions do not hold and make the conclusions be skipped), thoughtframes always repeating (recall that thoughtframes take no time, so you must be careful with their repetitions). You must also pay a lot of attention to the order at which things happen (or should happen). The VM checks all the workframes everytime an action or a conclusion takes place, to see which must be activated next. Sometimes an apparently trivial issue of ordering might cause the continuous repetitions of the same activity. Furthermore, composite activities must be used with attention: the simulation might get stuck if you do not have a coherent flow of procedures inside the composite activity itself. Finally, a simulation might never halt, as long as some workframe is active (a simulation might also never halt if the Virtual Machine has entered some kind of loop or impasse).

Remember how binding and beliefs are really crucial in workframes and thoughtframes (for what relates to beliefs, we are referring to agents here; objects act on facts). To use variables, you have to bind them with the preconditions (there are exceptions to this rule, and we will discuss them later in this section). But to evaluate the preconditions, your agents need beliefs. If your code is not working as you expect, try checking first if these crucial steps (beliefs and binding) have been coded correctly!

A final tip: remember to clean up the folder with the Brahms files you are working on from time to time, especially the bcc files - older versions might accumulate when you change the names of components, and create either confusion or compilation/simulation errors.

4.14.3 VALIDATION

Apart from debugging, you might want to know whether your simulation is really doing what you expect it to do. This is not an easy task when the simulation is complex and its behavior, by definition, unpredictable. The next chapter of this tutorial deals at length with these issues and is taken from Maarten Sierhuis's PhD dissertation.

4.14.4 FURTHER ISSUES AND EXERCISES

In this tutorial we have not covered all the aspects of the Brahms universe. If you want to know more, for example, about conceptual objects, java activities, and the Brahms real time components, you are invited to visit regularly the www.agentisolutions.com website. You might also want to consider the following developments for further exercises in the Atm scenario:

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

- what if each bank has more than one Atm? can you make the agent choose the closest one?
- can you model the objects (say, Banks and Atms) as agents? what will change? do you notice changes in the behavior related to detectables and communications?
- what if more students go to the same Atm at the same time? can you make the Atm handle just one order at a time, but the Bank central computer multiple orders coming from several Atms?
- can you make the agents randomly meet and interact (e.g., say hello each to the other)?
- what if you want the Atm to remember the number of the cards it has refused?

5. VALIDATION²²

In this chapter we will discuss issues related to the validation of your Brahms models: how do you know that they are doing what they should?

This is a more difficult question than what it might initially look like. Even when a model is compiling and is working, it might be doing something wrong. The more complex the model, the more difficult it is to validate. The Agent Viewer will be the best tool to help you validate a model. In this chapter, we will discuss what kind of considerations you should keep in mind when trying to use the Agent Viewer to validate your models.

5.1 MODELING WORK PRACTICE

Brahms was created to model and represent work practices. In order to use simulation as the method for understanding the work practice of an organization, we need a representational scheme that allows us to represent work practice, and a computational paradigm to simulate a model developed using the representational scheme. How can we achieve this result? This question leads to the main subsidiary question:

1. How can we model an organization's work practice in such a way that we include people's collaboration, "off-task" behaviors, multi-tasking, interrupted and resumed activities, informal interaction, knowledge and geography?

One answer to this question is to develop a modeling language and simulation program in which we can represent the way an individual or group of individuals work—i.e. their practice. This leads to further related questions:

- a. What is meant by the concepts in the question stated above?

Ethnographic fieldwork in the work place has shown that in looking at the way people work in practice we see a number of important aspects:

- (1) People collaborate with each other to accomplish what they have to do.
- (2) People often work on seemingly non-task related things, so called "off-task" behaviors.
- (3) People often work on more than one task at the same time, so called multi-tasking.

²² Source: Maarten Sierhuis' PhD Thesis.

- (4) People are often *interrupted* in their activities, and will *resume* what they were working on, after the interruption is over.
- (5) People have many interactions with others that were not planned before hand, and/or not part of the task at hand, so called *informal interactions*.
- (6) People use their domain knowledge, as well as their *social knowledge* about the organization and the culture to perform their daily work activities.
- (7) The environment is for most part a given. People are always situated in a three-dimensional space. Most of the time, people cannot change the work environment, and they can never ignore the constraints that the environment places on their activities.

Next, we need to operationalize these concepts. That is, to put them into a form in which they can be subject to testing by experiment. This leads to the following *operational questions*:

- b. What interpretation should be placed on these aspects of work practice?

In other words: How can we model them?

- c. How can these aspects be included in a computational modeling language?

In other words: What formal language can we create that makes it possible to simulate?

5.2 COMPUTATIONAL MODELS IN SIMULATION

We distinguish two aspects of a system, the *structural* aspect of the system and the *behavioral* aspect of the system. Computational models are models that show the behavioral aspects of a system, by *simulating* the behavior of the system over time. This is in contrast with static models, which only show the structural aspects, i.e. the system elements and their relations at one moment in time. As the complexity of a system increases, understanding how the system changes over time - its behavior - becomes increasingly difficult. This is especially true for non-linear systems. A computational model allows us to *observe* the result of changes in the system as time moves forward.

There are many ways in which computational models can be used in solving problems. However, we can classify the use of computational models in one of three ways (Table 3). *Descriptive models* are behavioral models of an existing system. The purpose of descriptive models is to describe the system in a way that makes us better understand the complexity of the system. Descriptive models are useful in analysis activities of complex dynamic environments. *Predictive models* are models that predict the way an existing system behaves in the future. The purpose of predictive models is to be able to know beforehand how the system will behave in the future. Such models are useful in tasks in which we need to make decisions based on future data from a complex dynamic environment. *Prescriptive models* are models of future—not yet existing—systems. The purpose of prescriptive models is to prescribe what a future system will or should look like. Such models are useful in design activities for complex dynamic environments.

Table 3. Use of Computational Models

Type of computational model	Use in problem domains
Descriptive model	Describe an existing system in order to understand it.
Predictive model	Predict the future of an existing system.
Prescriptive model	Prescribe a future system that does not exist yet.

5.3 TYPES OF MODELING SYSTEMS

Models help us to understand systems. There are four basic levels of knowledge about a system recognized by Klir (1985). At each level we know some important things about the system we did not know at lower levels (Table 4). The lowest level, the *source level*, identifies what part of the real world system we want to model, and the means by which we are going to observe it. It identifies the variables to measure and how to observe them. The next level, the *data level*, is the database of observations in terms of measurements of the variables from the source system. At the third level, the *generative level*, we have a model that can generate the data from the previous level. This is the level of system knowledge most people refer to as a *model* the system. At the fourth and highest level, the *structure level*, we have a description of the total system by coupling all generative components from the lower level together into a generative system for simulation.

Table 4. Klir's Levels of System Knowledge

Level	Name	System Knowledge
3	Structure	Components (at lower levels) coupled together to form a generative system, i.e. a simulation
2	Generative	Means to generate data in a data system

Printed on: This is an uncontrolled copy when printed.

3/31/11 3:08 PM Refer to the NX Brahms location for the latest version.

1	Data	Data collected from source system
0	Source	What variables to measure and how to observe them

Zeigler *et al* (2000), define three basic ways to deal with system problems, based on Klir's system knowledge levels; system analysis, system inference and system design. They allow us to move from one level of system knowledge to another. In *System analysis*, we try to understand the behavior of an existing or hypothetical system based on its known structure. *System Inference* is performed when we do not know the structure of the system before hand—we try to guess the structure from observations, allowing us to use this to predict future data. Finally, in *system design* we are investigating the alternative structures for a completely new system or the redesign of an existing system.

The important notion in Klir's levels of system knowledge is that in system analysis we are not generating new knowledge, as we move from a higher-level to a lower-level description of the system. In system analysis we are only making explicit what is implicit in the higher-level description. Klir does not consider this kind of subjective (modeler-dependent) understanding. Making something explicit that was implicit before, however, will lead to insight and understanding, which is a form of new knowledge. Even though in Klir's sense system analysis might not generate new knowledge, interesting properties of the system will come to light of which we were not aware before the analysis.

In both system inference and system design we move from lower levels to higher levels of system knowledge. Therefore, in these activities we are creating new knowledge that did not exist before, according to Klir's definition.

In Table 5, Zeigler's fundamental system problems are related firstly to the transitions in terms of Klir's levels, and secondly to the types of computational models that we are developing at the generative level. When we are in a system analysis activity we are developing a *descriptive model* of the system. The development of a descriptive computational model leads to an increased understanding of how the system works. In system inference we are trying to create a *predictive model*. Predictive in the sense that once we have created a computational model that can explain the generation of observed data, we can now use this model to predict future data of the system not yet observed. In system design we are developing a *prescriptive model*, in the sense that the model prescribes a future system.

Table 5. System problems related to model use and types

System problems	Model use	Transition between Klir's levels	Type of computational model
System Analysis	The system exists, and we try to understand its behavior.	Moving from a higher to a lower level of description, e.g. using information at the generative level to	<i>Descriptive model</i>

		<i>generate</i> the source data at the data level.	
System Inference	The system exists, and we try to infer how it works from observations of its behavior.	Moving from a lower to a higher level, e.g. having data and trying to find a means to generate it.	<i>Predictive model</i>
System Design	The system does not yet exist in the form we're contemplating, and we try to come up with a good design for it.	Moving from a lower to a higher level, e.g. having a means to generate data based a design at the generative level.	<i>Prescriptive model</i>

5.4 VERIFICATION AND VALIDATION

Let us define the concepts *verification*, *validation*, and to be complete, *credibility* as follows:

- *Verification* is the process whereby the modeler asks if the model is performing as it was designed. In this step in the V & V process, the objective is to determine if the logic of the computer model correctly implements the assumptions made in the conceptual model.
- *Validation* is the process whereby the modeler asks how accurately the model is representing reality. Here the objective is to determine whether the alternative hypothesis (the model does not represent reality) holds or not.
- A *credible* model is one that the client accepts as being valid enough to use in making decisions. It should be noted that in this experiment we do not have a client that will make such a credibility judgement. In some sense the reader is the one that will make a credibility judgement about the model representing the work practices of the Apollo 12 astronauts.

5.4.1 THE PURPOSE OF VERIFICATION AND VALIDATION

An important part of modeling and simulation is the verification and validation (V & V) of the model and the results of the simulation. Without a thorough V & V there is no ground in having any confidence in the model and the results of the simulation. Although it is important to realize that it is impossible to prove that a model is a general valid model. The reason for this is the fact that:

1. A model is only valid with respect to its purpose. For instance, a model that has been created for the purpose of predicting the future state of a system might not be valid as a prescriptive model of the current system.

2. There are different interpretations of the real world possible. Depending on our worldview, or *Weltanschauung*, we have a different interpretation of the real world and therefore, of the model and its validity .
3. The data used to develop the model may be inaccurate. Even if that is not the case, it should be realized that the data used and the data generated by the simulation are but a small data sample. Therefore, they can only be seen as a probabilistic answer and not a definitive one.

The conclusion is that, although in theory a model is either valid or invalid, in practice it is not easy and often not possible to prove that a model is valid. Therefore, we have to think in terms of the confidence we can place in the model. The V & V of the model in this experiment is not one of demonstrating that the model is correct, but in contrast it is a process of *falsification* of the model, i.e. demonstrating that the model is incorrect. In so doing, the purpose of V & V is to increase the confidence in the model, even though we might find inconsistencies and problems with the model according to the real-world data.

5.4.2 THE VERIFICATION AND VALIDATION PROCESS

Many authors have described the process of a successful simulation . All of them mention a series of processes that need to be followed. The high-level processes are shown in Figure 23, which is borrowed from Robinson (1999). A simulation study first starts with understanding the *real world*, as well as the problem to be tackled. When the real world is sufficiently understood the modeling activity starts, and a *conceptual model* is described. After this, the model can be coded into a computer model, in this case the Brahms language. When the model is complete, experiments are run to develop *solutions* to the real-world problem being handled. In this case, a greater *understanding* of the real world was obtained. In real-world projects it is hoped that the solutions found in the experiments can be implemented in the real world, or that the better understanding of the problem will lead to better decision making. Even though there is a natural sequence to following these steps, it is obvious that the real process is not strictly sequential, and that several iteration through the steps are necessary.

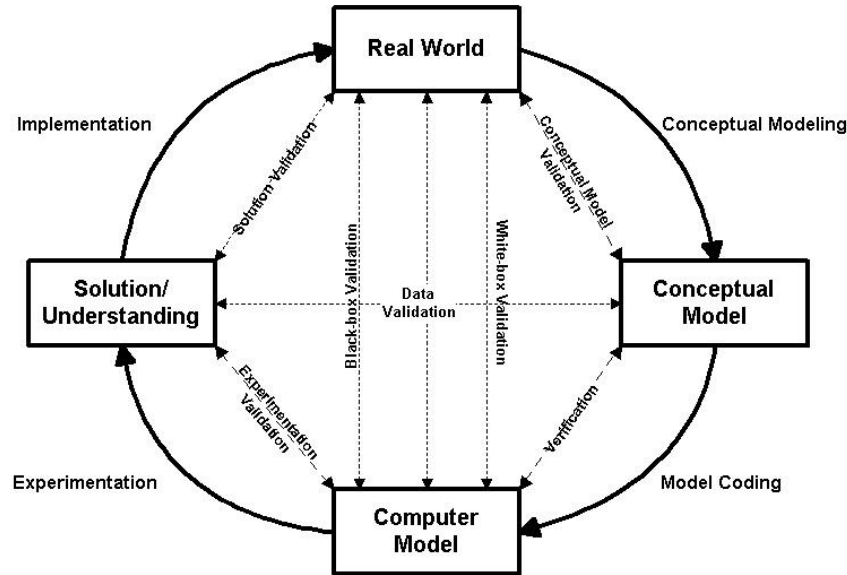


Figure 23. Simulation model verification and validation in the modeling process (borrowed from (Robinson, 1999))

5.4.3 DATA VALIDATION

As is shown in Figure 23, data validation is important at every step of the simulation process, because at each step in the process you use data. It can thus be said that, if the simulation data is verified against the original scenario's data, and it can be shown that the outcome is correct in relation to this data, the validity of the simulation model is high.

5.4.4 CONCEPTUAL MODEL VALIDATION

The purpose of the conceptual model validation is to determine that the scope and level of detail of the proposed model is sufficient, and that all assumptions are correct. To describe this validation, let me take a step back and restate the real-world problem I addressed in this study. The problem in this study was that of *showing that the Brahms modeling and simulation language is powerful enough to describe the work practices*. The level of model detail that is needed to test this hypothesis is given by the definition of what to include in a model of work practice.

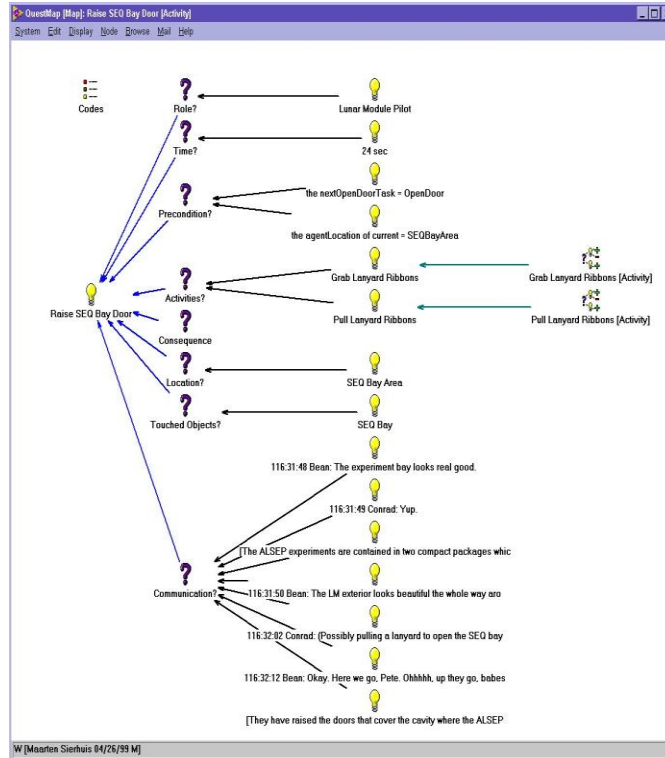


Figure 24. An Example of Conceptual Model for the Simulation of an Apollo Mission

If we take as a given the aspects of work practice, then we can validate that these aspects are indeed included in the model. Therefore, the validation method we can use for the conceptual model is to analyze the important aspects of modeling work practice, as described in the theory, and to make sure that the conceptual model included all of them. Computer model verification

The next phase in the modeling process is the design and implementation of the Brahms model source code. In this phase, the modeler needs to translate the activities, groups, agents, classes and objects represented in the conceptual model into the Brahms language. To do this, the modeler needs to be proficient in the Brahms language, and specifically in the multi-agent and activity programming concepts in Brahms. For first time Brahms modelers this is a painstaking process, and is similar to the compile-debug cycle in traditional programming languages, such as C++ or Java.

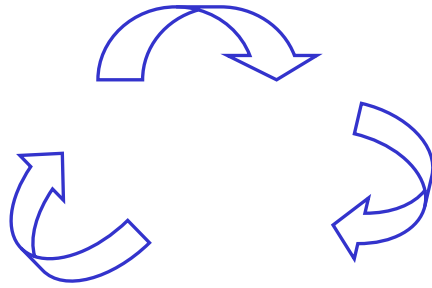


Figure 25. Brahms Compile-Debug Cycle

shows the modeling cycle, which first continues until the complete model can be compiled without syntax errors by the Brahms Compiler. However, verifying the model is more than getting the Brahms Compiler to compile the model without syntax errors. Although this is of course a first and important step in the process, the most important step is to compare the "functioning" of the model with the conceptual model. The model validation and verification steps are driving the Brahms model development process, shown in Figure 26.

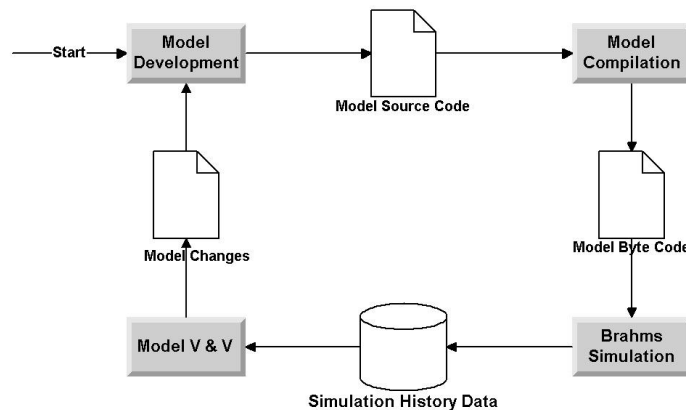


Figure 26. Brahms Model Development Cycle

The functioning of the model is visually verified using the *Agent Viewer* application. Using the Agent Viewer it is possible to visually inspect the simultaneous behavior of the agents and objects, and compare the expected behavior from the conceptual model with the actual behavior during the simulation.

5.4.5 EXPERIMENTATION VALIDATION

1. Comparing the model output to data from the real system is the most objective and scientific method of validation. Of course, this type of validation can only be performed if there is a real system, and real-world data that correspond to the simulation parameters are available – for example, historical data available to validate our model.

5.4.5.1 WHITE-BOX VERSUS BLACK-BOX VALIDATION

We consider two types of real-world data validation, *white-box* and *black-box* validation. The model verification described in section 0 is considered a white-box validation. Validating the simulated activity times with the timing of the activities based on the transcript of the voice loop communication is also a type of white-box validation. However, the second validation, that of the actual voice loop data, is considered a *black-box validation*.

White-box validation is a *micro validation of the content* of the model. In a white-box validation we try to validate the model by investigating the model content in detail. The purpose of this type of validation is to ensure that the content of the model is true to the real world. The use of a graphical visualization and spreadsheet tools are very appropriate in this type of validation.

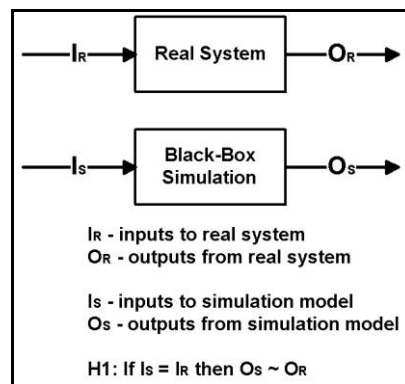


Figure 27 Black-box validation: comparison with the real system (from (Robinson, 1994))

In a black-box validation we are not looking inside the model, but we are *validating the overall behavior of the model* with the output of pre-specified real-world data. In this type of validation we need to validate that when we specify input data to the simulation model that is similar to that of the real system, the output data from the simulation should be relatively similar to that of the real system as well. This is a validation of the alternative hypothesis H1 (Figure 27).

6. INDEX

account2-6, 2-7, 2-9, 2-10, 4-34, 4-58, 4-63, 4-72, 4-88, 4-94, 4-97, 4-112, 4-113, 4-134, 4-136
activities2-5, 2-7, 2-10, 2-11, 2-12, 3-28, 4-35, 4-49, 4-50, 4-51, 4-52, 4-55, 4-57, 4-64, 4-65, 4-66, 4-67, 4-68, 4-69, 4-73, 4-76, 4-82, 4-83, 4-86, 4-88, 4-89, 4-90, 4-93, 4-94, 4-95, 4-97, 4-100, 4-101, 4-102, 4-103, 4-105, 4-108, 4-113, 4-116, 4-117, 4-118, 4-119, 4-120, 4-121, 4-124, 4-125, 4-126, 4-127, 4-128, 4-130, 4-132, 4-133, 4-134, 4-136, 4-137, 4-139, 4-142, 4-145, 4-146, 4-147, 5-149, 5-150, 5-151, 5-152, 5-156, 5-158
Activities2-5, 4-52, 4-66, 4-67, 4-68, 4-69, 4-100, 4-117, 4-133, 4-139
agency 4-49
Agent Viewer3-14, 3-24, 3-26, 3-28, 3-30, 4-79, 4-80, 4-81, 4-82, 4-99, 5-149
AgentiSolutions 3-31
Agents2-5, 2-11, 4-42, 4-49, 4-50, 4-52, 4-59, 4-61, 4-126
AgentViewervi, 3-14, 3-17, 3-25, 3-27, 3-28, 3-31, 4-81, 4-82, 4-87, 4-143, 4-147, 5-149
area2-6, 4-37, 4-41, 4-42, 4-44, 4-45, 4-48, 4-67, 4-74, 4-102
Atm2-11, 3-15, 4-34, 4-74, 4-97, 4-103, 4-125, 4-134, 4-136, 4-144
Atm Files 3-15
attributes2-7, 2-9, 4-35, 4-42, 4-48, 4-49, 4-50, 4-51, 4-52, 4-53, 4-55, 4-56, 4-57, 4-61, 4-62, 4-75, 4-88, 4-90, 4-91, 4-92, 4-93, 4-94, 4-95, 4-97, 4-103, 4-116, 4-120, 4-134, 4-145
bank2-6, 2-7, 2-9, 2-11, 4-34, 4-56, 4-58, 4-63, 4-88, 4-92, 4-93, 4-94, 4-95, 4-97, 4-108, 4-112, 4-120, 4-125, 4-134, 4-136, 4-137, 4-142, 4-148
Beliefs 2-5, 4-52, 4-58, 4-60, 4-61, 4-102
Brahms Builder 3-17, 3-24, 4-55, 4-146
Brahms Compiler 3-14
Brahms concepts 2-4, 4-36, 4-39, 4-40
BrahmsBuilder.bat 3-23, 3-30
Bvm 3-26
Classes 2-5, 4-88, 4-89
communication2-5, 2-11, 4-62, 4-100, 4-101, 4-102, 4-126, 4-132, 4-142, 5-158
Compilation Unit 4-37
composite2-5, 4-66, 4-67, 4-68, 4-76, 4-83, 4-101, 4-108, 4-117, 4-119, 4-120, 4-121, 4-124, 4-127, 4-131, 4-132, 4-137, 4-139, 4-146, 4-147
consequences2-5, 4-75, 4-100, 4-107, 4-108, 4-127, 4-137, 4-139
debugging 4-146, 4-147
Debugging 3-31
Detectable 4-65, 4-138
detectables4-62, 4-65, 4-66, 4-89, 4-108, 4-127, 4-131, 4-136, 4-141, 4-148
Facts 2-5, 4-52, 4-58, 4-59, 4-61
first-order predicate 4-52, 4-59
Geography 2-5, 2-12, 4-41
Groups 2-5, 4-49, 4-50
import 2-6, 3-23, 3-26, 4-37, 4-38, 4-40
Index 3-33, 6-159
Installation v, 3-13, 7-161
Intended Audience 1-2
invoice 2-6, 4-34, 4-136
Models 2-11, 3-22, 5-151
movement 2-5, 4-44, 4-84
object-oriented 1-1, 2-6, 2-10, 4-49, 4-89
Objects 2-5, 2-11, 4-61, 4-88, 4-92, 4-138
OOSee object-oriented, See object-oriented, See object-oriented, See object-oriented, See object-oriented
Overview 2-4, 3-13
package2-7, 3-23, 4-37, 4-38, 4-39, 4-40, 4-44, 4-54, 4-55, 4-56, 4-60, 4-91, 4-92, 4-93, 4-94, 4-95, 4-96, 4-103, 4-104
PIN 2-6, 4-34, 4-136
preconditions2-5, 4-62, 4-65, 4-66, 4-69, 4-71, 4-72, 4-75, 4-83, 4-84, 4-86, 4-92, 4-97, 4-100, 4-108, 4-110, 4-111, 4-113, 4-116, 4-119, 4-127, 4-128, 4-129, 4-130, 4-132, 4-147

Printed on: This is an uncontrolled copy when printed.

3/31/11 3:08 PM Refer to the NX Brahms location for the latest version.

primitive2-5, 2-7, 4-64, 4-67, 4-68, 4-69, 4-100, 4-101, 4-102, 4-103, 4-105, 4-117, 4-118, 4-119, 4-121, 4-132, 4-133
priorities4-35, 4-66, 4-68, 4-83, 4-86, 4-119, 4-130, 4-131, 4-132, 4-136, 4-139, 4-141, 4-142, 4-146
Private 4-53, 4-90
Protected 4-53
Public 4-53
random 4-35, 4-69, 4-76, 4-103, 4-109, 4-118, 4-125
relations2-7, 2-9, 4-35, 4-40, 4-42, 4-49, 4-51, 4-52, 4-55, 4-56, 4-75, 4-88, 4-90, 4-92, 4-93, 4-94, 4-95, 4-97, 4-112,
4-113, 4-116, 4-134, 4-145, 5-150
restaurant2-6, 2-11, 4-34, 4-58, 4-64, 4-74, 4-75, 4-84, 4-88, 4-97, 4-98, 4-99, 4-100, 4-105, 4-107, 4-108, 4-113, 4-
116, 4-124, 4-136
Simulation Engine 3-14
student2-6, 2-7, 2-10, 2-11, 4-34, 4-45, 4-47, 4-49, 4-56, 4-63, 4-74, 4-75, 4-105, 4-111, 4-112, 4-121, 4-125, 4-134,
4-136, 4-141
Support 3-31
thoughtframes2-5, 2-8, 2-11, 4-35, 4-50, 4-51, 4-52, 4-53, 4-55, 4-57, 4-84, 4-90, 4-92, 4-100, 4-101, 4-104, 4-108,
4-127, 4-147
variables2-8, 4-35, 4-65, 4-66, 4-68, 4-69, 4-70, 4-72, 4-74, 4-85, 4-93, 4-97, 4-98, 4-100, 4-101, 4-102, 4-107, 4-
108, 4-109, 4-110, 4-111, 4-112, 4-113, 4-114, 4-115, 4-116, 4-121, 4-122, 4-123, 4-124, 4-127, 4-141, 4-146, 5-
151, 5-152
Venn diagram 4-60
Virtual Machine3-13, 3-16, 3-17, 3-25, 3-31, 4-147, *See* Simulation Engine
workframes2-5, 2-8, 4-35, 4-50, 4-51, 4-52, 4-53, 4-55, 4-57, 4-62, 4-64, 4-65, 4-66, 4-68, 4-69, 4-74, 4-76, 4-82, 4-
83, 4-86, 4-89, 4-90, 4-93, 4-97, 4-100, 4-103, 4-105, 4-106, 4-113, 4-117, 4-118, 4-119, 4-120, 4-121, 4-127, 4-
131, 4-132, 4-133, 4-136, 4-137, 4-142, 4-146, 4-147
Workframes 2-5, 4-52, 4-64, 4-65, 4-66, 4-138
xml 3-22, 3-24, 3-25, 4-48, 4-77

Printed on:

This is an uncontrolled copy when printed.

3/31/11 3:08 PM

Refer to the NX Brahms location for the latest version.

7. REFERENCES AND OTHER LINKS

Brahms Development Team (1999-). *Brahms TM99-0008 - Brahms Language specification*, http://agentisolutions.com/documentation/language/ls_title.htm

Brahms Development Team (1999-). *Brahms Installation readme.txt*, <http://agentisolutions.com/download/download.htm>

Brahms Development Team (2001). *Brahms TM01-0002 – Brahms Tutorial Lite*, http://agentisolutions.com/documentation/tutorial/tt_title.htm

Klir, G. J. (1985). *Architecture of Systems Complexity*, Saunders, New York.

Robinson, S. (1994). *Successful Simulation: A Practical Approach to Simulation Projects*, McGraw-Hill, Maidenhead, UK.

Robinson, S. (1999). "Simulation Verification, Validation and Confidence: A Tutorial." *Transactions of The Society for Computer Simulation International*, Vol. 16(Number 2):63-69.

Sierhuis, M. (2001). *Modeling and Simulating Work Practice: Brahms, A multiagent modeling and simulation language for work systems analysis and design*, PhD thesis, University of Amsterdam, <ftp://www.agentisolutions.com/anonymous/MXStthesis/PrintVersion/>.

Zeigler, B. P., H. Praehofer, and Kim, T. G. (2000). *Theory of Modeling and Simulation*, Academic Press, San Diego, CA.