USING ABSTRACT STATE MACHINES TO
MODEL A GRAPHICAL USER INTERFACE
SYSTEM

by

Ming (Mike) Su
B.Sc., University of Ottawa, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Ming (Mike) Su 2006
SIMON FRASER UNIVERSITY
Spring 2006

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.
APPROVAL

Name: Ming (Mike) Su

Degree: Master of Science

Title of thesis: Using Abstract State Machines to Model a Graphical User Interface System

Examinining Committee: Dr. Jiangchuan Liu
Chair

Dr. Uwe Glässer, Senior Supervisor

Dr. Arthur (Ted) Kirkpatrick, Supervisor

Dr. Fred Popowich, SFU Examiner

Date Approved: Apr 4, 2000
DECLARATION OF PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection, and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada
Abstract

A graphical user interface (GUI) system is a visual tool for users to operate computer applications. In the software engineering world, verifying that the functions of a GUI system satisfy the perspective of users is one important goal. System modeling provides an opportunity to verify the functionality of the system before implementing it.

In this thesis, we model the GUI system of the CoreASM language debugger based on the abstract state machine (ASM) paradigm, and give a formal specification to the GUI system. This GUI system model provides a formal mathematical foundation to specify the architecture and the function form of the GUI system and to specify the interactive actions between the users and the computer application (the CoreASM engine). The design approach in this work incorporates both object-oriented and task-oriented approaches. A process of level-wise refinement is used to solve particular design problems.
This thesis is dedicated, with love, to my parents and my brother!
"Life is finite, while knowledge is infinite."

— Zhuang Zi
Acknowledgments

Many thanks to all the members of my committee for their guidance and thanks to colleagues working in the CoreASM project for discussion and feedback.

In particular, I would like to thank the following:

Dr. Uwe Glässer, whose guidance enabled me to complete this thesis.

Dr. Arthur (Ted) Kirkpatrick, whose advices assisted me to think over my work from expectations as an HCI reader.

Mr. Roozbeh Farahbod and Mr. Mashaal Memon. It is much appreciated in the discussions of the project and to work together.

Additional, I thank my parents and my brother for their patience and supports.
# Contents

Approval ii  
Abstract iii  
Dedication iv  
Quotation v  
Acknowledgments vi  
Contents vii  
List of Tables xii  
List of Figures xiii  
List of Specs xiv  
Abbreviations xvi

## 1 Introduction 1  

1.1 Motivation and objectives 2  
1.2 Thesis organization 3  
1.3 Related works 4  
1.4 Significance of work 5
## 2 Human computer interactive systems

- 2.1 General introduction to HCI .................................. 7
- 2.2 Graphical user interfaces ......................................... 8
- 2.3 User interface management systems .......................... 9
- 2.4 Scenarios .......................................................... 13
- 2.5 Task analysis in HCI .............................................. 14
  - 2.5.1 Introduction to task analysis .............................. 14
  - 2.5.2 Hierarchical task analysis .................................. 15
  - 2.5.3 Use case task analysis ...................................... 16
- 2.6 Software development life cycle in HCI ..................... 17
- 2.7 System design in HCI ............................................. 18

## 3 Abstract state machines

- 3.1 Abstract state machines ......................................... 21
- 3.2 Some definitions used in ASMs ................................. 22
  - 3.2.1 Universe ...................................................... 22
  - 3.2.2 Function ..................................................... 22
  - 3.2.3 Vocabulary .................................................. 24
  - 3.2.4 State .......................................................... 25
  - 3.2.5 Update set .................................................. 25
  - 3.2.6 Step and run ................................................ 25
  - 3.2.7 Rule and program .......................................... 26
- 3.3 Parallel abstract state machines ............................... 27
- 3.4 Non-deterministic abstract state machines ................. 27
- 3.5 Distributed abstract state machines ........................ 28
- 3.6 Turbo abstract state machines ................................ 29
- 3.7 Summary .......................................................... 30

## 4 Formal modeling approaches

- 4.1 Formal modeling approaches in GUI architecture and functionality design 31
- 4.2 A specification approach of GUI modeling in the CoreASM project 32
## 4.3 Summary

### 5 The GUI ASM model in the CoreASM project

- **5.1** Architecture of the CoreASM supporting tool environment .................................. 35
- **5.2** Actors in the GUI ASM model ............................................................... 38
- **5.3** Architecture of CASM_GUI ........................................................................ 40
  - **5.3.1** Components of CASM_GUI .................................................................. 40
  - **5.3.2** Communication structure in CASM_GUI .............................................. 44
- **5.4** Activities in CASM_GUI ............................................................................. 48
  - **5.4.1** Activity specifications ........................................................................... 49
- **5.5** Interfaces in the GUI ASM model ............................................................ 57
  - **5.5.1** Interfaces in the GUI ASM model ...................................................... 57
  - **5.5.2** The GUI ✯ User interface ..................................................................... 59
  - **5.5.3** The GUI ✯ Engine interface ................................................................. 59
- **5.6** Conclusion ................................................................................................... 61

### 6 Discussion

- **6.1** Four abstraction levels for the GUI ASM model .......................................... 63
  - **6.1.1** The first abstraction level - (the features) ........................................... 64
  - **6.1.2** The second abstraction level - (the interface design) ......................... 64
  - **6.1.3** The third abstraction level - (the internal architecture) ....................... 65
  - **6.1.4** The fourth abstraction level - (the concurrency problem) ................. 66
  - **6.1.5** An example of refining a model in four abstraction levels .......... 66
  - **6.1.6** Benefit of refinement by levels ............................................................ 74
- **6.2** Concurrency issues in the GUI system ....................................................... 75
  - **6.2.1** GUI ASM model constraints in concurrency issue ......................... 75
  - **6.2.2** GUI ASM model components .............................................................. 77
  - **6.2.3** Activities in ControlPanelProgram ..................................................... 78
- **6.3** Summary ..................................................................................................... 81

### 7 Implementation

- **7.1** Visual CoreASM language debugger ........................................................... 84
7.1.1 Function areas on the GUI.................. 84
7.1.2 Information organized in a tree structure .... 86
7.1.3 Run view.................................. 87
7.1.4 Control panel.............................. 88
7.2 Experiment of the ATM model................ 88
7.2.1 ATM model.................................. 89
7.2.2 Experiments................................ 90
7.2.3 Conclusion................................ 91

8 Conclusion..................................... 96
8.1 Conclusion.................................... 96
8.2 Future work................................... 97

A List of terms used in the GUI ASM model.... 99
A.1 Actors in the GUI ASM model................. 99
A.2 Components of CASM_GUI..................... 99
A.3 Signal pools in the components.............. 100
A.4 Activities in CASM_GUI....................... 100
A.5 Signals in CASM_GUI.......................... 102
A.5.1 USER_ACTIVITYREQUEST................... 102
A.5.2 PROGRAMV_COMMAND....................... 103
A.5.3 MESSAGEV_COMMAND......................... 103
A.5.4 OUTPUTV_COMMAND.......................... 104
A.5.5 RUNV_COMMAND............................. 104
A.5.6 STATEV_COMMAND........................... 104
A.5.7 UPDATESETV_COMMAND....................... 104
A.5.8 HISTORYV_COMMAND.......................... 105
A.5.9 VOCABV_COMMAND...........................
List of Tables

5.1 Components in CASM_GUI ........................................ 44
5.2 Signal pools in components ........................................ 46
5.3 Activities and signals to activate these activities ............ 50
6.1 Asynchronous activities and synchronous activities ........... 79
7.1 The components and the function areas in where the views of these components are ........................................ 86
A.1 Signal pools in components ........................................ 100
# List of Figures

2.1 The *Model-View-Controller* model ........................................... 10
2.2 The *Presentation-Abstraction-Control* model ............................ 11
2.3 User Interface (UI) design with an ASM approach ......................... 12
2.4 An example of scenario notation .............................................. 13
2.5 Hierarchical task analysis ....................................................... 15
2.6 Software development process: the waterfall model ...................... 17
2.7 Discussion of system specification model .................................. 19

5.1 CoreASM supporting tool environment architecture ......................... 37
5.2 Actors in one use case .......................................................... 39
5.3 Internal components of CASM_GUI ........................................... 42
5.4 GUI ↔ Engine interface ......................................................... 61

7.1 The graphical user interface of the visual CoreASM language debugger 85
7.2 The view of a state ............................................................... 87
7.3 The run view ......................................................................... 87
7.4 The control panel .................................................................... 88
7.5 The states in a successful run .................................................. 93
7.6 The output in the second step in a successful run .......................... 94
7.7 The states of an inactivated ATM ............................................. 95
List of Specs

5.1 Features of message view: display messages and clear messages ........ 43
5.2 The components in CASM_GUI ........................................... 45
5.3 Specification of a signal pool ............................................ 45
5.4 Choosing a signal from the signal pool of message view ............... 46
5.5 Sending a signal .......................................................... 47
5.6 Activity and sub-activity .................................................. 49
5.7 Controlled function example ............................................ 58
5.8 Monitored function example ............................................. 58
5.9 Activity commands ........................................................ 59
5.10 Rules in the GUI ↔ User interface .................................. 60
5.11 Rules and functions in the control interface of GUI ↔ Engine interface 60
5.12 Rules in the access interface of GUI ↔ Engine interface ............ 62
6.1 The activity ForwardRunActivity ...................................... 67
6.2 The activity ForwardRunActivity is activated in the main rule ....... 67
6.3 The actors in the activity ForwardRunActivity ....................... 68
6.4 The sequential execution of sub-activities in the activity ForwardRun-
  Activity ............................................................................. 68
6.5 The sub-activity getNumForwardStepsRequestedSA .................. 69
6.6 The sub-activity forwardRunSA ......................................... 69
6.7 The sub-activity getFRFeedbackSA ..................................... 69
6.8 API functions used in the activity ForwardRunActivity ............ 70
6.9 The activity ForwardRunActivity is activated in the control panel .. 71
6.10 Sending requests to other components in activity ForwardRunActivity 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.11</td>
<td>The component state view</td>
</tr>
<tr>
<td>6.12</td>
<td>The constraint to activate the activity <em>ForwardRunActivity</em></td>
</tr>
<tr>
<td>6.13</td>
<td>State of the activity <em>ForwardRunActivity</em></td>
</tr>
<tr>
<td>6.14</td>
<td>The activation of an activity</td>
</tr>
<tr>
<td>6.15</td>
<td>The parallel executing components</td>
</tr>
<tr>
<td>6.16</td>
<td>The constraint to activate an asynchronous activity</td>
</tr>
<tr>
<td>6.17</td>
<td>The constraint to activate a synchronous activity</td>
</tr>
<tr>
<td>6.18</td>
<td>Synchronous activities and asynchronous activities</td>
</tr>
<tr>
<td>7.1</td>
<td>The ATM model</td>
</tr>
</tbody>
</table>

7.1 The ATM model
Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>application program interface</td>
</tr>
<tr>
<td>ASM</td>
<td>abstract state machine</td>
</tr>
<tr>
<td>DASM</td>
<td>distributed ASM</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>HCI</td>
<td>human-computer interaction</td>
</tr>
<tr>
<td>JFC</td>
<td>Java Foundation Classes</td>
</tr>
<tr>
<td>MVC</td>
<td>model-view-controller</td>
</tr>
<tr>
<td>PAC</td>
<td>presentation-abstraction-control</td>
</tr>
<tr>
<td>UI</td>
<td>user interface</td>
</tr>
<tr>
<td>UIDS</td>
<td>user interface development system</td>
</tr>
<tr>
<td>UIMS</td>
<td>user interface management system</td>
</tr>
<tr>
<td>UML</td>
<td>unified modeling language</td>
</tr>
<tr>
<td>VM</td>
<td>virtual machine</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This thesis introduces a model for the graphical user interface application created for the CoreASM project, and gives the formal specification for it. As part of the project, the Software Technology Lab at Simon Fraser University is developing an executable working environment [Section 5.1] for the CoreASM language, a language that extends from pure abstract state machine principles [14]. CoreASM is still under development in the Software Technology Lab. The graphical user interface application is one tool in the working environment of the CoreASM language. It provides for interactive visualization and control of CoreASM simulation runs.

A graphical user interface application is one type of human-computer interaction system. Human-computer interaction is a huge research field. Computer science, psychology, information science, and engineering all influence research in this field. This field focuses on the interactions between computer systems and human users. Designing computer systems to better present information and to improve user work performance are some of the goals of this research work.

The CoreASM language comes from the original, basic definition of abstract state machines given in the Lipari Guide [18]. Abstract state machines are practical for modeling hardware and software architectures, programming languages, network protocols, etc. Some applications of ASM-based modeling include the ITU-T standard
for SDL [19], the IEEE language VHDL [5], the programming languages Java [28] and C# [4], and communication architectures [15] [16].

1.1 Motivation and objectives

Abstract state machine (ASM) methodologies have been proven practical for modeling complex systems. Application examples mostly involve existing systems. One way to make ASMs practical in industry is to use ASM methodologies to model predictively during the design phase, rather than for analyzing systems that already exist.

ASM methodologies, such as ground model [8][6] and stepwise refinement [7][30], can be applied in predictive modeling as well. We need tools to apply these practical methodologies to industrial designs. The CoireASM project is a step toward applying ASMs in industry. The project provides a supporting tool environment to make ASMs executable. Software engineers can do high-level design, experimental validation, and formal verification of abstract system models in the early design phases with the CoreASM supporting tool environment.

The graphical user interface application presents execution information for abstract system models. It is a component of the CoreASM supporting tool environment [Section 5.1]. The purpose of this thesis is to apply ASM methodologies in designing an interactive system, specifically to the graphical user interface application of the CoreASM language debugger, using the model-based design approach. The formal specification of the application is given as the part of the design documents. The formal specification describes the application on three aspects, the architecture, the functionality and the interfaces between the user and the graphical user interface (GUI) application and the underlying CoreASM engine.

The first task of this thesis research is to apply a specification approach based on the ASM paradigm to formally specify the architecture and the functionality of the new user interface system. The graphical user interface application did not exist prior to the project. The development of the application requires predicting the application features. These features can be captured and be documented as informal user requirements. The model in the thesis formally specifies these informal user
requirements. After the application is built, the model can be used to verify the prediction on the application features, and to validate the interfaces (APIs) between systems specified in the second task. The thesis will present the formal model [Chapter 5] and explain the specification approach applied in the process building the formal model. The user requirement capture and the verification of the application feature prediction and the validation of the APIs are not covered in the thesis. As the specific approach applied in the modeling process, the mixed approach of an object-oriented approach and a task-oriented approach [Section 4.2], and level-wise refinement [Section 6.1] will be discussed in this thesis.

The second task of this thesis research is to use ASM methodologies to specify interactions between systems. The interactions have been specified as a set of activities [Section 5.4]. Each activity interacts between the user, the GUI application and the CoreASM engine. The purpose of specifying these interactions is to know the information objects exchanged during interactions and the methods used to exchange these information objects. The above knowledge assists us in specifying interfaces (APIs) between systems [Section 5.5].

1.2 Thesis organization

The thesis is organized as follows.

Chapter 2 briefly overviews general design strategies and processes applied in the development of human-computer interaction systems. Chapter 3 introduces abstract state machine paradigms. The definitions of basic ASM terms are given in this chapter. These ASM terms are used to describe the execution processes and the states of ASM abstract models. The specification approach applied in this thesis is subsequently explained in Chapter 4. Chapter 5 describes three aspects of the GUI ASM model: the architecture, the activities, and the interfaces. Chapter 6 provides a discussion of the formal modeling process carried out during this thesis work. Chapter 7 introduces the implementation of CASM-GUI, a visual CoreASM language debugger, and then experiments an actual CoreASM model, the ATM model. Chapter 8 concludes the thesis and discusses possible future work.
1.3 Related works

It is widely recognized that designing a GUI system is difficult. Some researchers are starting to use formal modeling to design and test GUI systems.

Myers enumerated several reasons why a GUI system is difficult to design in his technical report [23]. In addition to designing any complex system, GUI system design has the following specific problems.

- Designers have difficulty learning the user's tasks.
- The tasks and domains are complex.
- There are many different aspects to the design which must all be balanced, such as standards, graphic design, technical writing, internationalization, performance, multiple levels of detail, social factors, legal issues, and implementation time.
- The existing theories and guidelines are not sufficient. They are too general or too specific, and no one theory or one guideline can address all issues.
- Iterative design is difficult. There are a few important issues needed to be considered. To get real users who actually use the system is important in iterative design. Also, it may be taken many times to do iterative testing on one particular problem in order to make sure the solution to that problem is correct.

Formal model specifications can be used for communication between designer and implementer, and for analysis about the system and behaviors of the system. Designer and implementer would have a better understanding about user tasks after reading the formal model specifications.

Standard software engineering formalisms can be used to specify an interactive system. There are three brands of formalism, model based, algebraic formalisms, temporal and deontic logics. Model-based specifications define the state of a system and the operations which change the state. Model-based formalisms use precisely defined mathematical notations to describe the behavior of a system in an abstract
CHAPTER 1. INTRODUCTION

language. The major model-oriented specification notations are $Z$ and $VDM$ and $ASM$. For example, $Z$ has been used to specify the GUI system, the presenter [31]. Algebraic specifications describe the effects of sequences of actions. The architecture of the system will not be specified with algebraic specification notations. The algebraic specification notations include $Larch$ and $ACT-ONE$. $ACT-ONE$ is the functional part of the ISO standard language $LOTOS$. Temporal and deontic logics have been used to specify certain properties of actions. Temporal logics describe when actions happen, and deontic logics describe permitted actions and responsibility.

Some formal models for GUIs have been proposed [22][31]. They mostly model the actions of a GUI's internal objects. One example of this kind of model was developed by Atif M. Memon, Martha E. Pollack and Mary Lou Soffa [22]. Their GUI model is constructed with a set of objects, a set of properties of these objects, and a set of actions on the change in the properties of these objects. The objects in their model are windows, menus, buttons, etc. This type of GUI models describes the detailed design of a GUI system through the actions of the GUI objects. The interaction between the GUI and other systems is not the model's concern. By contrast, the GUI model introduced in this thesis makes the interaction between the GUI application and the CoreASM engine a key research issue.

There are some researchers in the ASM field currently working on an executable ASM environment. They include the Spec# programming system project (successor to the ASML project) [1] and the XASM project [2], etc. The GUI application is a component of the tool developed in these projects. This is similar to the GUI tool developed in this thesis work. However, the GUI tools in the above projects are simple implementations. No ASM-based models are provided with them.

1.4 Significance of work

The GUI application described in this thesis was modeled with the help of ASM methodologies, specifically through a mixture approach of object-oriented design and task-oriented design approaches, and level-wise refinement. ASM methodologies have been used here to specify the functions of a graphical user interface system. They
are also used to define interfaces between interactive systems. The work shows that the ASM methodologies specified in this thesis can be applied to predictively model a general interactive system.

The GUI model described in this thesis was constructed as a foundation for analyzing the function form of a GUI system. The GUI components in the model execute independently. The functions of the interactive system are specified as activities. Each activity can be refined with its own detailed specification. This kind of independence provides the ability for the GUI model to customize the architecture and the functions of a GUI system according to requirements.

The GUI model can be used to compare the behavior of the model to the implementation of the system through the use of scenarios and to discover any errors after the GUI model gets well-refined for execution. This evaluation can be done in the testing phase.
Chapter 2

Human computer interactive systems

Graphical user interface is one type of human-computer interaction (HCI) system. The basic goal of HCI is to make computers more user-friendly and easier to use. Some methodologies and design processes can help developers achieve this goal when they design a HCI system, such as task analysis, scenario design and software development life cycle models.

2.1 General introduction to HCI

The term *human-computer interaction* has been widely used in computer research since the 1980s. During the past three decades, many different types of computing devices have come into popular use, especially after the introduction of the personal computer in the 1980s and the spread of the Internet during the 1990s. Computing devices and the computer technologies behind these devices are changing people’s work and personal lives.

Computing devices are becoming an important communications medium. People
use computers at work to keep in contact with colleagues or customers and to commu-
nicate with their relatives and their friends off work. One significant research field is to studying on the ways that humans use or interact with computers. This research field is not limited to computer science; psychology and engineering technologies are being applied in this research field as well. Lately, information science is also influencing HCI research. The computer can collect information and analyze it. Managing information during interactions between humans and computers is one issue in this field. Researchers are doing their work to improve user work performance and to figure out how to use theories to design better computer systems.

*Human-computer interaction* is defined as the interaction between user and computer. A *user* is not necessarily one human user. It may be a group of human users. These days the term *computer* is similarly not restricted to a simple computer. More and more machines have integrated with computers. These computing-enabled machines offer users more functions, and require more types of interactions with users. Researchers view any system that consists of computing-enabled components as a kind of *computer*.

### 2.2 Graphical user interfaces

A *Graphical User Interface* (abbreviated as GUI) is one type of interface in human-computer interaction. Generally, a GUI is a visual operation display that takes advantage of a computer’s graphical capabilities on a monitor screen to offer the computer operator the ability to operate computer applications or computer systems. The computer operator commands computer applications through the Graphical User Interface.

A *Graphical User Interface* assists the user to reduce the memory time required to remember the complex command language of computer applications. It can provide the user with a certain degree of guidance to do the next operation. For example, some buttons are disabled after a user does an operation. This forces the user to operate in the correct way and lets the user handle problems directly. A wizard is a good example of a way to guide a user’s operations.
A Graphical User Interface has three main components, a windowing system, an imaging model and an application program interface (API) [21]. The windowing system builds windows, menus, buttons, dialog boxes and other components. The imaging model defines graphics and fonts for a system. The application program interface for the GUI enables a system to draw graphical components on screen. There are a few common characteristics to a graphical user interface. They include a pointer (a cursor on a screen), a pointing device, icons, buttons, toolbars, menus, a desktop, windows, and dialog boxes. These common characteristics of a graphical user interface reduce the learning curves when users move from one application to another.

The first graphical user interface was designed by Xerox Corporation's Palo Alto Research Center in the 1970s. It did not gain commercial success. Apple Computer brought the graphical user interface to its Apple Macintosh and made the graphical user interface the most popular interface for the modern personal computer. Over the past three decades, the graphical user interface has spread from the personal computer to other computer devices, such as handheld devices.

There are three main paradigms influencing modern graphical user interface standards. Apple Macintosh was the first popular GUI on a personal computer. It introduced some common graphical user interface elements, such as menus, point-and-click and mouse-driven processes. The IBM SAA invented keyboard short-cut keys for the graphical user interface. The X-windows system, mostly used by Unix and its successor Linux, works directly with a network. The display and the applications can run on separate computers in a network. The three paradigms have different Look-and-Feel, but share most of the same graphical user interface characteristics. [21]

2.3 User interface management systems

Some research has been done toward providing a design platform for programmers developing interactive systems. This interactive system design platform consists of a set of services, including a conceptual architecture for interactive environments and techniques for implementing application semantics and its presentations [12]. This
Figure 2.1: The *Model-View-Controller* model

design platform is called *User Interface Management Systems (UIMS)*. Some people prefer another term, *User Interface Development Systems (UIDS)*[12].

One popular model for UIMS is the *model-view-controller* paradigm, or the *MVC* paradigm. This paradigm comes from the Smalltalk programming environment. The central architecture of the Java Foundation Classes (JFC) also relies on the MVC paradigm.

A system making use of the *Model-View-Controller* paradigm has three elements.

- *Model*: the application logic of the program. It is an abstraction that represents the nature and state of a user interface object.
- *View*: the representation of a user interface object to users.
- *Controller*: the component that synchronizes the model and its view. It keeps the view representative of the model's nature and state. It also accepts user inputs to update the model.

The *model* is the internal logic of the application. It maintains the states of a user interface object during interactions.

The *view* presents users with the nature or the state of a user interface object. In a GUI, the view renders user interface objects on the screen.
The view must present the model. Therefore, a controller is added to manage both the view and the model, and to provide a communication bridge between them. When the model is updated and the state of a user interface object is changed, the controller notices the view and refreshes the view to match the change in the model.

In this Model-View-Controller model, the view is the output channel to the user, presenting user interface objects. The input channel is the controller. Receiving input is the secondary role of the controller, which mainly acts as a bridge between the view and the model. The controller receives user inputs and then updates the model. The view may refresh with each model update.

The main idea in this Model-View-Controller paradigm is to separate the nature of a user interface object from the presentation of it. One model can bind with different pairs of input-output channels, views and controllers. The model is reusable, portable, and independent of devices. It therefore can reduce the development cost. In addition, the view-controller pair is customizable.

The Model-View-Controller is linked to a particular programming language environment. Coutaz suggests a more conceptual architecture than the MVC model. The architecture suggested by Coutaz is the Presentation-Abstraction-Control model, or PAC model [10].

The PAC model follows the MVC paradigm in that it separates the application semantics from the presentation of the application. The abstraction in the PAC model
Abstracts the application's semantics. The presentation takes charge of the view of the application. Both the abstraction and its presentation need the control to keep consistent with each other. In the PAC model, the input and output channels are both grouped into the presentation. The user does not interact with the control directly.

Both the MVC model and the PAC model are proposed for the implementation of a user interface system. To a software designer, the abstract model and its presentation are two important issues in developing a user interaction system. The controller in the above two models becomes the programming language that supports the communication between the abstract model and its presentation. It becomes a choice among implementation platforms and programming languages in the implementation phase.

The abstract model abstracts the internal logic and states of an application. It is obvious to a software designer. There is another set of abstractions supporting the presentation. The presentation abstraction is visible to the user. It should also be meaningful to the user. The GUI designer should bridge the two sets of abstractions together and keep them consistent.

The abstract model is the core of an application. A system designer should focus on the abstract model at the beginning of development. The GUI ASM model discussed
User opens a file in a computer application:

- User clicks "open..." in the menu of the computer application on the computer screen;
- Application reads file system;
- Application displays the file-chooser window. The file-chooser window shows files in the file system;
- User navigates file system through the file-chooser window;
- User picks one file in the file-chooser window;
- User clicks the "OK" button on the file-chooser window;
- The file-chooser window disappears;
- Application opens the file chosen by User and reads it from the file system;
- Application displays the file on screen.

Figure 2.4: An example of scenario notation

in this thesis is this abstract model [Chapter 5]. The presentation for the system, with specific GUI control elements, is left for the GUI designer.

2.4 Scenarios

Scenarios are stories of interaction processes. Software engineers write scenarios to record a set of interactions between systems and their environments. Scenarios are informal descriptions of these interaction processes. They are a well-known technique in the HCI field. Like a story, a scenario has actors and a description of interaction behaviors. Readers can learn and understand a system's interaction processes and figure out some functions of the system. Figure 2.4 gives an example of a scenario written in plain text. Scenarios can also be combined with sketches and screen shots. These are called story boards, and are used to provide details about interaction processes.

Scenarios can be useful throughout the system development life cycle. Scenarios

- Verify that the design of a system makes sense to both the user and the designer at the design phase;
- Enable communication with other parties, including users, designers, programmers and testers, during the entire development process;
- Validate other models of the same system. Testers can also design test cases dependent on scenarios.
2.5 Task analysis in HCI

Task analysis is often used in the specification phase of design. Hierarchical task analysis and use case task analysis are two popular task analysis techniques.

2.5.1 Introduction to task analysis

Task analysis is a process that involves the analysis of a task performed in a system and its environment. The term Task refers to a unit of activities. These activities are performed to achieve certain goals. To analyze tasks in HCI, it is necessary to analyze the activity that happens between an application and a user.

Task analysis can be applied in various fields, not only in computer science. Therefore, computer devices are not essential components in a system when we talk about task analysis.

Another thing should be clarified about systems. A system in task analysis is not a single system or application to be developed. It is a combination of an application, users, the system that supports the application, and other environmental factors.

A task has individual goals to achieve. We link a series of tasks to realize longer-term goals. This series of tasks is called a process, sometimes called a business process. One example of a business process is sending a piece of regular mail to a receiver. There are three main tasks in this process. Sender drops mail in a post office box; postal company delivers mail; receiver receives mail.

Task analysis is normally applied to analyze an existing system. The system exists, and analyst learns the system by observing the system's behaviors. However, task analysis is also useful when developing a new system. The analyst can often observe a similar existing system and then design a new system. This observe-design process contributes to the specification phase of development.

Task analysis has the following objects.

- Goal: understand the individual and overall task goals.
- Actors: understand who acts in each task.
- Environment: understand the environment that a task is performed in.
CHAPTER 2. HUMAN COMPUTER INTERACTIVE SYSTEMS

Figure 2.5: Hierarchical task analysis

- Actions: understand the actions in task and the order of those actions.
- Precondition: understand when and if preconditions exist when a task starts.

2.5.2 Hierarchical task analysis

Hierarchical task analysis is one standard approach to decomposing tasks. It involves dividing a main task into a few subtasks. The completeness of all subtasks ensures that the main task is completed. Subtasks can be divided into sub-subtasks, until the details of each task reach an acceptable abstract level. The analyst needs to do the following analysis: identify actors and actions for the tasks, plan the tasks, and analyze the preconditions of the tasks.

Tasks are organized by their goals. Certain subtasks are completed to achieve a goal one level up in the hierarchy. Hierarchical task analysis makes the to-be-analyzed tasks organized. An analyst can easily track the tasks. The relationships of the
subtasks are clear. If an analyst applies hierarchical task analysis during the design phase, it helps to plan the tasks in the process. Some subtasks can be performed simultaneously with other subtasks; some cannot because their parent tasks should be processed in a sequential way. This is one example of how an analyst can assess system performance by making comparative predictions as to the sequences of actions with the help of hierarchical task analysis.

2.5.3 Use case task analysis

The term use case task analysis [20] [3] is employed by some object-oriented software design methodologies. One example is building a task model with use cases in UML modeling [25].

Use case task analysis has two objects, actors and use cases.

Actors:

An actor is basically a user of the system. It is actually a user type or category. As defined here, a user is not a specific person. The important concern with actors is the role of users in the system operation.

Use Cases:

A use case is a scenario that describes a use of the system by actors, interacting with the system to accomplish a goal. A scenario is used to identify the main tasks that should be performed by actors.

The use case task analysis is normally undertaken to determine if a new system is satisfactory from the user's perspective. A use case should describe the way the tasks are performed when the new system is in place. A use case should also be documented in a way that is easy to understand for non-technical persons since a use case is usually a medium for developers and customers to confirm that the new system satisfies the customer's needs.

Use cases can be refined incrementally [26]. Analysts can set up a use case at the beginning, and then decompose it into separate small use cases. Each time, the analyst can pay attention to individual use cases.
2.6 Software development life cycle in HCI

Software engineers have developed techniques to manage the software development process. A few models have been built, such as the Waterfall Model, the Spiral Model and the Evolutionary Model. These techniques can also be applied to describe the process of the development of a human-computer interaction system. I am going to use the Waterfall model to discuss the software development life cycle in HCI.

The waterfall model displayed in the Figure 2.6 is an improved version of the original waterfall model. The need for improvement stems from the fact that the requirement-capturing activity in the requirement specification phase is often not properly carried out. The requirement specification may be inconsistent or incomplete. This situation exists normally in a real development process. The developer may go back and change both the specification and the design even during the late phases of the project.
System specification modeling can make it possible to start the evaluation process at a much earlier stage of the development. In order to test the usability properties of a design, the developer needs a working system to observe interactions between users and this working system and then to evaluate this working system to measure its performance. It is not wise to delay the evaluation until the system is fully implemented. By contrast, system specification modeling can generate a working model for an evaluation before complete system implementation. This working model makes evaluation possible at a much earlier development stage. Users can now be invited to do evaluation on the system specification model at an early stage. Early stage evaluation reduces errors in the requirement specification. Therefore it can also reduce the number of times that the designer has to revisit the earlier development phases.

System specification modeling assists the developer to efficiently build a correct system. Normally, the developer attempts to capture requirements based on a user's perspective. However, the user normally cannot imagine how to interact with a system that does not exist. Nor does the user generally have a clear understanding of the tasks and activities performed by the new system. The developer in turn cannot gain a clear sense of the user requirements. Dix, Finlay, Abowd, and Beale described this chicken-and-egg puzzle in their book [12]. System specification modeling may be a solution to this puzzle. First, capture requirements that derive from what users imagine about the new system. The developer builds a specification model for these requirements and then invites users to evaluate it. The model is continually corrected through discussions between users and the developer, until the model agrees with the users' views. The GUI ASM model built in this thesis is this kind of specification model, that has been built in order to verify the prediction about a new system.

2.7 System design in HCI

Human computer interactive system design needs to involve multiple design methodologies and design processes. ASM modeling design is one agile methodology to design an interactive system at the early design phase. An ASM model can be used to describe the business logic of a system and to abstract the appearance description from
the system design documentations. It separates the work between system designer and UI designer and provides UI designer a design freedom. Many task analysis technologies are design tools for system designers and assist them to refine their work. Scenarios design is another tool for system testers to build test cases. System testers can use these test cases to validate the consistence of the ASM model and the implementation of the system.
Chapter 3

Abstract state machines

The Abstract State Machine (abbreviated as ASM) paradigm, introduced by Yuri Gurevich in 1988 [17], offers a framework for high-level system design and analysis. It has been proved that ASM methodologies are practical in modeling and analyzing different sizes of systems, from small to complex. A few research groups have been applying ASM techniques in different areas, such as hardware and software architectures, programming languages, network protocols and algorithm verification.

ASMs offer methods for specifying systems at a highly abstract level. ASMs are both abstract and executable. Their abstract nature allows the system designer to focus on system concepts and not to be disturbed by the details. ASMs can specify a system at different abstract levels. The system designer can refine the system from a more abstract level to a less abstract level by providing more details and making more design decisions. The executability of a well-refined ASM assists system designers to verify and validate designed systems before coding. ASMs have precise semantics. They are constructed mathematically and logically.

ASMs can be applied to a predictive model or be used to describe an existing system. Many ASM researchers have successfully validated existing systems, such as Java [28], with ASM methodology. ASMs can also be employed to model new systems. The GUI ASM model introduced in this thesis is an exercise in modeling
such a predictive system. System designers do not usually know or understand most of the details of new systems at the beginning. They do not want to make many design decisions in the early design phases. Abstraction is a good tool for system designers to start their work with.

### 3.1 Abstract state machines

A basic Abstract State Machine (ASM) is defined as a set of transition rules of a form

\[
\text{if } \text{Condition} \text{ then Updates}
\]

which specify the transition between two abstract states. These two abstract states are the states before and after an abstract state machine executes.

A condition (also called as guard) is a first-order formula without free variables. A condition evaluates to \text{true} or \text{false}. The term updates refers to a finite set of updates of functions with a form

\[
f(t_1, \ldots, t_n) := t
\]

Functions with certain parameters get updated in parallel. The value of a function is changed (updated) to a new value. Briefly, updates occur when a finite set of functions change their values during the transition process from one state to another state.

The execution of an ASM machine is an update process. Starting from a given state, the values of finite functions with indicated parameters update in parallel to new values, according to rules specified by the ASM. If the updates of these functions are consistent, then a new state is achieved.

An ASM model has two objects: an ASM machine and its environment. The ASM machine is a core object in ASM modeling. The ASM machine always executes in an environment. The environment constrains that ASM machine. The ASM machine communicates with its environment through an interface. This interface will be discussed in the section Function [Section 3.2.2].
CHAPTER 3. ABSTRACT STATE MACHINES

An ASM machine consists of three components.

- Vocabulary: definitions of functions and universes [Section 3.2.3].
- Initial state: the beginning state of the ASM machine.
- Program: the main rule of the ASM machine.

3.2 Some definitions used in ASMs

This section introduces some terms used in ASMs. People use these terms to describe the execution of an abstract state machine. Because the GUI ASM model is built to specify a GUI application tool, that helps users of the tool to explore executions of an ASM machine, we will see these terms many times in the description of the GUI ASM model.

3.2.1 Universe

Universe is a set of elements used in an ASM machine. Some books call it the domain. An ASM machine can have multiple universes existing in it, such as an integer universe or a character universe.

\[ U_1 \equiv \{ 'a', 'b', 'c', 'd', 'k', 'o' \} \]

3.2.2 Function

A function is expressed in a form

\[ f(x_1, \ldots, x_n) \]

A function is identified with its function name and its arguments (arities). This identification is the signature of a function. When arguments have been assigned, a function returns a certain value. A function signature and its indicated parameters and the value of the function with those parameters construct a location. A location is
like a memory unit. The combination of function signature and indicated parameters is the location name. The value of the function with those indicated parameters is the value stored in that location. We say that value is the value of that location.

With particular parameter values \((t_1, \ldots, t_n)\), the location for the function \(f(x_1, \ldots, x_n)\) is \((f(x_1, \ldots, x_n), (t_1, \ldots, t_n))\), or more concisely \(f(t_1, \ldots, t_n)\). The value of the location is the value of \(f(t_1, \ldots, t_n)\).

Functions in ASMs are classified into two groups, static functions and dynamic functions.

A static function does not change its value during the execution of an ASM machine. This means that the value of a function with given parameters does not depend on the state of the ASM machine. The values of static functions are constant in all states.

A dynamic function can change its value during the execution of an ASM machine. The values of these dynamic functions may differ according to the state of the ASM machine.

Dynamic functions can be divided to three subclasses further, depending on the communication methods between an ASM machine \(M\) and its environment.

Dynamic functions\(^1\) can be described as

- Controlled,
- Monitored,
- Shared.

A controlled function can be updated but only updated by the ASM machine \(M\). It is internal for \(M\).

A monitored function can be updated and only updated by the environment. The ASM machine \(M\) can read the value of the monitored function. The function is external for \(M\).

---

\(^1\)Some books declare there is another subclass of dynamic function, the out function. An out function is updated but not read by ASMs machine \(M\) and is read but not updated by the environment or other agents. These out functions are most used in multi-agent systems. They are a kind of controlled function [9]. Therefore, the out function is not listed here.
A shared function can be updated by both the ASM machine $M$ and its environment, and can be read by both.

Controlled functions, monitored functions and shared functions construct the interface between the ASM machine $M$ and its environment. They provide channels for both communication and interaction.

### 3.2.3 Vocabulary

A vocabulary is a finite collection of signatures of functions and universes. The signature of a function contains the function name and a fixed number of arities. Two signatures of functions follow:

\[
\text{sum}( a : \text{Integer}, b : \text{Integer})
\]

and

\[
\text{sum}( a : \text{Integer}, b : \text{Float})
\]

In strong-typed ASMs, two functions are different because of the difference in the types of the arities $b$; in weak-typed ASMs, two signatures identify the same function since the function names are the same and the number of arities are the same.

The Lipari guide [18] does not mention whether ASM is strong-typed or weak-typed. Most of the ASM tools available have incorporated a strong-typed system into their specification languages (Spec# [1] and XASM [2]). Strong typing makes these languages possible to do type checking at compile time, and helps modelers of ASM find errors at the earlier time, in contrast to the run-time error checking. Weak-typed ASM relies heavily on run-time error checking to produce correct programs. The CoreASM language and its toolset focus on early phases of the software development process. It is recommended to build a rapid prototype with ASMs, starting with abstract and weak-typed models in early analysis and specification, with the CoreASM language. Weak-typed ASM satisfies this purpose. Modelers of ASM now have an ability to ignore typing restrictions when building ASM models. One particular
example is that identifiers can be used without declaration. Therefore, the CoreASM language has been developed as a weak-typed ASM language.

3.2.4 State

*State* is the notion of mathematical structures of elements from *universes*. A state *A* is a nonempty set *X* together with functions in vocabulary and the predicates. The set *X* is called the *base set* or *superuniverse* of *A*. A function signature with arity *t* is interpreted as *t*-ary operation over *X*.

3.2.5 Update set

An *update* of *A* is the pair of a location *loc* of *A* and an element *v* of *A*, (*loc*, *v*). It is in an assignment form \( f(t_1, \ldots, t_n) := v \), read as changing a value *v* in a location \( loc = f(t_1, \ldots, t_n) \).

Two updates \((loc_1, v_1)\) and \((loc_2, v_2)\) clash if \(loc_1 = loc_2\) but \(v_1 \neq v_2\).

An ASM machine *M* may fire a set of updates simultaneously from a state *A*_1 to reach another state *A*_2. This set of updates is called an *Update Set*. In the state *A*_1, the value of the location *f*(\(t_1, \ldots, t_n\)) is *v*_1. To fire the update, the value of the location *f*(\(t_1, \ldots, t_n\)) is changed to *v*_2. Because updates are fired simultaneously, *M* may assign different values to one location. If the clash happens during updating, we call this update set *inconsistent*. The new state of *M* will not be reached when an inconsistent update set exists.

3.2.6 Step and run

An ASM computation *step* is the process in which ASM machine *M* simultaneously fires all updates for all transition rules in a given state to reach a new state. If the update set in this process is consistent, then a new state of *M* is reached. If at least a pair of updates clashes, *M* will halt. The new state cannot be achieved at this step.

A *run* is a sequence of *steps* completed by an ASM machine *M* on a timeline. The ASM machine *M* changes from one state to another state. The number of *steps* in a
run may be endless. Therefore, the number of states of the ASM machine $M$ may be infinite in one run.

3.2.7 Rule and program

As per the definition of ASMs [Section 3.1], the basic rule in ASMs is the if-then transition rule. Generally, a rule in ASMs is one of the following basic rules or one compounded rule constructed from these basic rules.

The basic rules include the following.

- Skip rule: “skip”. Causes no change.
- Update rule: “$f(t_1, \ldots, t_n) := v$”. Assigns an element or expression to a location.
- Conditional rule: “if $e$ then $R_1$ else $R_2$”. A branch operation. Here $e$ is a Boolean expression. To execute this conditional rule, check $e$. If $e$ is true, execute $R_1$. If $e$ is false, execute $R_2$.
- Block rule: “do in-parallel $R_1 \ R_2$”. Executes rules $R_1$, $R_2$ simultaneously.
- Import rule: “import $x \ R_1(x)$”. Discovers any element $x$ of the reserve and executes the rule $R_1(x)$.

Rule is used usually to describe the behaviors of ASM machine $M$. Function in an ASM machine is different than a rule at this point. A function in an ASM machine is a formula, computing expression of values of the locations and elements in the universes. It does not describe the execution of ASM machine $M$. The meaning of function at this point is different than it is in other computing languages such as C.

Program is the main rule of ASM machine $M$. It describes one step of $M$. It executes repeatedly until the state does not change or no rule is applicable any more or the update set fired in one step is empty [9] (successful termination). A program may halt because of an inconsistent update set (failed termination).
3.3 Parallel abstract state machines

When we list the Block rule (do-in-parallel rule) as one of basic rules in ASMs, it declares that an ASM machine possesses a simultaneous execution ability. It is an important feature to support refinement in parallel or distributed implementations.

A do-forall rule is introduced here to enrich the notion of ASM rules.

- Do-forall rule: "forall x with do R_1(x)". It executes the rule $R_1(x)$ for each $x$ satisfying a given condition $\phi$. $x$ will have some free occurrences in $R$.

For each rule $R_1(x)$, there is a parallel ASM to simulate the given rule step for step.

3.4 Non-deterministic abstract state machines

In contrast, there is another kind of execution. An ASM machine chooses an $x$ to execute the rule $R_1(x)$, rather than executing for all $x$.

A choose rule is introduced here.

- Choose rule: "choose x with $\phi$ do $R_1(x)$". Executes the rule $R_1(x)$ for an $x$ satisfying a given selection property $\phi$.

The choose rule causes non-determinism in ASMs. The choose method is not specified in the choose rule. It produces a problem. The user can not predict which $x$ will be picked up without executing the ASM machine actually. There is a positive side to this non-determinism in ASM methodology. It helps the designer to avoid dealing with the details of the scheduling of rule executions. In the CoreASM supporting tool environment, a designer can provide a particular choose method by supplying a plug-in [14].
3.5 Distributed abstract state machines

Basic ASM (sequential ASM) and parallel ASM are ASM with a single agent. Distributed Abstract State Machines (DASMs) have a set of agents. Agent is an abstract state machine executed on its own local state. The agents interact by reading and writing on the shared locations in the global state of this distributed abstract state machine.

A run of distributed abstract state machines is defined as a partial order of moves of finite numbers of agents. A single computation step of an agent is called a move of this agent. The moves of a single agent can be atomic or durative. Agents in distributed ASM may execute their computation steps concurrently.

Formally, a run $\rho$ of a distributed ASM $M$ is given by a triple $(M, \lambda, \delta)$ satisfying all of the following four conditions [18]:

1. $M$ is a partially ordered set of moves where each move has only finitely many predecessors.

2. $\lambda$ is a function on $M$ associating agents with moves such that the moves of any single agent of $A$ are linearly ordered.

3. $\delta$ assigns a state of $A$ to each initial segment $Y$ of $M$, where $\delta(Y)$ is the result of performing all moves in $Y$; $\delta(Y)$ is an initial state if $Y$ is empty.

4. The coherence condition: if $x$ is a maximal element in a finite initial segment $X$ of $M$ and $Y = X - \{x\}$ then $\lambda(x)$ is an agent $\delta(Y)$ and $\delta(X)$ is obtained from $\delta(Y)$ by firing $\lambda(x)$ at $\delta(Y)$.

The definition of a run of a distributed ASM did not specify the order of executions of different agents. This offers a benefit to model builders, a freedom to design and analyze models for distributed systems without a prior commitment of scheduling.

The GUI ASM model in this thesis describes a GUI application system with multiple internal components and multiple actors (CASM.User, CASM.Engine, File Storage, and CASM.GUI). Each actor is one agent in the distributed system. An
amount of synchronous and asynchronous interactions exist between internal components and actors [Section 6.2.3]. In order to simplify the model, internal components are specified as parallel ASMs at the very high abstraction level. According to requirements about the system, internal components have been refined to distributed ASMs at the lower abstraction level. One example is refining the model by applying the special synchronization and communication concepts. Activities specified in the GUI ASM model are synchronous interactions and asynchronous interactions between internal components and actors. At the third abstraction level [Section 6.1.3] of the GUI ASM model, the moves of the agents are atomic. Then, the moves of the agents have been refined to durative moves when the concurrency problem of the activities [Section 6.2.2] needs to be discussed at the fourth abstraction level of the GUI ASM model.

The internal components and the actors in the GUI ASM model can be refined as agents in distributed ASMs. This offers some independence to these components and actors in the two aspects, the architecture and the execution. This kind of independence provides the ability for the GUI ASM model to customize the architecture and the functions of a GUI application system according to requirements.

### 3.6 Turbo abstract state machines

*Turbo Abstract State Machines (Turbo ASMs)* provide a practical composition and structuring principle that extends the basic ASM. The execution of a turbo ASM merges all update sets generated in individual sub-machines. A *sub-machine* is denoted as a rule \( R(t_1, t_2, \ldots, t_n) \). Each sub-machine is treated as a black box, hiding its local states and local updates. The state of a turbo ASM cannot be updated until the merging of all update sets generated by all sub-machines is successful. If a local update set in a sub-machine is inconsistent or the merging of update sets generated by the sub-machines fails, the turbo ASM halts (failed termination).

The turbo ASM has introduced some structuring programming principles into basic ASM, such as seq, iterate, sub-machine, recursion and value-return. The *seq*-construct has been used in the GUI ASM model in this thesis to specify the sequence
CHAPTER 3. ABSTRACT STATE MACHINES

of activities.

The turbo ASM seq -construct combines the simultaneous atomic updates of basic ASMs in a global state with sequential execution.

- Seq-construct: “R1 seq R2”. Starts to execute the rule R2 only when the rule R1 is completed and the update sets in R1 are consistent.

The merging of update sets generated by rules R1 and R2 is defined as the following. One update is (loc, v), where loc is a location and v is the value in that location. U is the update set generated by the rule R1; V is the update set generated by the rule R2; W is the update set that results from the merging of U and V. W is also the update set of the turbo ASM in that step.

\[ W = \begin{cases} \{(loc, v) \in U \mid loc \notin \text{Locations}(V)\} \cup V & \text{if } U \text{ is consistent} \\ U & \text{otherwise} \end{cases} \]

The definition implies that turbo ASM will get stuck at the first inconsistent rule.

3.7 Summary

Abstract state machine is used to build a model of a system and to describe the behaviors of the system. Parallel ASM, distributed ASM, and turbo ASM have been used in the thesis to build the GUI ASM model. After the ASM model is built, system designer can validate experimentally by executing the model in the CoreASM supporting tool environment and watching states and update sets of the ASM model in different runs.
Chapter 4

Formal modeling approaches

Formal modeling helps build more robust systems. ASM methodology was applied in the thesis research to formally specify a GUI system.

4.1 Formal modeling approaches in GUI architecture and functionality design

People often build a model before implementing an interactive system. The model is a structural description of the relevant information about the interactive system. A designer can use this model to specify and analyze the interactive system to be developed. This approach is often called the model-based design approach. UML is one technique applied in this kind of modeling.

In design, designers can use a few kinds of models to describe the system to be designed. These kinds of models include, but are not be limited to task models, user models and interaction models. These models provide particle views about the system. Not a kind of model can represent all the information about the system. Task model describes information about tasks that the system performs [32][13]. User model can offer information about user preferences and about forms of interaction [11].
CHAPTER 4. FORMAL MODELING APPROACHES

Interaction model can offer information about interaction and about user interface elements [22][31]. The GUI ASM model in this thesis has captured the architecture of the GUI application system and the tasks of that system. Each task is specified as an activity in the GUI ASM model. The interactions between users and the GUI application in each activity have been described at the third abstraction level of the model [Section 6.1.3]. There exists a difference between the GUI ASM model and an interaction model. No actual user interface elements have been specified in the GUI ASM model when talking about interactions. And, this thesis has not discussed the user model of the GUI application system. At a high abstraction level, the GUI ASM model in this thesis is more like a task model. A task model can be usually modeled in either of two approaches, the object-oriented approach and the task-based approach.

The traditional approach in UML is object-oriented. Designers identify the objects of the proposed interactive system and then analyze the activities of these objects. This very successful approach is currently being widely applied in the computer industry. In recent years, the model-based design approach has developed a new trend. Designers now focus on the tasks and the users of an interactive system. The tasks of the interactive system are identified first, and then the objects involved are manipulated. This approach is called a task-based approach [29].

4.2 A specification approach of GUI modeling in the CoreASM project

The object-oriented approach and the task-oriented approach are both applied throughout the whole modeling process, in different stages. Then a process of refinement is used to solve development problems level by level [Section 6.1]. The combination of the two approaches, plus level-wise refinement, makes the system designer consider the specification of such a system without resorting to heavyweight formal models at the beginning stages. Most designers would prefer to consider lightweight models in the initial interaction design [27].
At the beginning stage of modeling, an object-oriented approach using UML is applied in this project just as it would be with a traditional approach. The first step is to identify the actors in the system. One of main purposes of GUI modeling in this project is to identify the functions and the communications between the GUI application and its environment, especially between the GUI application and the CoreASM Engine. Therefore, they are two important steps to identify the actors and to build the overall architecture of the system, before working out the communication functions that exist among the actors. The object-oriented modeling approach is suitable for this architectural stage.

At the second stage of modeling, the task-based modeling approach is used to model CASM_GUI itself. As specified for the first level of GUI ASM model [Section 6.1.1], the activities are listed and minimally specified. These activities are the user-centered tasks of CASM_GUI. In a user’s view, each activity is the task that CASM_GUI should complete to achieve one user goal, such as viewing a state or processing one step. Each activity involves different actors in the system, even different components inside CASM_GUI. The way how involved these actors and CASM_GUI components would be analyzed in each activity one by one.

The GUI model can be refined when the developer is ready to pay detailed attention to each activity in turn. The following questions will be answered in the refined GUI model. What actors and/or CASM_GUI components are involved in a particular activity? What data is exchanged between these actors and CASM_GUI components? What is the sequence of actions in this activity? The answers are a resource for defining the GUI APIs. The GUI APIs created can then be verified after the GUI model is built.

ASM methodology is good for modeling complex systems, such as interactive systems, providing the developer with the ability to model a system at different levels of abstraction [Section 6.1].
4.3 Summary

The GUI ASM model is built with a mixed model-based design approach of the object-oriented approach and the task-oriented approach. Both approaches were applied separately at different levels of the model. This kind of modeling technique invites system architecture design into the task model building process. It also makes the GUI ASM model to be a foundation to analyze the interactions between systems.
Chapter 5

The GUI ASM model in the CoreASM project

The GUI ASM model explained in this thesis was built as part of my research for the CoreASM project. This GUI ASM model is one part of the CoreASM project. Formal methods were used to produce a design of the user-interface tool. An analysis of the interaction processes and their effect on human psychology is not an objective of this research.

5.1 Architecture of the CoreASM supporting tool environment

The abstract state machine method offers a framework for high-level system design and analysis. After specifying user requirements and building a model to match these requirements, the ASM approach can provide system designers with the valuable option of executing the model without implementing it in real programming code. The executability of an ASM-based model enables the system designers to discover system faults at an early development stage, assisting with verification and validation.
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

In the CoreASM project, we are using an ASM approach to construct a supporting tool environment that allows agile formalizing and high-level design and analysis of a system. A set of easy-to-use tools is provided in this supporting tool environment. This toolset includes an interpreter, CoreASM abstract storage, a debugger, a validator and more. The language parser, the interpreter, the CoreASM abstract storage, and the scheduler constitute the CoreASM engine. The CoreASM engine interprets ground models built with the CoreASM language. With the help of the scheduler, the CoreASM engine can execute not only sequential ASM ground models, but also parallel ASM ground models. The CoreASM supporting tool environment defines a set of GUI → Engine APIs for the CoreASM engine. Third party applications can control and access the CoreASM engine through the GUI → Engine APIs.

The GUI ASM model in this thesis specifies a graphical debugger application tool in the CoreASM supporting tool environment. This GUI ASM model has been used to design the graphical debugger and to validate the set of GUI → Engine APIs involved in the communications between the GUI tool and the engine. A graphical debugger implemented in Java will also be provided to apply the set of GUI → Engine APIs described in this project.

The CoreASM supporting tool environment provides an environment in this project to build graphical user interface tools for the CoreASM engine. The CoreASM engine is a running system for interpreting CoreASM models, and lacks the ability to assist system designers to understand the execution processes of the models. There are a few kinds of tools that can aid engineers to supervise the execution processes, such as a debugger and an animator. A graphical debugger offers a visual presentation of the execution processes of a model.

The GUI → Engine APIs provided in the CoreASM supporting tool environment form a bridge between the graphical user interface tool and the CoreASM engine. This bridge includes two main sets of functions, the control for the CoreASM engine and the access to the model entities. Everyone, including third parties, can build graphical tools to monitor ASM execution through this set of GUI → Engine APIs. Tool designers can decide on their own how to visually present an ASM model. A graphical debugger implemented in Java will be provided in this thesis as a working
Figure 5.1: CoreASM supporting tool environment architecture
example. This debugger will help engineers to explore the updates of an ASM model at each step and to trace function values at each step.

The CoreASM project has an important expandable feature. Yuri Gurevich introduced ASM principles and ASM language semi-formal definitions. The CoreASM language applied in this project comes from the basic ASM principles and provides only the core functions of the general ASM language. Anyone can extend our CoreASM language to their own ASM language by following ASM principles. In the same way, third party groups can extend our CoreASM engine to their own ASM engines for their specified ASM languages through plug-ins [14]. Then again, the graphical tools implemented for CoreASM may not satisfy other new extended ASM languages. Therefore, we have separated the graphical user interface tools from the CoreASM engine and provided the GUI ↔ Engine APIs as one basis of the CoreASM supporting tool environment. Third party groups can extend the GUI ↔ Engine APIs as well as the CoreASM engine and build a new GUI on the extended GUI ↔ Engine APIs. There is a second reason to separate the GUI and the CoreASM engine. Each research group may have its own ideas as to how to visually present its ASM models and their execution and may require different graphical tools to build, verify, test and validate the models. Other groups can build their own graphical tools on the top of the GUI ↔ Engine APIs provided in this project in order to satisfy their requirements.

5.2 Actors in the GUI ASM model

There are four actors in the GUI ASM model.

- CASM_User
- CASM_GUI
- CASM_Engine
- File Storage

The actor CASM_GUI is the research object of this thesis. It will be specified in detail with ASMs (main rule CoreASMGUIProgram). The other three actors,
CASM_User, CASM_GUI, and File Storage, are the environment for the ASMs machine CASM.GUI.

**CASM_User**

The human users of the CASM toolsets. The target users are software engineers, system analysts, and testers.

**CASM_GUI**

The graphical user interface toolset of the CoreASM project. It is the GUI of the visual CoreASM language debugger exactly as described in this thesis. CASM_User can load CoreASM models in CASM_GUI, and execute and explore (view) the models. CASM_GUI provides the visual representation of information about these models.

**CASM_Engine**

The engine to interpret CoreASM programs and to execute CoreASM models. Each CoreASM model is executed in CASM_Engine. CASM_GUI controls and accesses these models in CASM_Engine.

**File Storage**

The storage device to store CoreASM specification files. This device may be a local storage disk, or be a network storage device.
Human-computer interaction happens mainly between two actors, CASM_User and CASM_GUI. CASM_User does not access the CASM_Engine directly. CASM_GUI is the bridge connecting CASM_User and CASM_Engine.

The interactions between CASM_User and CASM_GUI are specified by activities activated by commands that CASM_User sends to CASM_GUI. An activity is a set of behaviors by the four actors. The details of the activities belonging to CASM_GUI have been specified in the GUI ASM model [Section 5.4]. The behaviors of CASM_User in an activity are specified abstractly as ASM rules, such as the rule `getFileURIToLoad (aUser: CoreASM-User, aFileStorage: CoreASM-FileStorage)`. These rules also identify the information data entities exchanged in interactions; for example, the file `uri` is needed in the rule `getFileURIToLoad (aUser: CoreASM-User, aFileStorage: CoreASM-FileStorage).

These actions are specified by ASM rules, not by ASM functions, since these are specifications of interaction behaviors. A system designer can refine these rules to analyze these interactions between human and computer.

The interactions between CASM_GUI and CASM_Engine are specified as GUI ↔ Engine APIs in the GUI ASM model [Section 5.5.1]. We will see how these APIs are applied in activities.

### 5.3 Architecture of CASM_GUI

CASM_GUI consists of a few components. Each component has its own features. A component in CASM_GUI communicates to another component by sending signals to the signal pool of the second component.

#### 5.3.1 Components of CASM_GUI

CASM_GUI consists of these components.

- `control panel`
- `output view`
Each component has its own features.

control panel:
The control panel is the control component of CASM_GUI. It controls other components of CASM_GUI. One major task for the control panel is to respond to activity requests from CASM_User and to activate related activity in CASM_GUI.

output view:
The output view displays the printout from the CoreASM machine executing.

message view:
The message view displays any warning/error messages produced during the execution of the CoreASM machine.

program view:
The program view displays the code for the CoreASM machine.

run view:
The run view displays a run in the execution of the CoreASM machine.

vocab view:
The vocab view displays the vocabulary of the CoreASM machine.

state view:
The state view displays a state of the CoreASM machine.

updateset view:
The update set view displays an update set during the execution of the CoreASM machine.
Figure 5.3: Internal components of CASM.GUI
history manager:

The history manager stores the history records of a run. The history records are mainly about states and update sets. The history manager collects the information about a run of a CoreASM machine and stores it into the history records, and then offers these history records to the run view in the background. The run view uses these history records to rebuild the trail of the run. Now, CASM_User can review the run of that CoreASM machine in the run view. In contrast to the run view, the history manager is invisible to CASM_User.

Each component is specified as an ASM rule. A component has behaviors and provides certain features for CASM_User. For example, the message view can display messages or can clear all messages in the view [Spec 5.1]. The history manager can add a history record or delete a history record or even clear all history records.

```
rule MessageViewProgram
  choose aSignal from messagevSignalPool
  remove aSignal from messagevSignalPool
  case firstOf(aSignal) of
    messagevcDisp →
      displayMsginMessageV(secondOf(aSignal))
    messagevcClearDisplay →
      clearDisplayinMessageV
```

Spec 5.1: Features of message view: display messages and clear messages

All components are executed in a parallel way in the actor CASM_GUI. The CASM_GUIis specified as a parallel ASM machine [Section 3.3]. If model designer needs to specify more details of the components, such as the architecture of components and the communications between these components, each component can be refined from a simple ASM rule in a parallel ASM to an agent in a distributed ASM [Section 3.5]. The time to refine components as distributed ASMs is decided by model designer, based on how abstractly the model designer wants to specify those components.
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

<table>
<thead>
<tr>
<th>CASM_GUI Component</th>
<th>CoreASM Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>control panel</td>
<td>rule ControlPanelProgram</td>
</tr>
<tr>
<td>output view</td>
<td>rule OutputViewProgram</td>
</tr>
<tr>
<td>message view</td>
<td>rule MessageViewProgram</td>
</tr>
<tr>
<td>program view</td>
<td>rule ProgramViewProgram</td>
</tr>
<tr>
<td>run view</td>
<td>rule RunViewProgram</td>
</tr>
<tr>
<td>vocab view</td>
<td>rule VocabViewProgram</td>
</tr>
<tr>
<td>state view</td>
<td>rule StateViewProgram</td>
</tr>
<tr>
<td>updateset view</td>
<td>rule UpdatesetViewProgram</td>
</tr>
<tr>
<td>history manager</td>
<td>rule HistoryManagerProgram</td>
</tr>
</tbody>
</table>

Table 5.1: Components in CASM_GUI

The ability to specify the features of a component in an ASM rule and the ability to execute components in a parallel way provide the independence to components. They make model designer to create or remove a component easily from the GUI system. The addition or the removing of a component will not affect other components in the system.

5.3.2 Communication structure in CASM_GUI

Each component in CASM_GUI has a signal pool. Other components send signals into that signal pool. The CASM_User in the environment sends signals of the type USER_ACTIVITYREQUEST into the control panel's signal pool. Each component checks its own signal pool and activates an activity or a set of activities if a signal is found. Each component has only one signal pool for incoming signals. There is no outgoing signal pool. The sending process from one component in CASM_GUI or CASM_User in the environment to another is atomic and completed immediately.

A signal pool is an abstract data structure. It may be a queue or a stack. For this specific GUI ASM model, a queue would be a better choice. When the designer makes the choose mechanism for the signal pool [Spec 5.4], the designer should make sure that the first-incoming signal is served first. In order to abstract this GUI ASM model, the type of the data structure required for the signal pool [Spec 5.3] and the
Spec 5.2: The components in CASM.GUI

details of the signal pool's choose mechanism [Spec 5.4] do not have been specified in the thesis. These design decisions are left for the designer to make later.

The signal pool is specified abstractly as a list of signals.

Spec 5.3: Specification of a signal pool

The choose mechanism for each signal pool has not been specified. The choose rule is used generally to describe this choose behavior.

The sending process for signals is specified as a rule, signalingTo( aSignalPool : SignalPoolName, aSignal : GUISignal). This rule checks the destination by signal pool name and then inserts the signal into the correct signal pool.

The signal is an abstract data structure. In the GUI ASM model, all signals are of the type GUISignal. A signal consists of two types of objects: command objects
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

<table>
<thead>
<tr>
<th>Component</th>
<th>Signal Pool</th>
<th>Signal Pool Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL PANEL</td>
<td>controlpanelSignalPool</td>
<td>controlpanel</td>
</tr>
<tr>
<td>OUTPUT VIEW</td>
<td>outputvSignalPool</td>
<td>outputv</td>
</tr>
<tr>
<td>MESSAGE VIEW</td>
<td>messagevSignalPool</td>
<td>messagev</td>
</tr>
<tr>
<td>PROGRAM VIEW</td>
<td>programvSignalPool</td>
<td>programv</td>
</tr>
<tr>
<td>RUN VIEW</td>
<td>runvSignalPool</td>
<td>runv</td>
</tr>
<tr>
<td>VOCAB VIEW</td>
<td>vocabvSignalPool</td>
<td>vocabv</td>
</tr>
<tr>
<td>STATE VIEW</td>
<td>statevSignalPool</td>
<td>statev</td>
</tr>
<tr>
<td>UPDATESET VIEW</td>
<td>updatesetvSignalPool</td>
<td>updatesetv</td>
</tr>
<tr>
<td>HISTORY MANAGER</td>
<td>historymanagerSignalPool</td>
<td>historymanager</td>
</tr>
</tbody>
</table>

Table 5.2: Signal pools in components

choose aSignal from messagevSignalPool

Spec 5.4: Choosing a signal from the signal pool of message view

and data objects.

The GUI.Signal is a tuple. Two functions defined for the data structure tuple are used to interpret signal values. The first value is the command. Its type is one of the defined signal types. The second value in GUI.Signal is a data object. The data object is the content of the signal.

The function to get the first value from a signal is defined as firstOf(signal:tuple). It returns the signal type.

The function to get the second value from a signal is defined as secondOf(signal:tuple). It returns the signal data object.

Signal types:

- PROGRAMV_COMMAND,
- MESSAGEV_COMMAND,
- OUTPUTV_COMMAND,
rule signalingTo(aSignalPool : SignalPoolName, aSignal : GUI.Signal) =
  case aSignalPool of
    programv ->
      add aSignal to programvSignalPool
    messagev ->
      add aSignal to messagevSignalPool
    outputv ->
      add aSignal to outputvSignalPool
    statev ->
      add aSignal to statevSignalPool
    updatesetv ->
      add aSignal to updatesetvSignalPool
    runv ->
      add aSignal to runvSignalPool
    historymanager ->
      add aSignal to historymanagerSignalPool
    vocabv ->
      add aSignal to vocabvSignalPool
    controlpanel ->
      add aSignal to controlpanelSignalPool

Spec 5.5: Sending a signal

- RUNV_COMMAND,
- STATEV_COMMAND,
- UPDATESETV_COMMAND,
- HISTORYV_COMMAND,
- VOCABV_COMMAND,
- USER_ACTIVITYREQUEST.
5.4 Activities in CASM_GUI

An activity is a series of actions performed by an actor or actors to achieve a certain objective. In an interactive system, the interaction between actors during an activity that is performed by these actors must be considered.

An activity is performed to achieve an objective. There are a few ways to identify activities in a system: for example, by the actions performed by individual actors, or by the actions performed by the whole system. A human-computer interaction system is designed for human use. Satisfying users is an important goal in HCI design. Therefore, it is good to identify activities from a user perspective. Activities in this GUI ASM model are identified in terms of the objectives that a user might expect that the system could complete. These activities include ForwardRunActivity (making the engine advance the run one step), ViewLastStateActivity (viewing a state) and so on. The objectives that a user might expect that the system could complete are analyzed and are achieved from the informal requirements. In the GUI ASM model, the objective of an activity represents the result of a certain feature of the GUI system.

In the GUI ASM model, an activity is specified in a way that seems all actions in the activity are completed by one actor CASM_GUI. In fact, an activity is often performed by more than one actor. Interactions do exist in an activity. The other actors, CASM_User, CASM_Engine, File Storage, are the environment for the actor CASM_GUI. Actions completed by these environment actors are abstracted from the activity. Interactions between all actors are specified as interfaces (APIs). CASM_GUI interacts with other actors by applying these interfaces (APIs) while performing activities.

An activity in the GUI ASM model represents a feature of the GUI system. At this point it is similar to describe an activity of an object in the object-oriented design. The difference between an activity in the GUI ASM model and an activity in the object-oriented designing is that an activity in the GUI ASM model is the interactions between actors, but an activity in the object-oriented design focuses on actions performed by one object.
An activity is described as a hierarchy of sub-activities. Each sub-activity consists of some actions. We define activity (sub-activity as well) as a rule. An activity is labeled as "Activity. A sub-activity is labeled as "SA.

The activity to load a file, called LoadFileActivity, is a rule. A sub-activity of the activity LoadFileActivity, getFileURIToLoadSA, obtains the file to be loaded, and is also a rule.

<table>
<thead>
<tr>
<th>rule LoadFileActivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule getFileURIToLoadSA</td>
</tr>
</tbody>
</table>

Spec 5.6: Activity and sub-activity

The sub-activities of one activity may form a sequence of actions. When analyzing interactions in a HCI system, such sequences are often found. The next action does not start until the previous action is completed. CASM_GUI does not know which specification file is to be opened until CASM_Gui chooses a specification file to load. Therefore, CASM_GUI performs the sub-activity loadFileSA following the sub-activity getFileURIToLoadSA.

Sequenced activities can be specified as

\[ AAActivity = \text{subactivity1SA seq subactivity2SA} \]

\[ AAActivity \] is performed by completing sub activity \( \text{subactivity2SA} \) after the completion of sub activity \( \text{subactivity1SA} \).

### 5.4.1 Activity specifications

Activities in the GUI ASM model include the following.

**StartupActivity**

This activity starts up the CoreASM work environment. The CoreASM work environment includes CASM_GUI and CASMEngine. The activity creates the GUI
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

Table 5.3: Activities and signals to activate these activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Command to activate the activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartupActivity</td>
<td>userStartup</td>
</tr>
<tr>
<td>LoadFileActivity</td>
<td>userLoadFile</td>
</tr>
<tr>
<td>CheckSpecActivity</td>
<td>userCheckSpec</td>
</tr>
<tr>
<td>InitActivity</td>
<td>userInit</td>
</tr>
<tr>
<td>ForwardRunActivity</td>
<td>userForwardRun</td>
</tr>
<tr>
<td>RollbackActivity</td>
<td>userRollback</td>
</tr>
<tr>
<td>StopActivity</td>
<td>userStop</td>
</tr>
<tr>
<td>InterruptActivity</td>
<td>userInterrupt</td>
</tr>
<tr>
<td>ViewProgramCodeActivity</td>
<td>userViewProgramCode</td>
</tr>
<tr>
<td>ViewLastOutputActivity</td>
<td>userViewLastOutput</td>
</tr>
<tr>
<td>ViewOutputInHistoryActivity</td>
<td>userViewOutputInHistory</td>
</tr>
<tr>
<td>ViewLastMsgActivity</td>
<td>userViewLastMsg</td>
</tr>
<tr>
<td>ViewMsgInHistoryActivity</td>
<td>userViewMsgInHistory</td>
</tr>
<tr>
<td>ViewLastUpdatesetActivity</td>
<td>userViewLastUpdateset</td>
</tr>
<tr>
<td>ViewLastStateActivity</td>
<td>userViewLastState</td>
</tr>
<tr>
<td>ViewUpdatesetInHistoryActivity</td>
<td>userViewUpdatesetInHistoryActivity</td>
</tr>
<tr>
<td>ViewStateInHistoryActivity</td>
<td>userViewStateInHistory</td>
</tr>
<tr>
<td>AddWatchActivity</td>
<td>userAddWatch</td>
</tr>
<tr>
<td>DeleteWatchActivity</td>
<td>userDeleteWatch</td>
</tr>
<tr>
<td>QuitActivity</td>
<td>userQuit</td>
</tr>
</tbody>
</table>

and the engine, then links them together. The CoreASM workspace environment is ready after this activity.

**rule StartupActivity**

createGUISA seq createLinkEngineSA

**LoadFileActivity**

This activity loads a CoreASM specification file into CASM.GUI and displays the file in the *program view*. There are two sub activities, getting the URI of the file, and
loading the file from File Storage. Two data objects are received by CASM_GUI from
its environment, through interfaces (APIs). They are the URI (guiAFileURI) of the
file that CASM_User wants to load and the file (guiAFile). The object guiAFileURI
is received by CASM_GUI from CASM_User and File Storage. The object guiAFile
is received by CASM_GUI from File Storage.

\[
\text{\textbf{rule LoadFileActivity}} \hspace{1cm} \\
\text{getFileURIToLoadSA seq loadFileSA}
\]

\textbf{CheckSpecActivity}

This activity sends a CoreASM specification file to CASM_Engine by CASM_GUI,
and then requires CASM_Engine to check for syntax or other specification errors in
this specification program. It then displays feedback on CASM_GUI. The sending
operation and the check-requesting operation are completed in the sub activity check-
SpecSA(currentEngine, guiAFile). The data object guiAFile is sent to CASM_Engine.
The feedback (output) is then received from CASM_Engine.

\[
\text{\textbf{rule CheckSpecActivity}} \hspace{1cm} \\
\text{checkSpecSA(currentEngine, guiAFile) seq getCSFeedbackSA}
\]

\textbf{InitActivity}

This activity initializes the ASM machine in CASM_Engine and then displays any
feedback in CASM_GUI (the initial state is displayed in the state view, the vocabulary
in the vocab view, error messages or warning messages, if any, in the message view).

\[
\text{\textbf{rule InitActivity}} \hspace{1cm} \\
\text{initSpecSA(currentEngine, guiAFile) seq getISFeedbackSA}
\]
ForwardRunActivity

This activity advances the CoreASM machine a few of steps in CASM.Engine and then displays any feedback in CASM.GUI (Messages appear in the message view, print outs in the output view, states in the state view, update sets in the update set view, error messages or warning messages, if any, in the message view). The first sub-activity gets the number of steps that CASM.User wants the CoreASM machine to go forward (guiNumStepsToForward). The value of guiNumStepsToForward is received by CASM.GUI from CASM.User after this sub-activity completes. Then, the CASM.GUI requires CASM.Engine to execute the ASM machine in the second sub-activity and receives feedback from CASM.Engine in the third sub-activity.

rule ForwardRunActivity
  getNumForwardStepsRequestedSA seq forwardRunSA seq getFRFeedbackSA

RollbackActivity

This activity rolls back a CoreASM machine held in CASM.Engine and then displays the machine's current state in state view. The first sub-activity gets the state id that CASM.User wants the machine to roll back to. The value of guiStateIdBackTo is received by CASM.GUI from CASM.User after this sub-activity is completed. Then CASM.GUI rolls back the machine in CASM.Engine. In contrast to the activity ForwardRunActivity, there is no sub-activity getFeedbackSA in RollbackActivity. The desired rolled back state can be found in the internal component history manager in CASM.GUI. It is not necessary in this case for CASM.GUI to interact with CASM.Engine.

rule RollbackActivity
  getStateIdBackToSA seq rollbackSA
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

StopActivity
This activity stops the running of the CoreASM specification machine. The machine stays at the last state and cannot be executed without reinitialization.

```
rule StopActivity
```

InterruptActivity
This activity interrupts the execution of the CoreASM machine in CASM_Engine.

```
rule InterruptActivity
interrupt(currentEngine) := true
```

ViewProgramCodeActivity
This activity allows CASM_User to view program source code in the program view.

```
rule ViewProgramCodeActivity
```

ViewLastOutputActivity
This activity allows CASM_User to view the output of the most recent step in the output view.

```
rule ViewLastOutputActivity
```

ViewOutputInHistoryActivity
This activity allows CASM_User to view the output of a completed step in the
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

history.

\[
\text{rule } \text{ViewOutputInHistoryActivity}
\]

\textbf{ViewLastMsgActivity}  
This activity allows CASM.User to view the error/warning messages for the most recent step.

\[
\text{rule } \text{ViewLastMsgActivity}
\]

\textbf{ViewMsgInHistoryActivity}  
This activity allows CASM.User to view the error/warning messages for a completed step in the history.

\[
\text{rule } \text{ViewMsgInHistoryActivity}
\]

\textbf{ViewLastUpdatesetActivity}  
This activity allows CASM.User to view the update set for the most recent step.

\[
\text{rule } \text{ViewLastUpdatesetActivity}
\]

\textbf{ViewLastStateActivity}  
This activity allows CASM.User to view the newest state.
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

\begin{quote}
\textbf{rule} ViewLastStateActivity
\end{quote}

\begin{quote}
\textbf{ViewUpdatesetInHistoryActivity}
This activity allows CASM.User to view one update set in the history. The activity has two actors, CASM.User and CASM.GUI, who interact. There are two sub-activities. The sub-activity \textit{getUpdateSetIdViewInHistorySA} gets the update set id of the update set that CASM.User wants to view from the history. The value of \textit{ guiUpdateSetIdViewInHistory} is assigned by CASM.User after this sub-activity completes execution. The next sub-activity allows CASM.User to view the update set.
\end{quote}

\begin{quote}
\textbf{rule} ViewUpdatesetInHistoryActivity
\begin{quote}
\textit{getUpdateSetIdViewInHistorySA}
\begin{quote}
\textit{seq}
\begin{quote}
\textit{viewUpdatesetInHistorySA(guiUpdateSetIdToViewInHistory)}
\end{quote}
\end{quote}
\end{quote}
\end{quote}

\begin{quote}
\textbf{ViewStateInHistoryActivity}
This activity allows CASM.User to view one state in the history. Interaction takes place in this activity. Two actors are involved, CASM.User and CASM.GUI. The sub-activity \textit{getStateIdViewInHistorySA} gets the id for the state that CASM.User wants to view from the history. The value of \textit{ guiStateIdToViewInHistory} is assigned by CASM.User after this sub-activity completes execution. The next sub-activity allows CASM.User to view the state.
\end{quote}
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

rule ViewStateInHistoryActivity
   getStateIdViewInHistorySA
   seq
   ViewStateInHistorySA(guiStateIdToViewInHistory)

AddWatchActivity
This activity allows CASM_User to add a watch in CASM_GUI. A watch is a label of a function, a function whose values CASM_User wants to watch during a run.

rule AddWatchActivity
   createAWatch(currentUser, guiVocab)

DeleteWatchActivity
This activity allows CASM_User to remove a watch in CASM_GUI.

rule DeleteWatchActivity
   selectAWatchToDel(currentUser)

QuitActivity
This activity destroys CASM_Engine and CASM_GUI to terminate the CoreASM work environment.

rule QuitActivity
   killEngineSA(currentEngine) seq killGUI SA
5.5 Interfaces in the GUI ASM model

CASM_GUI, CASM.Engine, CASM.User, and File Storage communicate through interfaces.

5.5.1 Interfaces in the GUI ASM model

The GUI ASM model is an application system. It interacts with other actors through interfaces. The interfaces in this GUI ASM model include the GUI ↔ User interface, the GUI ↔ FileStorage interface, and the GUI ↔ Engine interface. The GUI ↔ User interface is a good object for researchers to analyze interactions between human and computer. The GUI ↔ Engine interface specified with ASMs assists the designer to design the APIs (Application Program Interfaces) for the engine. The designer can validate these APIs through the GUI ASM model.

The interface can provide communication channels for two systems. A system can exchange information with another system through an interface. It can also use the interface to activate another system to carry out operations. The GUI ↔ User interface can offer the following opportunities. A user can activate activities in the GUI system by sending activity requests [Section 5.4] through the GUI ↔ User interface. A user can view information about a machine in the engine with the assistance of the GUI system. This view operation needs the GUI system to communicate with the engine through the GUI ↔ Engine interface. The GUI system also needs the user to provide certain information about operations through the GUI ↔ User interface, such as the number of steps that the user wants the machine in the engine to go forward.

In the GUI ASM model, the interface is specified as ASM rules and ASM functions.

If a feature of an interface is specified as an ASM rule, the feature requires an actor to execute certain actions. For instance in the GUI ↔ Engine interface, CASM_GUI requires CASM.Engine to execute the CoreASM machine in the CASM.Engine for a certain number of steps (rule step ( aEngine : CoreASM-Engine , anI : Integer )). This operation needs CASM.Engine to perform actions to compute results. Therefore, it is specified as a rule. Similarly, in the GUI ↔ User
interface, the rule $\text{getNumForwardStepsRequested} (\text{aUser} : \text{CoreASM-User})$ needs $\text{CASM-User}$ to operate on $\text{CASM-GUI}$, in order to provide the number of steps that $\text{CASM-User}$ requests the machine in $\text{CASM-Engine}$ go forward. These operations of $\text{CASM-User}$ on $\text{CASM-GUI}$ include choosing a textbox on the GUI, typing the number of steps and notifying $\text{CASM-GUI}$ to receive the value.

An ASM function in an interface is used to get a value when an actor is in a state. The function $\text{getLastUpdateSet} : \text{GUI-Watch} \rightarrow \text{GUI-UpdateSets}$ gets an update set after the machine in $\text{CASM-Engine}$ has been updated consistently. At that moment, $\text{CASM-Engine}$ is in a state between executions (steps).

Depending on the way the value of ASM functions are updated, the ASM functions in an interface can be controlled, monitored or shared.

A controlled function updates a value in the environment. For instance, it is possible to interrupt $\text{CASM-Engine}$ by setting the value "interrupt" in $\text{CASM-Engine}$ to be true. The interrupt function is specified as a controlled function.

\[
\text{controlled interrupt} : \text{CoreASM-Engine} \rightarrow \text{Boolean}
\]

Spec 5.7: Controlled function example

A monitored function allows the environment to update a value in the system model specified. $\text{CASM-Engine}$ sets the value of $\text{guiAnUpdateset}$ in $\text{CASM-GUI}$. $\text{CASM-Engine}$ sends a value to $\text{CASM-GUI}$ through an interface.

\[
\text{monitored getLastUpdateSet} : \text{CoreASM-Engine} \rightarrow \text{GUI-UpdateSets}
\]

Spec 5.8: Monitored function example

A shared function updates a value shared by both an environment and the specified
system model. There is no example of a shared function in this GUI ASM model.

5.5.2 The GUI ↔ User interface

The GUI ↔ User interface specified in the GUI ASM model consists of the commands to activate the activities and the operations of CASM.User on CASM.GUI. The analysis in this interface deals with the information data exchanged between CASM.User and CASM.GUI, such as the command signals and the number of steps that the user wants the machine to go forward.

CASM.User sends these activity commands to CASM.GUI through the GUI ↔ User interface.

```
enum domain USER_ACTIVITYREQUEST = {userStartup, userLoadFile, userCheckSpec, userInit, userForwardRun, userRollback, userStop, userInterrupt, userViewProgramCode, userViewLastOutput, userViewOutputInHistory, userViewLastMsg, userViewMsgInHistory, userViewLastUpdateset, userViewUpdatesetInHistory, userViewLastState, userViewStateInHistory, userAddWatch, userDeleteWatch, userQuit}
```

Spec 5.9: Activity commands

The CASM.User provides the information data to CASM.GUI through the GUI ↔ User interface.

5.5.3 The GUI ↔ Engine interface

The GUI ↔ Engine interface consists of a control interface and an access interface. CASM.GUI can control executions of CASM.Engine and can check the properties of CASM.Engine through the control interface. The control operations on
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

rule getFileURIToLoad (aUser : CoreASM-User, 
                           aFileStorage : CoreASM-FileStorage)
rule getNumForwardStepsRequested (aUser : CoreASM-User)
rule getStateIdBackTo (aUser : CoreASM-User)
rule getStateIdViewInHistory (aUser : CoreASM-User)
rule getUpdateSetIdViewInHistory (aUser : CoreASM-User)
rule createAWatch (aUser : CoreASM-User, aVocab : GUI.VOCAB)
rule selectAWatchToDel (aUser : CoreASM-User)

Spec 5.10: Rules in the GUI ↔ User interface

CASM.Engine include checking a specification in CASM.Engine, initializing a specification machine in CASM.Engine, forwarding, rollbacking, interrupting, creating a CASM.Engine and terminating a CASM.Engine [Spec 5.11].

The access interface enables CASM.GUI to access information entities about the CoreASM machine being executed in CASM.Engine, obtaining information such as the states of the CoreASM machine or the value of a function in the CoreASM machine [Spec 5.12].

rule checkSpecification (aEngine : CoreASM-Engine, aFile : File)
rule initSpecification (aEngine : CoreASM-Engine)
rule step (aEngine : CoreASM-Engine, an1 : Integer)
rule rollback (aEngine : CoreASM-Engine, anId : StateID)
rule createEngine
rule killEngine (aEngine : CoreASM-Engine)
controlled interrupt : CoreASM-Engine → Boolean
monitored getEngineMode : CoreASM-Engine → ENGINE-MODE

Spec 5.11: Rules and functions in the control interface of GUI ↔ Engine interface
5.6 Conclusion

The GUI ASM model has specified the architecture and the functionality of a GUI application system, with ASM methodologies. During the refinement process, the GUI ASM model has assisted to define interfaces between systems. Two objectives [Section 1.1] have been fulfilled in this chapter. The functions of the GUI application system are specified as twenty activities. Each activity represents a sequence of interactions. The internal components of CASM_GUI and the actors in the GUI ASM model are specified as parallel ASM or distributed ASM. Communications between internal components are transferred through the signal type communication structure. The exchanges of information between systems rely heavily on the interfaces of the systems.
CHAPTER 5. THE GUI ASM MODEL IN THE COREASM PROJECT

monitored \text{getGUI-Vocab} : \text{CoreASM-Engine} \rightarrow \text{GUI-Vocab}
monitored \text{getOutput} : \text{CoreASM-Engine} \rightarrow \text{GUI-Outputs}
monitored \text{getCurrentGUIState} : \text{CoreASM-Engine} \rightarrow \text{GUI-States}
monitored \text{getLastUpdateSet} : \text{CoreASM-Engine} \rightarrow \text{GUI-UpdateSets}
monitored \text{getPreviousGUIState} : \text{CoreASM-Engine}
  * \text{GUI-Watch} * \text{StateID}
  \rightarrow \text{GUI-States}
monitored \text{getPreviousUpdateSet} : \text{CoreASM-Engine}
  * \text{GUI-Watch} * \text{UpdateSetID}
  \rightarrow \text{GUI-UpdateSets}
monitored \text{getSpec} : \text{CoreASM-Engine} \rightarrow \text{File}

Spec 5.12: Rules in the access interface of GUI ↔ Engine interface
Chapter 6

Discussion

This chapter discusses in depth two issues that were carried out in the formal modeling process and gives solutions for these two issues. These two issues are level-refinement in the modeling process and the concurrency issue in the graphical user interface application.

6.1 Four abstraction levels for the GUI ASM model

At the beginning of the modeling stage, a system designer faced with too many informal requirements and no clear idea where to start might feel overwhelmed. One common methodology to overcome the previous problem is top-down development. ASM methodology gives the power of abstraction to top-down modeling, allowing the designer to model the system at different levels of abstraction. The system designer is able to focus on solving different questions at different levels of abstraction. The levels have a top-down relationship as the system designer refines the system model from a highly abstract level to a less abstract level. The goal of this refinement is to provide more implementation details to satisfy particular requirement specification and these implementation details should be appropriate to each level of abstraction.

The GUI ASM model developed for the CoreASM project has four abstraction
levels. The model has been refined from level 1 to level 4, in a top-down development process. The four abstraction levels of specification cover the CASM.GUI architectures, features, and interface design.

6.1.1 The first abstraction level - (the features)

The first abstraction level of the GUI ASM model specifies the activities to be performed by the system. The GUI system is viewed as a whole. The architecture of the GUI system is not analyzed at this level. The activities specified in the model at this point are user-oriented, intended to achieve certain user goals through the system operations. The activity, ViewLastStateActivity, would be an example of this. The objective of modeling at this abstraction level is to see what the system can do. The activities specified here are abstract activities. No implementation is specified as to how these activities are to be performed. These activities can be treated as major features of the GUI system. The model also specifies all signals that can activate the above activities.

6.1.2 The second abstraction level - (the interface design)

The second abstraction level of the GUI ASM model specifies the model's interactions. The main problems solved are what are the interactions between systems in the GUI ASM model and what are the data objects and control signals exchanged in these interactions. To answer these questions, the first step is to specify the architecture of the GUI ASM model and to identify the actors in it [Section 5.1].

An ASM model usually consists of a machine and its environment. As the research target of this thesis, the actor CASM.GUI is the machine in this model. Other actors, CASM.User, CASM.Engine, and File Storage, form the environment of the ASM machine. The interactions in the model are the interactions between CASM.GUI and its environment.

The analysis of the interactions and the data objects exchanged in these interactions is important. We can define API functions through these interactions, and the data objects that serve as their parameters. One of the objectives in building the GUI
ASM model is to define the APIs. The GUI ASM model can also verify these API functions as satisfying the requirements of the CoreASM GUI system.

6.1.3 The third abstraction level - (the internal architecture)

The third abstraction level of the GUI ASM model refines CASM.GUI by building the architecture of CASM.GUI itself [Section 5.3]. The internal components of CASM.GUI include the control panel, the output view, the message view, the program view, the run view, the state view, the updateset view, the history manager, and the vocab view. Each component provides certain features to be defined at this level of the GUI ASM model. A component in CASM.GUI is a collection of behaviors and functions. It is specified as an ASM rule in the model. A component is an abstract concept. At a later point, the GUI designer can find a GUI object to implement a component, such as JEditorPane in Swing or RichTextBox in C#, or even a television screen. In the ASM model, a component abstracts behaviors and functions from the actual objects. Therefore, a model using ASMs will not limit the GUI designer's imagination when it comes to visually representing that component.

The components are specified with Distributed ASM notation. Each component executes independently. All components execute in parallel inside CASM.GUI. These components are specified as distributed ASMs to satisfy concurrency issues which are intrinsic to GUI systems. More discussion of concurrency issues can be found in Section 6.2.2.

This third abstraction level introduces a signal communication structure to the GUI ASM model. Each component has one incoming signal pool. A component checks the signal in its signal pool at each step of the ASM's execution run and activates the relevant activities or operations if a signal is found. To communicate between components, a component inserts a signal into another component's signal pool. Two kinds of information are contained in a signal: the command that indicates what activities or operations should be activated by the signal, and the data objects that are needed for execution of the activities or operations activated. As an example, the activity InitActivity involves the interaction of certain components, including the
control panel, the message view, run view, the state view, the history manager and
the vocab view. After initialization, the control panel will update the vocab view.
The control panel sends the vocab view a request to update the vocabulary through
the command \texttt{vocabvcDisp}. The data object vocabulary, \texttt{guiVOCAB}, is attached
in the same request signal. A similar mailbox-like communication mechanism in a
distributed system is specified in [15] [16].

6.1.4 The fourth abstraction level - (the concurrency problem)

The model designer can discuss individual issues at this abstraction level generally.
In this GUI ASM model, we pay attention to the problem of two concurrent activities
taking place in CASM_GUI. The GUI application is a multithreading application. It
can respond to user requests simultaneously; therefore one activity can be activated
before a previous activity has been completed. One question arises, what kind of
activities in CASM_GUI can be executed simultaneously? For example, it does
not seem reasonable that \texttt{ForwardRunActivity} and \texttt{RollbackActivity} can be performed
concurrently. Not all activities can execute concurrently with other activities. The
GUI ASM model will specify this constraint as part of the fourth abstraction level
[Section 6.2.3].

6.1.5 An example of refining a model in four abstraction levels

The activity \texttt{ForwardRunActivity} serves here as an example to show how to specify
and refine an interaction in four abstraction levels. In the activity, the user requires
the machine in the engine to execute forward a few of steps, and then the GUI
displays the results of these steps. This activity involves three actors, CASM_User,
CASM_GUI and CASM_Engine, and consists of three sub-activities, \texttt{getNumForward-
StepsRequestedSA}, \texttt{forwardRunSA} and \texttt{getFRFeedbackSA}

The first abstraction level of the GUI ASM model specifies activities performed
by CASM_GUI. The activity *ForwardRunActivity* is defined as an ASM rule. That means this activity is a set of operations in CASM_GUI. At this abstraction level, we specify that this activity exists. The details of the operations in the activity are not of concern at this level. Therefore the operations are not specified here.

```
//Activity Rule
rule ForwardRunActivity
```

Spec 6.1: The activity *ForwardRunActivity*

At this abstraction level, the GUI ASM model also specifies how the activity is activated. The command in the signal, *userForwardRun*, activates the activity *ForwardRunActivity*. This happens inside the actor CASM_GUI.

```
// -- Main rule of the CoreASM GUI --
main rule CoreASMGUIProgram =

//Signal handling by CASM_GUI
case nextUserActivityRequest( ) of
  userForwardRun →
    rule ForwardRunActivity
```

Spec 6.2: The activity *ForwardRunActivity* is activated in the main rule

At the second abstraction level of the GUI ASM model, there are two objectives: to define the APIs and to verify their necessity in the model. Analyzing the interactions in the system is the beginning point for defining APIs. The first step is to specify the actors in these interactions, CASM_Engine, CASM_User and File Storage, in addition to CASM_GUI itself.
Spec 6.3: The actors in the activity ForwardRunActivity

Next, we refine the activity ForwardRunActivity by analyzing this interaction. The activity ForwardRunActivity has three sub-activities, getNumForwardStepsRequestedSA and forwardRunSA and getFRFeedbackSA. They execute in sequential order.

Spec 6.4: The sequential execution of sub-activities in the activity ForwardRunActivity

The sub-activity is also an ASM rule. We specify API functions and their parameters in each sub-activity rule. The sub-activity getNumForwardStepsRequestedSA gets the number of steps that CASM_User wants the machine to go forward. This is an interaction between CASM_GUI and CASM_User. The data object guiNumStepsToForward is needed in the sub-activity. The function monitored getNumForwardStepsRequested : CoreASM-User → Integer obtains the data object needed from CASM_User.
Spec 6.5: The sub-activity `getNumForwardStepsRequestedSA`

The second sub-activity `forwardRunSA` forwards CASM.Engine `guiNumStepsToForward` steps, by executing the rule `step(currentEngine, guiNumStepsToForward)` defined in the GUI $\leftrightarrow$ Engine interface.

Spec 6.6: The sub-activity `forwardRunSA`

The third sub-activity `getFRFeedbackSA` makes CASM.GUI to receive feedback from CASM.Engine about the state, the update set and the output.

Spec 6.7: The sub-activity `getFRFeedbackSA`

After verifying the functions specified in the activities, we can list these functions safely at the beginning of the GUI ASM model.
CHAPTER 6. DISCUSSION 70

/

//Vocabulary of the model CoreASM - GUI

vocabulary :

//Interface of GUI

//User ← GUI Functions

//get the number of steps that the CASM_User requests
//the model to forward
rule getNumForwardStepsRequested (aUser : CoreASM-User)

//GUI ← Engine Functions

// -- Control Functions

monitored step : CoreASM-Engine × Integer → Boolean

// -- Access Functions

monitored getOutput : CoreASM-Engine → GUI_Outputs
monitored getCurrentGUI_State : CoreASM-Engine → GUI_States
monitored getLastUpdateSet : CoreASM-Engine → GUI_UpdateSets

Spec 6.8: API functions used in the activity ForwardRunActivity

The ASM rules and the ASM functions in the code section GUI ← Engine Interfaces are API functions we need to define in this project.

At the third abstraction level, the GUI ASM model has specified the architecture of CASM_GUI and has introduced a communication structure between the internal components of CASM_GUI. We now need to answer the following questions. What internal components of CASM_GUI are involved in the activity ForwardRunActivity? How do these components react in the activity ForwardRunActivity? What behaviors do these components perform to complete the activity ForwardRunActivity?

The GUI ASM model has to be refined to answer the above questions.

The activation of the activity ForwardRunActivity is managed in the internal component control panel. The component control panel executes in parallel with other internal components in CASM_GUI. Therefore, the main rule of CASM_GUI, CoreASM_GUIProgram, has been written and refined as parallel executions of internal
components (Spec 6.2 to Spec 6.9). The component control panel handles activations of activities.

```
main rule  CoreASMGUIProgram =
          VocabViewProgram
          OutputViewProgram
          MessageViewProgram
          ProgramViewProgram
          RunViewProgram
          StateViewProgram
          UpdateSetViewProgram
          HistoryManagerProgram
          ControlPanelProgram

rule ControlPanelProgram
  //Signal handling by control panel in CASM_GUI
  case nextUserActivityRequest() of

    userForwardRun →
    ForwardRunActivity
```

Spec 6.9: The activity ForwardRunActivity is activated in the control panel

We add a group of operations into the activity rule ForwardRunActivity. The component control panel requests other components in CASM.GUI to display the results after the sub-activity forwardRunSA completes. The display requests are sent through the communication structure by signaling other internal components in CASM.GUI.

Other components receive the signal and then perform the display requests. For example, the component state view finds that the request signal is the statevcDisp command. Then the state view executes the rule displayStateInStateV(secondOf(aSignal)) to display the new state.

As the third abstraction level of the GUI ASM model shows, a system designer can find answers for the questions asked in the page 70. The internal components control panel, message view, run view, state view and history manager are involved in the
Spec 6.10: Sending requests to other components in activity ForwardRunActivity

The fourth abstraction level of the GUI ASM model concentrates on solving the activity concurrency problem. Some kinds of activity in CASM_GUI cannot be interrupted by another activity and must be completed before quitting (The activity InterruptActivity is a special case). The GUI system specified in this model is a multithread application. It can respond to multiple user requests simultaneously. Therefore activity concurrency exists in this GUI ASM model. In this kind of multithread application, we know at least one constraint exists: the activity ForwardRunActivity cannot start when CASM_Engine is executing forwarding or rollbacking actions. When refining this abstraction level, we add a constraint into the GUI ASM model. The control panel checks whether a forward or rollback action is executing before
CHAPTER 6. DISCUSSION

Spec 6.11: The component state view

activating an activity *ForwardRunActivity*. The checking is done in the function *isSynchronousActivityRunning()*.

Spec 6.12: The constraint to activate the activity *ForwardRunActivity*

We have assigned each activity a flag (guiSAMode_***A) to label the activity states *has-not-started*, *in-process* and *completed*. The function *isSynchronousActivityRunning()* checks whether an initialization or execution activity, such as activity *ForwardRunActivity* or activity *RollbackActivity*, is executing in CASM_GUI. The flag helps the control panel to detect the activity execution. We refine the activity rule.
ForwardRunActivity by assigning the activity state guiACompleted to the activity flag guiSAMode FORWARDRUNA at the end of the activity ForwardRunActivity.

\[
\text{guiSAMode\_ForwardRunA} := \text{guiACompleted}
\]

Spec 6.13: State of the activity ForwardRunActivity

The refinement of the activity ForwardRunActivity in the four different abstraction levels of GUI ASM model is a process to make design decisions. These design decisions help to build a system that satisfies the requirements. The GUI ASM model has answered the following questions during the refinement process. Is the activity ForwardRunActivity one behavior of CASM.GUI? How does CASM.GUI interact with other actors (the environment) in the activity ForwardRunActivity and what are the API functions in the interaction? What internal components in CASM.GUI are involved in the activity ForwardRunActivity and what roles do they play? Can the activity ForwardRunActivity execute in parallel with other activities in CASM.GUI?

6.1.6 Benefit of refinement by levels

When a system designer specifies a complex system, he or she would probably like to start from a high level of design; during the system development life cycle, the designer will also likely prefer that the specified system model be easily modified several times, whenever new detailed questions arise or when the requirements are changed.

This thesis introduces a mixed model-building approach that applies ASM methodology [Section 4.2]. In this approach, the system designer starts with the traditional object-oriented approach to build the architecture of the system as a whole and to point out the actors in the system and the roles of these actors. The system designer takes a global view of the complex system and decides its boundaries. Then the system designer switches to a task-based approach, analyzing the tasks that the system can perform. The task analysis at this stage is user-oriented if the system
is interactive. The tasks analyzed in the model should complete certain user goals. The specified model receives continuing refinement at each of the different abstraction levels, enabling the designer to analyze the behaviors of the system and to match specification requirements. Each abstraction level should focus on a specific group of questions. The system designer should decide the number of abstraction levels and objectives of each abstraction level depending on the projects.

A mixed development approach helps system designers to specify a complex system model from the top to the bottom. System designers can think about the main objectives at the beginning and prevent marginal and detailed questions from obstructing decision making. ASM methodology is another tool to assist system designers. It has the power to abstract behaviors and functions of objects at a high level. In a graphical user interface system, ASM methodology can abstract an object in a model's behaviors and functions from its appearance. This type of specification model leaves the graphics designer a considerable amount of freedom in designing the interface's graphic appearance. The refinement process in the ASM methodology also makes it possible to modify the model if the requirements should change.

6.2 Concurrency issues in the GUI system

A GUI system is normally a multithreading system. The concurrency issues in GUI application design can be solved by defining a few constraints and by taking advantage of distributed ASMs.

6.2.1 GUI ASM model constraints in concurrency issue

In some activities, CASM.GUI would send requests to an outside actor in the environment, such as CASM.Engine, and wait for a response. It may take a long time for CASM.GUI to receive a response. For instance, requesting CASM.Engine to execute a step and this step takes a long time. During this waiting time, CASM.GUI is supposed to be available to respond to some user requests, such as a request to view program code.
Activity Concurrency Policy:

1. CASM_GUI cannot receive any user request for execution or initialization when a user request for execution or initialization is already in process;

2. Multiple instances of an activity cannot be processed in CASM_GUI simultaneously;

We have specified that there is only one CASM_User in the system environment. This single CASM_User interacts with a single CASM_GUI. CASM_User cannot execute the same activities simultaneously with one CASM_GUI. It is safe to say two activities, for example two uses of ViewProgramCodeActivity, initiated by one user will not occur at the same time.

There is another situation needed to be discussed here. The case might exist where CASM_User requests CASM_GUI to process an activity when the previous instance of the same activity is in process. For example, CASM_User might change his or her mind and want to view a previous state record when he or she is already viewing a different previous state record. Both activities are the activity ViewStateInHistory-Activity. In this situation, the second activity instance interrupts the previous one if this activity instance is the same kind of view activity. We introduce the policy 2 to avoid the situation that might occur if multiple instances of one activity simultaneously execute. The second instance of an activity cannot be processed if the previous activity instance is still alive (Spec 6.14). This keeps the GUI ASM model simple at this fourth level of abstraction. Execution or initialization activities, such as InitActivity, ForwardRunActivity, RollbackActivity and so on, are non-interruptible by user requests for the same activity. The policy 2 satisfies the needs of initialization or execution activities too.

One execution or initialization activity should not be interrupted by the same activity, and it normally should not be interrupted by other execution or initialization activities, either (there is an exception: userInterrupt). To satisfy these two requirements, there are two solutions. One is to hold the second execution or initialization activity user request in CASM_GUI until the previous execution or initialization activity has been completed. The second solution is to ensure that CASM_GUI cannot
Spec 6.14: The activation of an activity

receive another execution or initialization activity user request during the period of
time when CASM_GUI is dealing with an execution or initialization activity. For this
GUI ASM model, the second option was selected (policy 1). At times, a GUI designer
normally sets up the system to close the input channel to prevent CASM_User from
sending an activity user request. For example, at certain times, the Forward button
might be disabled.

6.2.2 GUI ASM model components

The GUI ASM model in this thesis consists of nine components expressed as ASM
rules, seven ViewPrograms and one HistoryManagerProgram and one ControlPanel-
Program. All components execute in parallel. They are specified as distributed
abstract state machines (DASMS).

The ViewProgram provides functions for CASM_User to view the executing
specification model on CASM_GUI. The HistoryManagerProgram manages the state
history records and update set history records in CASM_GUI. The ControlPanel-
Program is the program controlling CASM_GUI. It has interfaces for the environ-
ment (CASM_User, CASM_Engine, File Storage) to interact with CASM_GUI. One
important task for the control panel is to respond to UserActivityRequests from
CASM_User and to activate the corresponding activities in CASM_GUI. In GUI
design, the control panel is a collection of GUI controls located on the GUI, including
the initializing button, the step button, etc.
CHAPTER 6. DISCUSSION

All components should work in parallel. One component in an execution state does not block another component’s execution. Each component is an ASM rule and has its own behaviors. All components execute in parallel according to the main rule \textit{CoreASMGUIProgram} of CASM.GUI (Spec 6.15).

\begin{verbatim}
// -- Main rule of the CoreASM GUI --
main rule CoreASMGUIProgram =
  VocabViewProgram
  OutputViewProgram
  MessageViewProgram
  ProgramViewProgram
  RunViewProgram
  StateViewProgram
  UpdatesetViewProgram
  HistoryManagerProgram
  ControlPanelProgram
\end{verbatim}

Spec 6.15: The parallel executing components

The component \textit{control panel} is specified as the only component in CASM.GUI to handle user requests from CASM.User and to send and receive responses from CASM.Engine. The \textit{control panel} dispatches activities in response to user requests. It is this approach that guarantees that the processes are thread-safe.

6.2.3 Activities in ControlPanelProgram

The GUI component \textit{control panel} dispatches activities in response to user requests and controls these activities. On one step of ASM rule \textit{ControlPanelProgram}, \textit{control panel} handles one user request and dispatches the corresponding activity. The second response of the \textit{control panel} is to keep uncompleted previous activities executing.

There are two kinds of activities in CASM.GUI, synchronous activity and asynchronous activity. A synchronous activity is an activity that should be executed
without interruption from another activity. For example, ForwardRunActivity has to complete before RollbackActivity starts. ForwardRunActivity is a synchronous activity. Most synchronous activities in this GUI ASM model are those activities that involve execution by CASM.Engine. By contrast, InterruptActivity is a special activity. It can interrupt other synchronous activities.

In the ViewStateInHistoryActivity, CASM.GUI gets the state from the history manager and displays that state in the state view. This activity does not need CASM.GUI to communicate with CASM.Engine. Therefore, we treat the activity ViewStateInHistoryActivity as an asynchronous activity. An asynchronous activity is an activity that can be interrupted by another activity and the interruption will not affect the execution of the CoreASM machine.

<table>
<thead>
<tr>
<th>Asynchronous Activity</th>
<th>Synchronous Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewProgramCodeActivity</td>
<td>LoadFileActivity</td>
</tr>
<tr>
<td>ViewLastOutputActivity</td>
<td>CheckSpecActivity</td>
</tr>
<tr>
<td>ViewOutputInHistoryActivity</td>
<td>InitActivity</td>
</tr>
<tr>
<td>ViewLastMsgActivity</td>
<td>ForwardRunActivity</td>
</tr>
<tr>
<td>ViewMsgInHistoryActivity</td>
<td>RollbackActivity</td>
</tr>
<tr>
<td>ViewLastUpdatesetActivity</td>
<td>StopActivity</td>
</tr>
<tr>
<td>ViewLastStateActivity</td>
<td>InterruptActivity *</td>
</tr>
<tr>
<td>ViewStateInHistoryActivity</td>
<td></td>
</tr>
<tr>
<td>StartupActivity</td>
<td></td>
</tr>
<tr>
<td>QuitActivity</td>
<td></td>
</tr>
<tr>
<td>AddWatchActivity</td>
<td></td>
</tr>
<tr>
<td>DeleteWatchActivity</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Asynchronous activities and synchronous activities

An activity is activated by a user request in the control panel. If this activity is an asynchronous activity, control panel allows the activity to process when the same activity is not being executed at the same time (guiSAMode_****A = undef) (Spec 6.16) (Policy 2).

If the activity is a synchronous activity, control panel allows that activity to be
case nextUserActivityRequest() of
  // if the model is not processing the activity ViewStateInHistory,
  // then process it.

  userViewStateInHistory →
    if guiSAMode.ViewStateInHistoryA = undef then
      ViewStateInHistoryActivity

Spec 6.16: The constraint to activate an asynchronous activity

activated only when no other synchronous activity is executing in CASM.GUI at the

time (Spec 6.17).

case nextUserActivityRequest() of
  // if the model is not processing a synchronous activity
  // then process it.

  userForwardRun →
    if not isSynchronousActivityRunning() then
      ForwardRunActivity

Spec 6.17: The constraint to activate a synchronous activity

The next task for control panel is to keep uncompleted activities in CASM.GUI executing. The GUI ASM model applies parallel ASMs here again and executes all uncompleted activities in parallel (Spec 6.18)

An asynchronous activity does not require the result of the activity to display on the GUI immediately. Two asynchronous activities can overlap each other. For instance, a user can interrupt a viewing state and switch to viewing program code, then switch back. The CASM.GUI executes the uncompleted activities ViewProgramCodeActivity and ViewStateInHistoryActivity in parallel to specify this situation
(Spec 6.18). The only constraint is to make sure no identical activity is executing currently (Spec 6.16). This does satisfy the policy 2.

A synchronous activity cannot be interrupted by another synchronous activity. The policy 1 is defined to satisfy this requirement. The system checks whether a synchronous activity is being executed currently before activating a synchronous activity. Two synchronous activities, ForwardRunActivity and RollbackActivity, seem as if they can execute in parallel (Spec 6.18). In fact, this will not happen given the constraint (Policy 1).

One synchronous activity may take a long time to complete and at the same time it may hold an important resource, an outside actor, such as CASM_Engine or File Storage. This produces one problem. The CASM_User may want to stop a synchronous activity if it takes too long. InterruptActivity is introduced as a special synchronous activity to interrupt other synchronous activities.

An asynchronous activity and a synchronous activity are allowed to be executed simultaneously. For example, a user can view program code while waiting for the engine processes one step. The activities ViewProgramCodeActivity and ForwardRunActivity execute in parallel (Spec 6.18). CASM_GUI will not freeze when a synchronous activity is running. The control panel will activate an asynchronous activity when a user request for the asynchronous activity is incoming, even if a synchronous activity is running in CASM_GUI (Spec 6.16). A synchronous activity will not block other incoming asynchronous activity requests.

6.3 Summary

This chapter discussed the level-refinement during the modeling process and the concurrency issue in the graphical user interface application.

Level-refinement refined the GUI ASM model at abstraction levels in a top-down development process. Each abstraction level had particular requirement specification goals. The activity ForwardRunActivity was used as an example to explain the level-refinement process.

Concurrency issue normally exists in a GUI system. The GUI ASM model took
advantage of distributed ASMs to model GUI components that execute in parallel. An activity concurrency policy was also given in the GUI ASM model to solve the activity concurrency problem in GUI systems.
CHAPTER 6. DISCUSSION 83

//synchronous activity
if guiSAMode_RollbackA = guiACompleted then
  // quits this activity
  guiSAMode_RollbackA := undef
else
  // if the activity is in process, continue the activity
  if not (guiSAMode_RollbackA = undef) then
    RollbackActivity
//synchronous activity
if guiSAMode_FowardRunA = guiACompleted then
  // quits this activity
  guiSAMode_FowardRunA := undef
else
  // if the activity is in process, continue the activity
  if not (guiSAMode_FowardRunA = undef) then
    ForwardRunActivity
//asynchronous activity
if guiSAMode_ViewProgramCodeA = guiACompleted then
  // quits this activity
  guiSAMode_ViewProgramCodeA := undef
else
  // if the activity is in process, continue the activity
  if not (guiSAMode_ViewProgramCodeA = undef) then
    ViewProgramCodeActivity
//asynchronous activity
if guiSAMode_ViewStateInHistoryA = guiACompleted then
  // quits this activity
  guiSAMode_ViewStateInHistoryA := undef
else
  // if the activity is in process, continue the activity
  if not (guiSAMode_ViewStateInHistoryA = undef) then
    ViewStateInHistoryActivity

Spec 6.18: Synchronous activities and asynchronous activities
Chapter 7

Implementation

This chapter introduces the implementation of the GUI ASM model, a visual CoreASM language debugger, and then experiments an actual CoreASM model, the ATM model.

7.1 Visual CoreASM language debugger

A visual CoreASM language debugger is currently under development. The GUI of the debugger is the implementation of CASM_GUI described in Chapter 5.

The programming language used to implement the debugger is Java. The reason to choose Java as the programming language is to allow the visual debugger to be cross-platform. The minimum Java virtual machine required to execute the GUI of the debugger is the version 1.2 or higher. The GUI is implemented with the Swing technology.

7.1.1 Function areas on the GUI

There are four main function areas on the GUI.

- Program Edit Area: the area where the user can edit the code of a CoreASM
Figure 7.1: The graphical user interface of the visual CoreASM language debugger

- **Machine View Area**: the area that displays the properties of a CoreASM machine. The properties of the machine are vocabulary, states and update sets. This area locates on the upper-right corner of the GUI (labeled by 2 in Figure 7.1).

- **Execution Information Area**: the area that displays the information of the execution of a CoreASM machine. The information includes the messages about the execution of the machine and the output of the machine and the history of a run of the machine. This area locates on the lower-left corner of the GUI (labeled by 3 in Figure 7.1).

- **Control Panel Area**: the area where the user can operate to execute a CoreASM machine in. This area locates on the lower-right corner of the GUI (labeled by 4 in Figure 7.1).
CHAPTER 7. IMPLEMENTATION

The views of most components of CASM_GUI can be found on this GUI implementation. The table 7.1 shows these components and the function areas where the views of these components are in.

<table>
<thead>
<tr>
<th>CASM_GUI Component</th>
<th>Function Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>program view</td>
<td>Program Edit Area</td>
</tr>
<tr>
<td>vocab view</td>
<td>Machine View Area</td>
</tr>
<tr>
<td>state view</td>
<td>Machine View Area</td>
</tr>
<tr>
<td>updatesset view</td>
<td>Machine View Area</td>
</tr>
<tr>
<td>message view</td>
<td>Execution Information Area</td>
</tr>
<tr>
<td>output view</td>
<td>Execution Information Area</td>
</tr>
<tr>
<td>run view</td>
<td>Execution Information Area</td>
</tr>
<tr>
<td>control panel</td>
<td>Control Panel Area</td>
</tr>
<tr>
<td>history manager</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: The components and the function areas in where the views of these components are

The history manager is invisible to CASM_User. Therefore, it does not have a view on the GUI.

7.1.2 Information organized in a tree structure

Users learn a CoreASM machine by observing the states and the updates of the machine in a run. A state is a set of elements together with functions. An update set in an update is a set of locations and elements. The function signatures can be used to identify the locations. Therefore, all the information items in a state or an update set can be grouped by functions. A tree view is a good representation of this kind of information organization.

In Figure 7.2, the information of the state is classified to three groups, the function stepNum, the function isEven and the function iconColor. Four locations, isEven(0), isEven(1), isEven(2), and isEven(3) are in the group of the function isEven.

The information is organized in a tree structure. This applies to both views, the state view and the update set view. The purpose of the information organization is
CHAPTER 7. IMPLEMENTATION

7.1.3 Run view

The run view displays the trail of a run of a CoreASM machine. A user clicks one state button in the run view, then the information about the machine at that state will display in the state view and in the update set view.

for users to find a particular location and its value easily.
CHAPTER 7. IMPLEMENTATION

7.1.4 Control panel

In the GUI ASM model, the component control panel responds all activity requests from CASM.User. In the implementation, the panel in the Control Panel Area on the GUI is only a partial view of the component control panel, because most, but not all, of the activity request buttons can be found in the Control Panel Area. The initialization request button and the execution request buttons are in the Control Panel Area. The other activity requests, such as view requests, are going to be received in the other function areas on the GUI. For instance, a user needs to click on one state button in the run view if he/she wants to view the history record about that state.

7.2 Experiment of the ATM model

We chose an actual CoreASM model to test the GUI implemented in Java. This CoreASM model is the ATM model.
7.2.1 ATM model

The ATM model abstractly models a cash machine control. There are three separate active entities that are involved in ATM operations. The active entities are an ATM manager, and an authentication manager, and an account manager. To simplify the model, the ATM manager is specified as a CoreASM machine, and the authentication manager and the account manager are in the system environment.

The withdrawal operation of the ATM should follow these steps:

1. Input the bank card and the PIN code.
2. Check the validity of the bank card and the PIN code.
3. Input the amount to be withdrawn.
4. Check the account balance against the credit line.
5. On approval update the account balance.
6. Output cash or notification about denial of transaction.

The timeout mechanisms of the ATM may cause the cancellation of transactions at any time.

The ATM model is given in Spec 7.1, with refinement. So the ATM model can be experimented in the visual CoreASM language debugger [Section 7.1].

There are a few simplifying assumptions in this ATM model.

- The bank card and the PIN code are input in the same step.
- The bank card and the PIN code are always authenticated.
- The transaction is always valid.
- The mode of the ATM is always idle at the initial state.
- It is known in advance at the initial state whether CancellationEvent will occur during the withdrawal operation.
7.2.2 Experiments

The ATM model is ready to be executed in the visual CoreASM language debugger. To experiment the perspective of the ATM on the withdrawal operation, a few test cases were prepared.

**Test case: successful operation**

The ATM is in the idle mode and is activated at the initial state. No \textit{CancellationEvent} will occur.

The initial state in the test case:

```
init :
  mode := idle
  isActivated := true
  isCancelled := false
```

The states of the ATM model in a withdrawal operation are displayed in Figure 7.5 on the page 93.

The output of the ATM model in a withdrawal operation are displayed in Figure 7.6 on the page 94.

In this test case, the ATM manager starts from the idle mode, then turns into the processing mode, finally goes back to the idle mode. This is a test case for a successful operation. The changes of the mode are same as the perspective of the ATM. The output of the ATM model in the second step [Figure 7.6] also matches the perspective of the ATM.

**Test case: inactivated ATM**

The ATM is in the idle mode, but it is not activated at the initial state. No \textit{CancellationEvent} will occur.
CHAPTER 7. IMPLEMENTATION

The initial state in the test case:

```
init :
  mode := idle
  isActivated := false
  isCancelled := false
```

The states of the ATM model in a withdrawal operation are displayed in Figure 7.7 on the page 95.

There is no output of the ATM model in a withdrawal operation.

In this test case, the ATM manager is inactivated. The mode keeps being idle during steps. No further actions are to be performed. This matches the perspective of the ATM. The withdrawal operation cannot be performed if the ATM is not activated.

7.2.3 Conclusion

In the previous experiment, it shows that the visual CoreASM language debugger can be used to experiment the perspective of a system by running test cases. The test cases are developed from the scenarios that written in the specification phase [Figure 2.6]. If the ATM model gets further refinement, for instance, refining the function isAuthenticated, more test cases can be prepared. Then, the designer can explore more behaviors of the ATM.
asm ATMmanager

vocabulary :
    enum domain Mode = {idle, processing}

definitions :
    main rule CoreASMGUIProgram =
        if Idle and ActivationEvent then
            data := getCardData
            code := getPinCode
            amount := getAmount
            mode := processing
        if Processing and IsAuthenticated(data, code) then
            if IsValidTransaction(data, amount) then
                ReleaseCash(amount)
                UpdateAccountBalance(data, amount)
            else
                OutputCancellationNotification
            mode := idle
        if Processing and ( not (IsAuthenticated(data, code)) or CancellationEvent) then
            OutputCancellationNotification
        mode := idle
    where
        rule Idle
            mode = idle
        rule Processing
            mode = processing
        rule ActivationEvent
            isActivated = true
        rule CancellationEvent
            isCancelled = true
        rule IsAuthenticated(data, code)
            isAuthenticated(data, code) = true
        rule IsValidTransaction(data, amount)
            isValidTransaction(data, amount) = true

Spec 7.1: The ATM model
CHAPTER 7. IMPLEMENTATION

Figure 7.5: The states in a successful run
CHAPTER 7. IMPLEMENTATION

Releash cash.
Update account balance.

Figure 7.6: The output in the second step in a successful run
Figure 7.7: The states of an inactivated ATM
Chapter 8

Conclusion

8.1 Conclusion

This thesis presents the work of designing a GUI application system at a high abstract level with model-based design. Abstract state machines methodologies have been applied in the modeling process. The GUI ASM model is specified with the CoreASM language. It provides a formal mathematical foundation to specify the architecture and the function form of the GUI system and to specify the interactive actions between the user and the CoreASM engine.

The formalization of the functions of the GUI system shows the ASM model is a precise semantic foundation to model human-computer interaction system at a high abstract level. The GUI ASM model describes the architecture of the system and specifies the functions of the system as activities. This model is a formal foundation for analysts to analyze GUI system behaviors and is a formal model for designers to validate the GUI system by walk-through inspection. The inspection can be switched to experimental validation through simulation and testing and formal verification when the CoreASM working environment is ready.

The GUI ASM model is built with a mixed design approach of the object-oriented approach and the task-oriented approach [Section 4.2]. The object-oriented design
creates the architecture of the CoreASM supporting tool environment and the architecture of the GUI application system (CASM_GUI) at the beginning. The task-oriented design guides the system designer to focus on one core object in the GUI system, CASM_GUI. As the design process progresses, level-wise refinement can push the GUI ASM model to become more detailed. An ASM model is scalable. With multiple levels, each level is concerned with particular features of the system and keeps other features abstracted. The higher level of specification can be easier proved. The lower level needs only be proven correct with respect to the previous higher level. Level-wise refinement makes the design process traceable and manageable.

One challenge of this project research in the graphical user interface design is to present users with information about execution of a CoreASM machine. This challenge consists of two questions. 1. What is this information? 2. How should this information be presented visually? The thesis shows a way to collect the information in the second abstraction level of the GUI ASM model as well as a way to create APIs (GUI → Engine interface) with these information. The Java implementation of the graphical debugger for the CoreASM language is an example of the visual presentation of this set of information.

8.2 Future work

The GUI ASM model in this thesis has captured the architecture and functions of the GUI application system. This model is built on the informal requirements of a GUI application system in the CoreASM project. To define the requirements for interactive systems, William M. Newman and Michael G. Lamming suggested that the requirements should be defined in such areas: defining functional form, identifying the users and setting performance requirements [24]. The GUI ASM model in this thesis has specified the functional form of the GUI system and has identified the users. The users are specified as CASM_User in the GUI ASM model and are identified (/abstracted) by their behaviors and functions that they provide for CASM_GUI. More studies can be done to formally model the users in their behaviors and to analyze the interactions between the users and the GUI system. To specify the performance
requirements with ASMs would be another challenge.
Appendix A

List of terms used in the GUI ASM model

This appendix presents the terms defined in the GUI ASM model.

A.1 Actors in the GUI ASM model

The actors:

- CASM_User
- CASM_GUI
- CASM_Engine
- File Storage

A.2 Components of CASM_GUI

The internal components of CASM_GUI:

- control panel
- output view
A.3 Signal pools in the components

The following table shows the signal pool in each component.

<table>
<thead>
<tr>
<th>Signal Pool</th>
<th>Signal Pool Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlpanelSignalPool</td>
<td>controlpanel</td>
<td>control panel</td>
</tr>
<tr>
<td>outputvSignalPool</td>
<td>outputv</td>
<td>output view</td>
</tr>
<tr>
<td>messagevSignalPool</td>
<td>messagev</td>
<td>message view</td>
</tr>
<tr>
<td>programvSignalPool</td>
<td>programv</td>
<td>program view</td>
</tr>
<tr>
<td>runvSignalPool</td>
<td>runv</td>
<td>run view</td>
</tr>
<tr>
<td>vocabvSignalPool</td>
<td>vocabv</td>
<td>vocab view</td>
</tr>
<tr>
<td>statevSignalPool</td>
<td>statev</td>
<td>state view</td>
</tr>
<tr>
<td>updatesetvSignalPool</td>
<td>updatesetv</td>
<td>updateset view</td>
</tr>
<tr>
<td>historymanagerSignalPool</td>
<td>historymanager</td>
<td>history manager</td>
</tr>
</tbody>
</table>

Table A.1: Signal pools in components

A.4 Activities in CASM_GUI

The activities in CASM_GUI are classified into two groups, synchronous activity and asynchronous activity.
The synchronous activities:
- LoadFileActivity
- CheckSpecActivity
- InitActivity
- ForwardRunActivity
- RollbackActivity
- StopActivity
- InterruptActivity

The asynchronous activities:
- ViewProgramCodeActivity
- ViewLastOutputActivity
- ViewOutputInHistoryActivity
- ViewLastMsgActivity
- ViewMsgInHistoryActivity
- ViewLastUpdatesetActivity
- ViewLastStateActivity
- ViewUpdatesetInHistoryActivity
- ViewStateInHistoryActivity
- StartupActivity
- QuitActivity
- AddWatchActivity
- DeleteWatchActivity

Each activity has three modes. Each mode is defined as the following.
- undef: The activity does not exist in the machine.
- guiAInProcess: The activity is in process
- guiACompleted: The activity is completed, but still exists in the machine.
A.5 Signals in CASM_GUI

Signals are used to activate certain activities.

The types of these signals are,

- USER_ACTIVITYREQUEST
- PROGRAMV_COMMAND
- MESSAGEV_COMMAND
- OUTPUTV_COMMAND
- RUNV_COMMAND
- STATEV_COMMAND
- UPDATESETV_COMMAND
- HISTORYV_COMMAND
- VOCABV_COMMAND

A.5.1 USER_ACTIVITYREQUEST

These signals are the signals that activate the activities of CASM_GUI. They are sent by CASM_User.

The signals are,

- userStartup
- userLoadFile
- userCheckSpec
- userInit
- userForwardRun
- userRollback
- userStop
- userInterrupt
- `userViewProgramCode`
- `userViewLastOutput`
- `userViewOutputInHistory`
- `userViewLastMsg`
- `userViewMsgInHistory`
- `userViewLastUpdateset`
- `userViewLastState`
- `userViewUpdatesetInHistory`
- `userViewStateInHistory`
- `userAddWatch`
- `userDeleteWatch`
- `userQuit`

### A.5.2 PROGRAMV.COMMAND

These signals are the signals that activate the activities of program view. They are received by program view.

The signals are,

- `programvcDisplayFile`: to display the CoreASM machine code in program view;
- `programvcClearDisplay`: to clear the display in program view.

### A.5.3 MESSAGEV.COMMAND

These signals are the signals that activate the activities of message view. They are received by message view.

The signals are,

- `messagevcDisp`: to display the warning/error messages in message view;
- `messagevcClearDisplay`: to clear the display in message view.
A.5.4 OUTPUTV.COMMAND

These signals are the signals that activate the activities of output view. They are received by output view.

The signals are,

- outputvcDisp: to display the printout in output view;
- outputvcClearDisplay: to clear the display in output view.

A.5.5 RUNV.COMMAND

These signals are the signals that activate the activities of run view. They are received by run view.

The signals are,

- runvcAddAStep: to add a new state icon in run view;
- runvcDelSteps: to delete icons in run view;
- runvcHighlightAStep: to highlight an icon in run view;
- runvcClearDisplay: to clear the display in run view.

A.5.6 STATEV.COMMAND

These signals are the signals that activate the activities of state view. They are received by state view.

The signals are,

- statevcDisp: to display a state of the CoreASM machine in state view;
- statevcClearDisplay: to clear the display in state view.

A.5.7 UPDATESETV.COMMAND

These signals are the signals that activate the activities of updateset view. They are received by updateset view.
The signals are,

- updatesetvcDisp: to display an update set in updateset view;
- updatesetvcClearDisplay: to clear the display in updateset view.

A.5.8 HISTORYV_COMMAND

These signals are the signals that activate the activities of history manager. They are received by history manager.

The signals are,

- historymanagercAddAStep: to add a history record into history manager;
- historymanagercDelSteps: to delete history records from history manager;
- historymanagercClear: to clear all history records in history manager.

A.5.9 VOCABV_COMMAND

These signals are the signals that activate the activities of vocab view. They are received by vocab view.

The signals are,

- vocabvcDisp: to display the vocabulary of the CoreASM machine in vocab view;
- vocabvcClearDisplay: to clear the display in vocab view.
Appendix B

Abstract model of CASM_GUI

This appendix presents the GUI ASM model at the fourth abstraction level.

B.1 Actors in the GUI ASM model

There are four actors in the GUI ASM model, CASM_User, CASM_GUI, CASM_Engine, and File Storage.

CASM_GUI is specified as an ASM machine.

\begin{verbatim}
asm CASM_GUI
\end{verbatim}

The other three actors are the environment for CASM_GUI.

\begin{verbatim}
vocabulary :

domain CoreASM-Engine //The actor CASM_Engine
domain CoreASM-User   //The actor CASM_User
domain CoreASM-FileStorage //The actor FileStorage
\end{verbatim}
B.2 Components of CASM.GUI

CASM.GUI consists of nine components, output view, message view, program view, run view, vocab view, state view, updateset view, history manager, and control panel.

Component: output view
The tasks of the output view:

- Display the printout from the CoreASM machine executing;
- Clear the display in the view.

\[
\text{rule OutputViewProgram} \\
\text{choose aSignal from outputvSignalPool} \\
\text{remove aSignal from outputvSignalPool} \\
\text{case firstOf(aSignal) of} \\
\quad \text{outputvDisp } \rightarrow \\
\quad \text{displayMsginOutputV(secondOf(aSignal))} \\
\quad \text{outputvClearDisplay } \rightarrow \\
\quad \text{clearDisplayinOutputV}
\]

Component: message view
The tasks of the message view:

- Display any warning/error messages produced during the execution of the CoreASM machine;
- Clear the display in the view.
Component: program view

The tasks of the program view:

- Display the code of the CoreASM machine;
- Clear the display in the view.

Component: run view

The tasks of the run view:

- Add a new state icon in the view;
- Delete icons in the view;
- Highlight an icon in the view;
- Clear the display in the view.

```
rule RunViewProgram
  choose aSignal from runvSignalPool
  remove aSignal from runvSignalPool
  case firstOf(aSignal) of
    runvcAddAStep ->
      addRunViewElementinRunV(secondOf(aSignal))
    runvcDelSteps ->
      delRunViewElementsinRunV(secondOf(aSignal))
    runvcHighlightAStep ->
      highlightRunViewElementinRunV(secondOf(aSignal))
    runvcClearDisplay ->
      clearDisplayinRunV

Component: vocab view
The tasks of the vocab view:

- Display the vocabulary of the CoreASM machine;
- Clear the display in the view.
Component: state view
The tasks of the state view:

- Display a state of the CoreASM machine;
- Clear the display in the view.

Component: update set view
The tasks of the update set view:

- Display an update set produced during the execution of the CoreASM machine;
APPENDIX B. ABSTRACT MODEL OF CASM_GUI

- Clear the display in the view.

```
rule UpdatesetViewProgram
  choose aSignal from updatesetvSignalPool
  remove aSignal from updatesetvSignalPool
  case firstOf(aSignal) of
    updatesetvDisp →
      displaySetinUpdatesetV(secondOf(aSignal))
    updatesetvClearDisplay →
      clearDisplayinUpdatesetV
```

Component: history manager
The tasks of the history manager:

- Add a history record into the history manager;
- Delete history records from the history manager;
- Clear all history records in the history manager.

```
rule HistoryManagerProgram
  choose aSignal from historymanagerSignalPool
  remove aSignal from historymanagerSignalPool
  case firstOf(aSignal) of
    historymanagercAddAStep →
      addARuninHistoryManager(secondOf(aSignal))
    historymanagercDelSteps →
      delRunsinHistoryManager(secondOf(aSignal))
    historymanagercClear →
      clearinHistoryManager
```
Component: control panel
The tasks of the control panel:

- Activate an activity;

```
rule ControlPanelProgram

case nextUserActivityRequest( ) of

  userStartup →
   if guiSAMode_StartupA = undef then StartupActivity

  userLoadFile →
   if not isSynchronousActivityRunning( ) then LoadFileActivity

  userCheckSpec →
   if not isSynchronousActivityRunning( ) then CheckSpecActivity

  userInit →
   if not isSynchronousActivityRunning( ) then InitActivity

  userForwardRun →
   if not isSynchronousActivityRunning( ) then ForwardRunActivity

  userRollback →
   if not isSynchronousActivityRunning( ) then RollbackActivity

  userStop →
   if not isSynchronousActivityRunning( ) then StopActivity

  userInterrupt →
   if guiSAMode_InterruptA = undef then InterruptActivity
```
APPENDIX B. ABSTRACT MODEL OF CASM_GUI

userViewProgramCode →
   if guiSAMode_ViewProgramCodeA = undef then
      ViewProgramCodeActivity
userViewLastOutput →
   if guiSAMode_ViewLastOutputA = undef then
      ViewLastOutputActivity
userViewOutputInHistory →
   if guiSAMode_ViewOutputInHistoryA = undef then
      ViewOutputInHistoryActivity
userViewLastMsg →
   if guiSAMode_ViewLastMsgA = undef then ViewLastMsgActivity
userViewMsgInHistory →
   if guiSAMode_ViewMsgInHistoryA = undef then ViewMsgInHistoryActivity
userViewLastUpdateset →
   if guiSAMode_ViewLastUpdatesetA = undef then ViewLastUpdatesetActivity
userViewLastState →
   if guiSAMode_ViewLastStateA = undef then ViewLastStateActivity
userViewUpdatesetInHistory →
   if guiSAMode_ViewUpdatesetInHistoryA = undef then ViewUpdatesetInHistoryActivity
userViewStateInHistory →
   if guiSAMode_ViewStateInHistoryA = undef then ViewStateInHistoryActivity
userAddWatch →
   if guiSAMode_AddWatchA = undef then AddWatchActivity
userDeleteWatch →
   if guiSAMode_DeleteWatchA = undef then DeleteWatchActivity
userQuit →
   if guiSAMode_QuitA = undef then QuitActivity
B.3 Main rule of CASM_GUI

The machine of the CASM_GUI model is specified as parallel executions of components.

\[
\text{main rule } \text{CoreASMGUIProgram} = \\
//\text{GUI component programs} \\
\text{Vocab ViewProgram} \\
\text{Output ViewProgram} \\
\text{Message ViewProgram} \\
\text{Program ViewProgram} \\
\text{Run ViewProgram} \\
\text{State ViewProgram} \\
\text{Updateset ViewProgram} \\
\text{History Manager Program} \\
\text{Control Panel Program}
\]
B.4 Signals in CASM_GUI

These signals are the activity requests from CASM_User to activate the activities of CASM_GUI.

```plaintext
vocabulary:

enum domain USER_ACTIVITYREQUEST = {userStartup,
                                          userLoadFile, userCheckSpec, userInit,
                                          userForwardRun, userRollback, userStop,
                                          userInterrupt,
                                          userViewProgramCode,
                                          userViewLastOutput, userViewOutputInHistory,
                                          userViewLastMsg, userViewMsgInHistory,
                                          userViewLastUpdateset, userViewUpdatesetInHistory,
                                          userViewState, userViewStateInHistory,
                                          userAddWatch, userDeleteWatch, userQuit}
```
The following signals are the signals that are transferred between internal components of CASM_GUI.

```plaintext
vocabulary :

  enum domain PROGRAMV_COMMAND = {
    programvcDisplayFile, programvcClearDisplay
  }

  enum domain VOCABV_COMMAND = {
    vocabvcDisp, vocabvcClearDisplay
  }

  enum domain MESSAGEV_COMMAND = {
    messagevcDisp, messagevcClearDisplay
  }

  enum domain OUTPUTV_COMMAND = {
    outputvcDisp, outputvcClearDisplay
  }

  enum domain STATEV_COMMAND = {
    statevcDisp, statevcClearDisplay
  }

  enum domain UPDATESETV_COMMAND = {
    updatesetvcDisp, updatesetvcClearDisplay
  }

  enum domain RUNV_COMMAND = {
    runvcAddAStep, runvcDelSteps, runvcHighlightAStep, runvcClearDisplay
  }

  enum domain HISTORYMANAGER_COMMAND = {
    historymanagercAddAStep, historymanagercDelSteps, historymanagercClear
  }
```
The sending process for signals is specified as a rule, \textit{signalingTo( aSignalPool : SignalPoolName, aSignal : GUI Signal)}.

\begin{center}
\begin{verbatim}
rule signalingTo(aSignalPool : SignalPoolName, aSignal : GUI Signal) =
case aSignalPool of
  programv ->
    add aSignal to programvSignalPool
  messagev ->
    add aSignal to messagevSignalPool
  outputv ->
    add aSignal to outputvSignalPool
  statev ->
    add aSignal to statevSignalPool
  updatesetv ->
    add aSignal to updatesetvSignalPool
  runv ->
    add aSignal to runvSignalPool
  historymanager ->
    add aSignal to historymanagerSignalPool
  vocabv ->
    add aSignal to vocabvSignalPool
  controlpanel ->
    add aSignal to controlpanelSignalPool
\end{verbatim}
\end{center}
APPENDIX B. ABSTRACT MODEL OF CASM_GUI

B.5 Interfaces in the GUI ASM model

There are three interfaces specified in the GUI ASM model. They are the GUI → User interface, the GUI → FileStorage interface, and the GUI → Engine interface.

B.5.1 The GUI → User interface

The GUI → User interface specified in the GUI ASM model consists of the commands to activate the activities and the operations of CASM_User on CASM.GUI.

```plaintext
rule getFileURIToLoad (aUser : CoreASM-User,
                        aFileStorage : CoreASM-FileStorage)
rule getNumForwardStepsRequested (aUser : CoreASM-User)
rule getStateIdBackTo (aUser : CoreASM-User)
rule getStateIdViewInHistory (aUser : CoreASM-User)
rule getUpdateSetIdViewInHistory (aUser : CoreASM-User)
rule createAWatch (aUser : CoreASM-User, aVocab : GUI-VOCAB)
rule selectAWatchToDel (aUser : CoreASM-User)
```

B.5.2 The GUI → FileStorage interface

The GUI → FileStorage interface specified in the GUI ASM model provides an I/O channel to read a file stored on File Storage.

```plaintext
rule loadFile (aFileStorage : CoreASM-FileStorage, aUri : FileURI)
```
APPENDIX B. ABSTRACT MODEL OF CASM_GUI

B.5.3 The GUI ↔ Engine interface

The GUI ↔ Engine interface consists of a control interface and an access interface.

The control interface of GUI ↔ Engine interface:

\[
\begin{align*}
\text{rule} & \text{ checkSpecification} \ (aEngine : \text{CoreASM-Engine}, \ aFile : \text{File}) \\
\text{rule} & \text{ initSpecification} \ (aEngine : \text{CoreASM-Engine}) \\
\text{rule} & \text{ step} \ (aEngine : \text{CoreASM-Engine}, \ anI : \text{Integer}) \\
\text{rule} & \text{ rollback} \ (aEngine : \text{CoreASM-Engine}, \ anId : \text{StateID}) \\
\text{rule} & \text{ createEngine} \\
\text{rule} & \text{ killEngine} \ (aEngine : \text{CoreASM-Engine}) \\
\text{controlled} & \text{ interrupt} : \text{CoreASM-Engine} \rightarrow \text{Boolean} \\
\text{monitored} & \text{ getEngineMode} : \text{CoreASM-Engine} \rightarrow \text{ENGINE-MODE}
\end{align*}
\]

The access interface of GUI ↔ Engine interface:

\[
\begin{align*}
\text{monitored} & \text{ getGUI-VOCAB} : \text{CoreASM-Engine} \rightarrow \text{GUI-VOCAB} \\
\text{monitored} & \text{ getOutput} : \text{CoreASM-Engine} \rightarrow \text{GUI-Outputs} \\
\text{monitored} & \text{ getCurrentGUI-State} : \text{CoreASM-Engine} \rightarrow \text{GUI-States} \\
\text{monitored} & \text{ getLastUpdateSet} : \text{CoreASM-Engine} \rightarrow \text{GUI-UpdateSets} \\
\text{monitored} & \text{ getPreviousGUI-State} : \text{CoreASM-Engine} \\
& \quad * \text{GUI-Watch} * \text{StateID} \\
& \quad \rightarrow \text{GUI-States} \\
\text{monitored} & \text{ getPreviousUpdateSet} : \text{CoreASM-Engine} \\
& \quad * \text{GUI-Watch} * \text{UpdateSetID} \\
& \quad \rightarrow \text{GUI-UpdateSets} \\
\text{monitored} & \text{ getSpec} : \text{CoreASM-Engine} \rightarrow \text{File}
\end{align*}
\]
APPENDIX B. ABSTRACT MODEL OF CASM_GUI

B.6 Activities of CASM_GUI

The activities in CASM_GUI are specified as ASM rules. And they are classified into two groups, synchronous activity and asynchronous activity.

B.6.1 Activities

The followings are the activities specified at the second abstraction level of the GUI ASM model. The details about the signals transferring among the internal components and the usage of the interfaces in each sub-activity are specified formally at the third and the fourth abstraction levels. The full version of the four abstraction levels of the GUI ASM model will be provided as required.

StartupActivity

| rule StartupActivity | createGUIA seq createLinkEngineSA |

LoadFileActivity

| rule LoadFileActivity | getFileURIToLoadSA seq loadFileSA |

CheckSpecActivity

| rule CheckSpecActivity | checkSpecSA(currentEngine, guiAFile) seq getCSFeedbackSA |
APPENDIX B. ABSTRACT MODEL OF CASM_GUI

InitActivity

rule InitActivity
initSpecSA(currentEngine, guiAFile)
seq getISFeedbackSA

ForwardRunActivity

rule ForwardRunActivity
getNumForwardStepsRequestedSA seq forwardRunSA seq getFRFeedbackSA

RollbackActivity

rule RollbackActivity
getStateIdBackToSA seq rollbackSA

StopActivity

rule StopActivity

InterruptActivity

rule InterruptActivity
interrupt(currentEngine) := true


**ViewProgramCodeActivity**

```
rule ViewProgramCodeActivity
```

**ViewLastOutputActivity**

```
rule ViewLastOutputActivity
```

**ViewOutputInHistoryActivity**

```
rule ViewOutputInHistoryActivity
```

**ViewLastMsgActivity**

```
rule ViewLastMsgActivity
```
APPENDIX B. ABSTRACT MODEL OF CASM.GUI

ViewMsgInHistoryActivity

\[ \text{rule } \text{ViewMsgInHistoryActivity} \]

ViewLastUpdatesetActivity

\[ \text{rule } \text{ViewLastUpdatesetActivity} \]

ViewLastStateActivity

\[ \text{rule } \text{ViewLastStateActivity} \]

ViewUpdatesetInHistoryActivity

\[ \text{rule } \text{ViewUpdatesetInHistoryActivity} \]

\[ \text{getUpdateSetIdViewInHistorySA} \]

\[ \text{seq} \]

\[ \text{viewUpdatesetInHistorySA(guiUpdateSetIdToViewInHistory)} \]
APPENDIX B. ABSTRACT MODEL OF CASM.GUI

ViewStateInHistoryActivity

rule ViewStateInHistoryActivity
getStateIdViewInHistorySA
seq
viewStateInHistorySA(guiStateIdToViewInHistory)

AddWatchActivity

rule AddWatchActivity
createAWatch(currentUser, guiVocab)

DeleteWatchActivity

rule DeleteWatchActivity
selectAWatchToDel(currentUser)

QuitActivity

rule QuitActivity
killEngineSA(currentEngine) seq killGUI SA
B.6.2 The function \textit{isSynchronousActivityRunning}\((\ )\)

The function is defined to check if a synchronous activity is currently processing in CASM\_GUI.

\begin{center}
\begin{verbatim}
isSynchronousActivityRunning\(\)
  if (guiSAMode_StarupA = undef)
    and (guiSAMode_LoadFileA = undef)
    and (guiSAMode_CheckSpecA = undef)
    and (guiSAMode_InitA = undef)
    and (guiSAMode_ForwardRunA = undef)
    and (guiSAMode_RollbackA = undef)
    and (guiSAMode_StopA = undef)
    and (guiSAMode_InterruptA = undef) then

    isSynchronousActivityRunning := true
\end{verbatim}
\end{center}
Appendix C

ATM executable model

This appendix presents the ATM executable model.

```
asm ATMmanager
vocabulary :
  enum domain Mode = {idle, processing}
```
definitions :
main rule CoreASMGUIProgram =
  if Idle and ActivationEvent then
    data := getCardData
code := getPinCode
amount := getAmount
mode := processing
if Processing and IsAuthenticated(data, code) then
  if IsValidTransaction(data, amount) then
    ReleaseCash(amount)
    UpdateAccountBalance(data, amount)
  else
    OutputCancellationNotification
    mode := idle
if Processing and (not (IsAuthenticated(data, code)) or CancellationEvent) then
  OutputCancellationNotification
  mode := idle
where
  rule Idle
    mode = idle
  rule Processing
    mode = processing
  rule ActivationEvent
    isActivated = true
  rule CancellationEvent
    isCancelled = true
  rule IsAuthenticated(data, code)
    isAuthenticated(data, code) = true
where

**rule** IsValidTransaction(data, amount)

isValidTransaction(data, amount) = true

**rule** ReleaseCash(amount)

print "Releaseshash."

**rule** UpdateAccountBalance(data, amount)

print "Updateaccountbalance."

**rule** OutputCancellationNotification

print "Theoperationiscancelled."

static isAuthenticated(data, code) := true

static isValidTransaction(data, amount) := true

getCardData := "2345673242"

getPinCode := "3452011"

getAmount := 400.00

data : String
code : String

amount : Double
mode : Mode

isActivated : Boolean

isCancelled : Boolean
Bibliography


BIBLIOGRAPHY


