INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Application of Use Case Maps to System Design With Tool Support

by

Andrew Miga, B. A. Sc. (E.Eng.)

A Thesis Submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
July 1998

©1998, Andrew Miga

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-36893-9

The undersigned recommend to the Faculty of Graduate
Studies and Research acceptance of the thesis

**Application of Use Case Maps to
System Design With Tool Support**

submitted by Andrew Miga, B. A.Sc. (E.Eng.)
in partial fulfillment of the requirements for
the degree of Master of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University

July 1998

Abstract

The specification of complex software systems is one that often overwhelms designers with low level detail and does not provide a high level system view. Numerous software engineering methodologies exist for describing designs at the level of inter-object messages, data, and detailed class relationships, all of which is necessary for the detailed design stage but does not describe a high level system picture at an early stage of design. This thesis describes the design, implementation and applicability of the UCM Navigator, a prototype graphical editor to support the Use Case Map methodology, a methodology aimed at providing a high level system view where implementation details can be deferred. The UCM Navigator provides support for complex multilevel designs through support for the concept of stubbing in UCMs where symbols along a causal flow path may refer to separate submaps, to any level of complexity. The UCM Navigator also generates a linear textual form of entered designs, allowing it to be used as a front end for other software engineering tools as well as provides extensions to describe the execution characteristics of systems for its use as a front end for performance prediction simulations.

Table of Contents

Glossary of UCM Symbols	1
1. Introduction	4
1.1 Background and Motivation	4
1.2 Objectives	6
1.3 Overview of the thesis work	7
1.4 Contributions	8
1.5 Thesis outline	9
2. Background	11
2.1 Introduction	11
2.2 Description of Use Case Maps	11
2.3 Previous Work done on UCM Editors	11
2.3.1 Description of Use of Hypergraphs to Represent Use Case Maps	11
2.3.2 Work Performed by Undergraduate Students	13
2.3.3 Prototype C++ UCM Editor	14
2.4 Hypergraph Data Structures	15
2.5 The Representation of Use Case Maps with Hypergraphs	16
3. Use Case Map Navigator	20
3.1 Introduction	20
3.2 Requirements for a Use Case Map Editor	20
3.2.1 Basic Requirements	20
3.2.2 Requirements to support multilevel maps	21
3.2.3 Requirements to support performance prediction	22
3.3 The Use Case Map Navigator	22
3.3.1 Capabilities to support multi-level maps	27
3.3.2 Capabilities to Support Performance Prediction	28
4. Design and Implementation of the UCM Navigator	31

4.1	Introduction	31
4.2	Determination of Desired Characteristics for a UCM Editor	31
4.2.1	Deficiencies of UCM editor prototype	32
4.2.2	Summary of Desired Characteristics	33
4.3	Relation of Graphical Objects to Underlying Hypergraphs	33
4.4	Design Issues	35
4.5	Performance Issues	36
4.6	Achieving Separation of Concerns	40
4.7	UCM Navigator Classes	41
4.8	Description of Classes in the UCM Navigator	45
4.9	Use Case Maps of UCM Navigator Operation	45
5.	Generation of UCM Linear Form	52
5.1	Introduction	52
5.2	UCM Linear Form Structure	52
5.3	Example of Linear Form Generation	53
5.3.1	UCM Linear Form Generated From Example Map	59
5.4	The LinearGenerator Linear Form Generation object	62
5.5	Generation of Component Structure Specification	62
5.6	Generation of Path Structure Specification	63
5.6.1	Hypergraph Traversal	63
5.6.2	Generation of Path Atoms	64
5.6.3	Generation of Semi-Paths	64
5.6.4	Generation of Paths	65
5.6.5	Generation of Path Composition Specification	66
5.7	Summary of Linear Form Generation Algorithm	66
6.	Extensions for Performance Prediction	68
6.1	Motivation for using UCMs for Performance Prediction	68
6.2	Comparison - UCMs and Activity Sequence Networks	68
6.2.1	Differences in Path Interpretation between UCM and ASN	71
6.2.2	Differences in use of data between UCM and ASN	71
6.3	Extensions necessary to UCMs for Performance Prediction	72
6.3.1	Specification of choices between alternate scenarios	74

6.3.2	Verification of Response Times	74
6.3.3	Specification of Input Events	75
6.3.4	Specification of Responsibility Characteristics	75
6.4	Extensions to UCM Navigator for Performance Prediction	76
6.4.1	Timestamps	77
6.4.2	Response Time Requirements	77
6.4.3	Arrival Processes for Scenarios	77
6.4.4	Specification of Choices between Scenarios	78
6.4.5	Specification of Resource Functions for Responsibilities	78
6.4.6	Mutual Exclusion of Data Stores	78
6.4.7	Specification of Hardware Devices	78
6.5	Linking of UCM Navigator to Simulation Environment	79
6.6	Conclusion	79
7.	Case Studies	80
7.1	Introduction	80
7.2	Case Study A - Agent Project	80
7.3	Transformation of UCM Design into Agent Systems	87
7.3.1	Agent Models	88
7.3.2	Agent Internal Model	89
7.3.3	Conversational Model	91
7.4	Summary of Agent Case Study	93
7.5	Case Study B - Performance Prediction Project	94
7.5.1	Specification of performance annotations for root map	95
7.5.2	Specification of performance annotations for plug-in map	101
7.5.3	Global Specifications to UCM design	104
7.6	Process of System Simulation from UCM designs	108
7.6.1	Simulation Results	109
7.7	Summary of Performance Prediction Case Study	111
8.	Discussion	113
8.1	Introduction	113
8.2	User Interface Issues	113
8.2.1	Addition of UCM Notation and Desired Features	113
8.2.2	Advanced Editing Features	115
8.2.3	Manipulation of Display Properties	115

8.2.4	Enhanced visibility of transformations	116
8.3	Design Scalability Issues	116
8.3.1	Fisheye Viewing of Multilevel Maps	117
8.3.2	Multiple Root Maps and Selective Viewing of Paths	119
8.3.3	Substitution of Behaviour Patterns from Standard Libraries	121
8.3.4	Support for layering	121
8.4	Performance Issues	122
8.5	Integrating the UCM Navigator With Other Methodologies	124
8.6	Integration of the UCM Navigator With Other Applications	125
8.6.1	Transformation of Linear Form into UCM File Format	125
8.6.2	Display of Feedback from Performance Simulations	126
8.6.3	Transformation of UCM designs by external applications	128
8.6.4	Visualization of Scenarios	128

9. Conclusions and Future Work 130

9.1	Conclusions	130
9.1.1	UCM Editor Issues	130
9.1.2	Extensions for Performance Prediction	131
9.2	Thesis Contributions	132
9.3	Future Work	133
9.3.1	Advanced Editing and Display Properties	133
9.3.2	Design Scalability Support	133
9.3.3	Performance Improvements	135
9.3.4	Integration with other methodologies and applications	135

References 137

Appendix - Class Descriptions and Interfaces 139

Hyperedge subclasses	139
Class Hyperedge - Abstract	139
Class MultipathEdge - Abstract	142
Class WaitingPlace - Abstract	142
Class Connect - Concrete	144
Class Empty - Concrete	144
Class OrFork - Concrete	148
Class OrJoin - Concrete	150
Class Responsibility - Concrete	151

Class Result - Concrete	152
Class Start - Concrete	154
Class Stub - Concrete	156
Class Synchronization - Concrete	159
Class Timer - Concrete	162
Class Timestamp - Concrete	162
Class Wait - Concrete	164
Figure subclasses	164
Class HyperedgeFigure - Abstract	166
Class OrFigure - Concrete	167
Class OrNullFigure - Concrete	168
Class PointFigure - Concrete	169
Class ResponsibilityFigure - Concrete	170
Class Dynamic Arrow*	171
Class StubFigure	172
Class SynchronizationFigure - Concrete	173
Class SynchNullFigure - Concrete	174
Class TimestampFigure - Concrete	175
Map Component classes	176
Class Map	176
Class Hypergraph	178
Class Node	179
Class Label	180
Class Component	181
Class Path	185
Manager Classes	187
Class ConnectionManager	187
Class TransformationManager	190
Class FileManager	191
Performance Prediction classes	192
Class DataStoreDirectory	192
Class DeviceDirectory	194
Class ResponseTime	196
Class ResponseTimeManager	197
Presentation classes	198

List of Figures

Figure 1:	Representation of a use case map with hyperedges	18
Figure 2:	Elements of new hypergraph section	18
Figure 3:	Resulting hypergraph after transformation	19
Figure 4:	The Use Case Map Navigator	24
Figure 5:	Elements of UCM Navigator Main Window	25
Figure 6:	Responsibility and Component Characteristics	26
Figure 7:	Selection and Binding of Submaps	28
Figure 8:	Extensions for Performance Prediction	30
Figure 9:	Example UCM and its visible objects	34
Figure 10:	Underlying Hypergraph of Example UCM	35
Figure 11:	Graphical Objects in UCM Navigator Display	38
Figure 12:	Relationships of Graphical Objects	39
Figure 13:	Class Hierarchy Diagram of UCM Navigator	42
Figure 14:	Class Relationship diagram(excluding inheritance)	43
Figure 15:	CCD of ConnectionManager	44
Figure 16:	Hyperedge CCD	44
Figure 17:	Select Stub Expansion UCM	47
Figure 18:	Draw Screen Stub Expansion UCM	48
Figure 19:	Validate Transformation Stub Expansion UCM	49
Figure 20:	And Fork Creation UCM	50
Figure 21:	Stub Expansion UCM	51
Figure 22:	Example UCM to illustrate linear form generation	54
Figure 23:	Data Structures Present in Example UCM	55
Figure 24:	Object Structure to Linear Form Transformation	56
Figure 25:	Comparison between UCM and ASN	70
Figure 26:	Differences between UCMs and ASNs	73
Figure 27:	Root Map - Telephony Feature Interaction Example	83
Figure 28:	Originating - Default Plug-in for CSP	83
Figure 29:	OCS: Originating Call Screening for CSP	84
Figure 30:	Terminating: Default Plug-in for ASP	85
Figure 31:	CF: Call Forwarding for ASP	86

Application of Use Case Maps to System Design with Tool Support

Figure 32:	Agent Communication - Tuple Space (following [16])	87
Figure 33:	Mappings between UCM-agent concepts (following [16])	90
Figure 34:	Video Conference System UCM	96
Figure 35:	Resource Functions for A/V functions in Main Map	97
Figure 36:	Resource Functions for Keyboard Processing	98
Figure 37:	Specification of Arrival Event Time Distributions	99
Figure 38:	Specification of Branch Choices	100
Figure 39:	Specification of component to processor allocation	101
Figure 40:	Remote Stream Processing Stub	102
Figure 41:	Resource Functions and Layered Data Store Access	103
Figure 42:	TSVC Response Time Requirements	106
Figure 43:	Hardware Device Characteristics	107
Figure 44:	Layered Data Store Characteristics	108

Glossary of UCM Symbols

This glossary includes basic UCM symbols of the type that appear in this thesis. It is not complete relative to the full notation.

Path Elements

Path Start Point: Starting delimiter for a UCM path which signifies the starting of a scenario when appropriate stimulus is received

Path End Point: Terminating delimiter for a UCM path which signifies the end of a scenario

And Fork: Indicates that a single path is split into many concurrent forks

And Join: Indicates that numerous concurrent paths synchronize into a single path

Responsibility: Specifies an action to be performed by the system at that point of the path. May be bound to a software component

Stub: An element of decomposition in UCMs where a submap may be defined. Solid outline indicates a **static** stub with a single submap while dashed outlines indicate **dynamic** stubs with many submaps

Waiting Place: Specifies a synchronization point for a scenario where a scenario pauses until a triggering event is received

Timed Waiting Place: A Waiting Place which may have a timeout period defined at which point an exception action is taken

Dynamic Arrows: Small arrows pointing to or from paths indicate software components moving into or out of paths. May specify component, creation, destruction or copying

Move-Stay Arrow: An arrow with a perpendicular bar signifies the copying of a software component and its movement into or out of a path.

Creation, Destruction Arrows: Arrows with positive or negative signs correspond to component creation and destruction respectively

Scenario labelling specifying the end-to-end paths for scenarios labelled A and B. The section before the fork is labelled A+B (A OR B) to signify that it is a common (shared) segment, with no concurrency implications, for both scenarios A and B while the sections after the fork are only part of their respective scenario.

Scenario labelling specifying the end-to-end paths for scenarios labelled A and B. The section before the and fork is labelled A.B (A AND B) to signify that it is a common (shared) segment of both scenarios A and B while the sections after the fork are desynchronized after the fork and independently follow their own respective scenario.

Software Components

Team: A generic component which may be of any type and structurally contain any other component

Object: A passive low level component which may not contain other components

Process: An active component which has its own thread of control. May contain passive objects.

Pool: A storage area for operationally inactive dynamic components. Content of pools must move into slots to become visible and active

Agent: Software component similar to a team representing a software agent

ISR: Active object representing an interrupt service routine

A **stack** of components indicates a set of operationally identical but separate components

A **slot** of a particular component type indicates a placeholder for a dynamic component and is indicated by a dashed outline

1. Introduction

The specification of large, complex software systems often overwhelms designers with low level detail at the level of inter-object messages [1], low-level data, and detailed class relationships [2] [3] but often does not provide a high level view at which the behavior of a system may be understood. The Use Case Map methodology was developed to fill a void in the specification of software systems in that it aims to provide a high level view of the behavior of a system. This thesis describes the design, implementation, and applicability of the UCM Navigator, a prototype tool which supports the UCM methodology and provides support for multilevel designs, for descriptions of the performance characteristics of systems for its use as a front end for performance simulations and for the generation of a linear textual representation of entered designs allowing the UCM methodology to be used as a front end for other software engineering tools.

1.1 Background and Motivation

The understanding of complex software systems is not easily achieved as traditional software system descriptions focus on low level details from which end to end behavior signatures must be pieced together. These details are often of the level of inter-object messages, explicit class interfaces and hierarchies and low-level data which are often only well understood by the implementors of the code themselves. As systems grow to complexities beyond which are implementable by small groups of developers, often into the millions of lines of code, there is often no clear specification of the end to end behavior of complex systems. Industry often relies on experts who understand the complex systems as there is no clear way to describe system operation to new employees. The end result is complex software systems which are complex to understand, and therefore difficult to modify and extend. This results in increased development costs and greater unreliability of software as systems which cannot be understood cannot be modified without introducing new bugs.

There has been little work done in the high level visualization of systems as most methodologies either provide far too much detail or ignore visualization aspects entirely. Many methodologies exist which provide detailed component intercommunication sequences, such as message sequence charts [1]. These provide far too much low level

Application of Use Case Maps to System Design with Tool Support

detail to give a high level view. Other methodologies exist which provide textual descriptions, at the requirements engineering stage, of desired scenario execution such as Jacobson's work in use cases [4]. These methodologies provide a description of observable end-to-end behaviour but do not describe how the actions are performed such as sequences of communication events between the system components. Many methodologies exist to describe the object structures and inter-object behaviour of a design in terms of messages, interfaces and state machines [2] [3]. While these descriptions are very much necessary in the implementation of a software design they are often far too detailed to provide a system level view.

The Use Case Map methodology is based on the concept of describing end to end scenarios through a system as a causal flow of events and actions through an underlying component substrate. Details of inter-component communication are considered lower level detail which can be determined at a later stage. The UCM methodology allows end to end causal scenarios to be described at any level of abstraction allowing the behavior of complex systems to be described with ease. Support for design scalability is built into the UCM methodology in two forms, support for the concept of stubbing which allows path segments to be expanded as separate diagrams and support for the concept of layering where a Use Case Map of a certain layer describes certain behavior and considers other operations to be lower level detail, to be described by a UCM of a lower layer. A complex system described properly with a set of UCMs allows for much greater understanding by all designers and implementors resulting in increased efficiency for a software operation.

The difficulty with the use of the UCM methodology has been the lack of tool support. Tool support is desired for two main reasons, the increased ease of manipulation of UCM designs and the desire to use UCM design specifications as input to other CASE tools as the UCM methodology was never meant to be a replacement for traditional methodologies but rather a supplement. Proper tool support for the UCM methodology would provide a platform for designers to study the issues in the use of UCMs in the description of large systems. It would also allow the exploration of other uses of the UCM methodology such as the use of UCMs to describe the performance models of systems.

In addition the UCM methodology with its high level descriptions of behavior provides an excellent description for complex systems such as agent systems where inter-agent behavior is difficult to visualize. The development of a UCM editor would allow exploration of the use of UCMs to describe agent systems where the behavior of systems

changes dynamically as most current methodologies don't allow for the specification of behavioral dynamics in quite the same manner. Certain methodologies [2] allow for the specification of structural dynamics among objects involving component creation but do not address the description of systems with substitutable components or of end-to-end behaviour signatures across substitutable components. The ROOM methodology [5] with its concept of imported actors allows dynamic restructuring of systems, as is capable of describing behaviour, but it is often low level detailed behaviour which does not provide high level visualization of scenarios.

The major effort in this study is the development of a prototype UCM editor which would provide the ability to edit and easily manipulate UCMs in an environment with syntax checking as well as provide a platform for the use of the methodology as a design specification medium for further processing and specification. Although it provides a general tool for the use of UCMs in the specification of software systems, the work was motivated by two main projects, the use of the UCM methodology in the description of agent systems for Mitel Corporation and the use of Use Case Maps in the description of the performance behavior of systems for the purpose of performance prediction simulations from UCM designs, a Nortel sponsored project.

1.2 Objectives

The primary purpose of this study was to demonstrate the system design capabilities and high level visualization of software systems of Use Case Maps through the development of a graphical editor tailored to the UCM methodology. Specific sub-goals in this study are:

- To represent system structures and behaviors at high levels of abstraction through the use of complex multilevel maps
- To generate from graphical designs entered in the tool a linear text output, constructed from a context free grammar, which can be used as input to other tools
- To experiment with the use of Use Case Maps as a front end for performance simulation of systems through the addition of performance parameters to UCMs
- To develop a working tool as a vehicle to explore the application of Use Case Maps to large software systems

1.3 Overview of the thesis work

The primary effort in this study was the building of a graphical editor for Use Case Maps. This would serve as a vehicle both to explore the description of software systems at a high level and for use as the initial stage of software design where further processing on designs could be performed such as performance predictions and LOTOS simulations to verify system design.

The tool described in this study was initially created by the rewriting of a prototype use case map editor written by another student [10]. In addition the data structures used to represent use case maps, called hypergraphs, were developed by yet another student [8]. The basic editing functions offered in the prototype were recreated with a more solid foundation.

The work in this thesis extended the basic editor in many directions, such as the ability to specify complex multilevel maps using stubbing, and to generate a linear form output. Much of this work was motivated by the Mitel Agent project which required tool support for the agent concepts of alternate plans which describe agent behavior which are executed depending on the state of certain conditions. The tool supports this concept through support for alternate plug-ins for stubs.

The generation of a UCM linear form is partially motivated by the desire to use the tool as a front end for agent systems design as if a way could be found for describing the structure of UCMs in a textual manner, as through a BNF, this output could then be used as input to many other tools such as Lotos verifiers, performance simulators, and agent specification tools. If the use of UCMs as a front end for these other applications could be proven very beneficial then it would give prove the usefulness of both a tool to edit and manipulate UCMs and the UCM methodology itself.

The work performed in this thesis consisted of the construction of a UCM editor which would explore the concepts both of the use of Use Case Maps for the high level design of systems and of the use of a tool for manipulating UCM's as a front end for many other software analysis tools. It was desired to construct a tool that would be an intelligent editor for UCMs in that it would guide users in generating valid UCMs and possibly complex systems of UCMs with many hierarchical submaps. Such a tool would explore issues in the use of UCMs as a behavioral description methodology for large systems. The work involved the construction of a graphical UCM editor which could be used to create and

Application of Use Case Maps to System Design with Tool Support

manipulate UCMs with valid syntax, to create hierarchical multilevel designs involving submaps to express complex behaviour in a higher level map and which could create a textual representation of the structure of the designs entered into the system enabling the tool to be used as a front end for other analysis tools. A more complete description of the capabilities of the constructed tool, the UCM Navigator, can be found in chapter 3.:Use Case Map Navigator.

The work began as a rewriting of the prototype written by Jeromy Carriere in C++. It was decided that beginning from an implementation that had basic functionality implemented and which had an accoladed user interface would save time over an implementation from scratch. The tool was rewritten to use internal data structures to store UCMs in memory based on hypergraphs in order to leverage the previous work described in the preceding sections.

Once the goal of producing a stable, usable UCM editor capable of producing flat maps was achieved numerous enhancements were performed. The first was the addition of the capability to expand the description of the plug-ins of stubs as separate maps. This allowed the creation of multilevel maps. The second was the generation of the textual linear form for UCMs that was defined by the team of UCM researchers at Carleton. The third was the series of extensions to capture information needed for performance prediction using UCMs to describe the systems.

The product of this work was the UCM Navigator, a graphical editor for the UCM methodology with multilevel map and performance prediction capabilities. It is written in C++ for UNIX systems and uses the XForms GUI library and the X-Windows system for its graphical display. It comprises approximately 25500 lines of code with about 20000+ lines being written by the author. The remaining code was from the C++ prototype (~3000-3500 lines) and code that was translated from Smalltalk from the failed undergraduate implementation (~1000-1500 lines).

1.4 Contributions

The UCM Navigator tool described in this thesis is a working system which supports the Use Case Map methodology and as such is an early work in which issues such as high level design of systems using UCM's can be explored.

The ability of the UCM Navigator to generate a (previously defined) linear form

Application of Use Case Maps to System Design with Tool Support

output permits it to be used as a graphical interface for exploring other aspects of the use of UCM's for system design such as design validation with other tools. The linear form itself had already existed, only the generator has been implemented.

The extensions to UCM Navigator for performance prediction allow the tool to be used as a graphical interface to various simulation engines which is early work in the field of attaching performance information to the UCM design methodology.

The UCM Navigator provides explicit support for the definition of dynamically changing behavior signatures through the use of alternative plug-ins for stubs. This was motivated by the Mitel Agent project.

The case studies presented on the application of UCMs to agent prototyping and performance prediction constitute descriptions of applications of the UCM methodology.

1.5 Thesis outline

Chapter 2 provides background for this thesis. Previous work done on graphical editors for UCM's is described as are their strengths and weaknesses. In addition the data structures (which are based on previous work) used to store and manipulate UCM's in the tool are described.

Chapter 3 describes the requirements for a UCM tool and introduces the Use Case Map Navigator tool. Its major functionality and capabilities are described.

Chapter 4 is devoted to the design of the tool as well as the implementation issues that arose in the development of UCM Navigator. The description of the design also provides an example of how software systems can be specified with the UCM Navigator. Design decisions that affect the performance of the tool and its user interface are explained in detail.

Chapter 5 describes the generation of UCM linear form output from the data structures created during map drawing. The design of the translation process and the objects involved are described in detail.

Chapter 6 focuses on the extensions to the UCM Navigator that are necessary for

Application of Use Case Maps to System Design with Tool Support

performance evaluation of software. The added capabilities are described and an example application is given.

Chapter 7 gives two small case studies of applications designed using the UCM methodology. The chosen applications are the Mitel agent project and the Nortel performance prediction project.

Chapter 8 provides a discussion of the major contributions of the thesis and of the issues raised.

Chapter 9 summarizes the contributions of the thesis and provides directions where this work can be extended.

2. Background

2.1 Introduction

This section will describe background material needed for the understanding of this thesis. This includes previous work done in the development of graphical editors for UCMs, a description of the data structures used to represent the use case maps internally as well as how they are used to allow users of the tool to manipulate use case maps. In addition related work in software engineering in the area of scenario description languages/notations as well as tools will be described.

2.2 Description of Use Case Maps

Use Case Maps are a graphical path-based notation used for specifying causal sequences of actions through systems. It is not the purpose of this chapter or this thesis to describe UCM concepts. A complete description of the Use Case Map notation can be found in Buhr [6] [7] . The graphical symbols used in the Use Case Maps in this thesis are described in the previously displayed glossary.

2.3 Previous Work done on UCM Editors

There have been three previous attempts at defining and/or implementing a UCM editor. The first was an uncompleted thesis on the use of hypergraphs to represent UCMs, the second was an attempt by undergraduate students to use the hypergraph representation to build a UCM editor, and the third was a working but incomplete prototype of a UCM editor written by a graduate student as a course project.

2.3.1 Description of Use of Hypergraphs to Represent Use Case Maps

The first attempt at describing the workings of a UCM editor was a partial thesis done by a previous student which described how hypergraphs, a graph notation using edges as the elements of the domain and nodes between the edges as the connection mechanisms, could be used to describe the paths of use case maps[8]. These hypergraph data structures would be the underlying data structures for a use case map editor based on the

Application of Use Case Maps to System Design with Tool Support

concept of hypergraph transformations in which Use Case Maps or at least use case paths would be drawn by a series of transformations requested by the user. Hypergraph structures consisting of hyperedges representing the UCM elements (start point, responsibilities, and/or forks/joins, stubs) and nodes as the connection mechanisms between the hyperedges would describe valid UCMs in memory. Elements would be added to or removed from the path by transformations which replaced sections of the hypergraph (consisting of chains of hyperedges and nodes) with new sections. In this manner the map being drawn can be assured to be valid. The hyperedges would be programmed with enough intelligence to know their functions and the transformations in which they can be part. This combination of underlying easily manipulated graph grammar data structures and programmed intelligence in the graph elements could be used to build a UCM editor that edited UCMs as UCMs and not as a collection of lines and boxes. Maps drawn would always be valid and such an editor could easily be programmed to disallow invalid operations.

However the work performed was incomplete. Many of the ideas in the thesis were not fully developed and not enough concern was given as to how the maps would be displayed and manipulated on the screen of a graphical editor. This resulted in some of the ideas being too strict to implement an editor with a pleasing user interface. There was no implementation performed in this attempt which might have explored the graphical issues. In addition there was no support for software components, only for use case paths meaning that any support for both the drawing and graphical manipulation of components and more importantly the binding of path elements (namely responsibilities) to these components would have to be added on top of the hypergraph structure. This is not a major flaw in the work, only a comment on the incompleteness of the partial thesis as a description for a possible UCM editor. As it is generally desired to separate the concerns of graphical display and management from the underlying data structures (as will be explained in chapter 4) this partial thesis provides a good starting point for how hypergraphs can be used to represent the internal structures of use case paths. However the entire design of a UCM editor is incomplete from this document.

What was used from this document was the basic underlying concept of the use of hypergraphs to represent UCMs and the concept of editing maps as a series of hypergraph transformations. As this effort involved no implementation there was of course no code of this effort used in the UCM Navigator tool.

2.3.2 Work Performed by Undergraduate Students

This second attempt at building a UCM editor was a fourth year project of two undergraduate students[9]. They attempted to use the partial thesis describing the use of hypergraphs to describe UCMs as the basis for building a working editor in Smalltalk. The attempt failed as at the end of the project time the program simply did not work. What resulted was about 5000 lines of object oriented code of how an implementation could be made. While the code was incomplete and often incorrect it was substantial as a starting point for an implementation. Many transformations were coded and the code illustrated the application of the hypergraph concept to a working editor. The concept of validating and performing transformations was visible in the code

However the code was incomplete as it did not cover the graphical aspects of the editor well enough and did not contain support for drawing software components, the same two major shortcomings of the partial thesis. This implementation used HotDraw, a Smalltalk graphics framework to draw the paths. This proved to not have the functionality needed to display use case paths properly and so the graphical code was very incomplete while the code for hypergraph manipulations was much more structured and usable to some extent.

The major difference between this implementation attempt and the UCM Navigator as well as the C++ prototype written by Jeromy Carriere described in the next section, upon which the UCM Navigator's interface was based, was the user interface. This implementation was based on the concept of automatic placement of map elements which besides being difficult to implement and not being user-friendly did not work at all with the concept of software components being part of maps and having responsibilities bound to these components. This meant that the entire graphical part of the tool which was incomplete anyway could not be used even for ideas on how to implement a new version. It also meant that the transformations that existed in the code would have to be modified as one of the assumptions of the partial thesis would have to be changed. This assumption was there would be exactly one empty point (which corresponds to an empty hyperedge , which will be explained later) between any other UCM elements. A more friendly user interface would allow users to draw empty points wherever they pleased and later place map elements at these points. This would require that multiple consecutive empty points be possible on use case paths, something that the partial thesis and thus this implementa-

Application of Use Case Maps to System Design with Tool Support

tion disallowed. While adding the ability to have consecutive empty points along a path is simple enough it removes a basic assumption of the structure of hypergraphs on which the transformations in this partial implementation were coded.

Some of the hypergraph manipulation code was used in the UCM Navigator as were the definitions of some of the basic objects. This amounted to approximately 1000-1500 lines of code being reused (out of the total line count for the UCM Navigator of approximately 25000) although much of it was highly modified. There was little point in reusing most of the transformations as they involve quite simple hypergraph manipulations and needed to be rewritten anyway both as a result of the assumption about consecutive empty points mentioned above changing and the fact that as the entire graphical part of the tool would have to be written from scratch, the calls made from the transformations which tell the graphical part of the tool what to display on screen would have to be redone.

2.3.3 Prototype C++ UCM Editor

The third effort in the area of UCM tool development was a prototype UCM editor written in C++ for UNIX systems as a course project by Jeromy Carriere[10]. It had the most impact on this thesis as it was the first implementation that worked and that aroused interest in a tool for editing UCMs that could not only allow users to draw them but that could act as a front end to other tools. It is described in more detail in section 4.2. It had a very intuitive and pleasing interface with which users could simply draw paths by clicking where they wanted path nodes to appear. Path elements could be placed along these paths by selecting nodes and choosing selections from a popup menu. The tool also supported software components and provided bindings between path elements and the components. The tool also provided a list of responsibilities and their characteristics as well as post-script output.

However the tool was very unstable as it was written quickly without much analysis. There were no underlying data structures where the UCM structure was recorded. There were only graphical objects which had no intelligence programmed into them as to what operations they could and could not perform. This resulted in very many map editing operations causing the tool to crash as the data structures became disorganized. The tool was quite unusable however it had a very positive effect on those who had seen it as a result of its interface. This lead the research in the area of a stable extensible UCM editor

with the same type of user interface.

The tool in this thesis began as a rewrite of this prototype with completely new program internals. As this prototype had a good user interface, using the public domain GUI library and builder XForms, it would be a good starting point for the UCM Navigator. The interface could be built up incrementally as new elements and dialogs were needed. The need for new internal data structures was painfully clear from the operation of the prototype and the fact that the majority of the code was badly designed and completely unextensible.

Some of the code used in this prototype is still used in the UCM Navigator. It comprises mostly basic functions such as code for handling events in the main window, drawing area, and dialog boxes. Also the interface to the X-Window system and the ability to produce postscript files came from this prototype as did the code for drawing and manipulating software components. Approximately 3000-3500 lines of code from this prototype exist in the UCM Navigator although much of it has been modified by the author.

2.4 Hypergraph Data Structures

The data structures to store the UCM structures are based on a variation of a standard graph grammar called a hypergraph. Basically a hypergraph is a graph described by a collection of nodes and edges where there is a hypermedia principle whereby certain nodes can have the property of storing complete subgraphs such that if a hypergraph structure were being viewed and a node with a subgraph were selected the subgraph would be displayed.

There is no single correct interpretation of hypergraphs as there are many variations, some with additional characteristics. The standard interpretation of a hypergraph is of a collection of nodes which are connected together by hyperedges. In this interpretation nodes can have any number of input and output edges and edges may only connect two nodes. Variations exist however where the roles of edges and nodes are reversed. In one of these variations hyperedges may have any number of input and output nodes and nodes have the additional property of having "colors" in that a node's color would determine the type of node and by extension the type of graph to which the node is connected.

It is this interpretation with reversed edge and node semantics and node coloring that was used by the earlier UCM researchers to describe the structure of UCMs in mem-

ory[8]. It was chosen for two main reasons. First the concept of hypergraph links whereby a hyperedge could contain a link to a subgraph would be very useful in specifying sub-maps of UCM stubs allowing hypergraphs to be used to represent complex multilevel UCM designs. Secondly the concept of the coloring or typing of nodes could be used to differentiate the types of connections that exists between UCM elements. The two types of connections would be primary connections between the map elements along a path and triggering connections whereby a path element along one path triggers asynchronously a waiting place along a second path. This use of node coloring was used in the tool to differentiate the two types of inter-element connections.

For the purposes of the design presented in this thesis the hypergraph elements have the following definitions.

Hyperedge: an atomic item with an ordered set of input and output attachments

Node: atomic item which connects two hyperedges

2.5 The Representation of Use Case Maps with Hypergraphs

The UCM editor developed for this thesis uses hypergraphs for the internal data structures of the paths as the C++ prototype required new internal data structures and it was desired to leverage the previous work done in the area of hypergraph representation of UCMs, described in the earlier sections of this chapter. The use of hypergraphs, one of many graph notations, would provide needed stability to the tool as the structure of use case paths could be stored in simple but highly modifiable graph structures, providing the ability to manipulate maps in memory by simple, highly analyzable graph transformations. It would also provide the ability to program intelligence into the graph elements which represent UCM elements so that each map element could perform transformations on itself at user request which would transform the hypergraph representing the path structures at the point of the transforming element. This would also, as stated previously, enable each map element to determine which transformations it may perform and which are valid at the current time.

A Hypergraph is formed of a collection of Hyperedges and Nodes. Hyperedges are connected by means of source and target nodes to form hypergraphs. Each Use Case Map element (start point, responsibility, fork/join, stub) is mapped to a hyperedge. Nodes are simply objects which connect the hyperedges into chains representing paths. Transforma-

Application of Use Case Maps to System Design with Tool Support

tions on hypergraphs are based on the concept of hypergraph replacement, that is replacing one section of a hypergraph with another. The section being replaced is either that of a new map element being added to the path at the selected point or that of an empty section which replaces the section of an element which is being deleted from the path.

A map is drawn initially on screen as the simplest possible valid UCM, a path consisting of a start point followed by an empty point and end bar. This corresponds to a start hyperedge, and empty hyperedge, and a result hyperedge. The empty hyperedge does not correspond to a UCM element per se, it rather corresponds to a point along a use case path. Empty hyperedges corresponding to these points are used as placeholders in the hypergraph structure. That is it is at these points that transformations to add new elements are made. As empty hyperedges are the places at which new elements are added there is a requirement that every two map elements be separated by at least one empty point. The connections between the empty hyperedges and the preceding and following hyperedges are broken and a new section of hypergraph is inserted in its place. The old section is then deleted. Connections are made and broken by changing the source and target nodes to which hyperedges point and changing the hyperedges to which the nodes point. Figure 1 shows the hypergraph of a simple path consisting of a start point, a responsibility, and an end point. The dashed line representing the path is added only as a visual aid. The objects which draw the splines representing path segments are not part of the hypergraph structure but rather part of the graphical representation objects in the UCM Navigator.

Use case map transformations with hypergraphs are based on hypergraph replacement. For example, to add a timer after the responsibility in the previous path the following actions must be performed.

Figure 1: Representation of a use case map with hyperedges

The first action is to create a new hypergraph section(requires creating empty hyperedges as each pair of use case map elements must be separated by an empty hyperedge). Figure 2 shows the hypergraph section created to add a timer hyperedge to a use case path at a given point. It simply consists of a timer hyperedge placed between empty hyperedges which are connected by several nodes to form a path section. In practice the UCM Navigator only creates as many empty hyperedges, which correspond to empty points, as are needed to maintain the rule of at least one empty point between every two map elements. For example if the user of the tool added a map element at an empty point which was preceeded and followed by other empty points no new empty points would need to be created.

Figure 2: Elements of new hypergraph section

The second action is to install the new section by replacing the empty segment at

Figure 3: Resulting hypergraph after transformation

- 19 -

3. Use Case Map Navigator

3.1 Introduction

This chapter will describe the requirements for a Use Case Map editor in terms of the desired basic functionality as well as desired extensions. The chapter will continue with the introduction of the UCM Navigator and a description of its features as well as a discussion of how its features match the desired requirements. The operation of the editor will be illustrated through a series of screen captures.

3.2 Requirements for a Use Case Map Editor

This section will enumerate the desired requirements for a UCM editor in terms of the basic map editing functionality needed and the ability to create either static or dynamic stubs to provide support for multilevel maps and dynamically changing behaviour signatures. In addition the extensions that are needed for such an editor to be used as a front end for performance simulations, consisting of the addition of execution information normally not present in Use Case Maps, will be enumerated.

3.2.1 Basic Requirements

The primary function of a Use Case Map editor would be to draw edit, and manipulate Use Case Maps as UCMs and not as collections of lines and boxes which are connected in a certain way to appear as a UCM as one would draw them with a standard drawing package. The editor must restrict what can be drawn so that it is always a valid UCM. Such an editor would be based on the concept of transformations on use case maps. The concept of hypergraph transformations was described in chapter 2. Such an editor would allow users to create maps by initially placing an empty path where they desired and then add UCM elements to this path by selecting points along the path and performing whatever UCM transformations are applicable at that point, or points if multiple points are selected. Possible transformations include adding elements such as responsibilities, waiting points or timers, creating forks and joins (either Or or And) , splitting paths, joining paths and composing connections between paths.

Application of Use Case Maps to System Design with Tool Support

The editor must be able to draw and manipulate software components (processes, objects) , support nested components, as well as bind these components to the responsibilities that are contained inside them. This binding would also extend to all map elements which are inside the component boundaries. This would result in both the bindings being recorded in a textual description of the map structure and the ability to move and resize components with the path elements inside providing quick sketching capabilities. Additional basic requirements of a UCM editor are as follows:

- Support the description of the responsibilities in a use case map. This includes textual descriptions as well as graphical notations expressing the structural dynamics of software components
- Support the description of textual conditions and events at points along paths. These include pre/postconditions and triggering/resulting events which may exist at the start and end points of paths, at waiting places along paths, at stubs or simply along path segments.
- Support the generation of a textual EBNF linear form of the structure of maps which can be used as input to other tools such as agent specification tools, performance simulators, and LOTOS simulators
- Provide a means to generate graphical output such as postscript files (this capability existed from the prototype and was not a contribution of the author)

An additional design requirement for the editor is that it should separate the graphical manipulation and display from the underlying data structures so as to provide a modifiable, extendable code platform.

3.2.2 Requirements to support multilevel maps

This section will describe the requirements of a UCM editor to support the definition of submaps whose origins exist in higher level maps. This will allow the editor to create and manage multilevel maps.

The concept of decomposition is provided in Use Case Maps with stubs. Stubs appear along use case paths and represent either simply decomposed path segments which could be inserted into the path at the stub position or alternately submaps which may extend beyond the boundaries of the parent map. Submaps for stubs are termed plug-ins. A stub may be either static or dynamic. Static stubs have only a single plug-in while

Application of Use Case Maps to System Design with Tool Support

dynamic stubs may have many plug-ins which represent alternate behaviour signatures that are selected based on run time conditions. The requirements for a UCM editor to support multilevel maps are:

- Allow the creation of submaps for both static and dynamic stubs which may be viewed and edited as any other map
- Support the binding of the input and output points of a stub to the path start and end points in the submap. These bindings will be recorded in the linear form of the map structure so that end to end behaviour across levels can be documented
- Support the navigation of multilevel maps through stub expansions

3.2.3 Requirements to support performance prediction

This section will describe the requirements of a UCM editor to support its use as a front end for performance simulations. Chapter 6.:Extensions for Performance Prediction describes the necessary extensions in more detail although the basic requirements will be presented here.

To support the performance description of systems UCMs drawn with an editor need to be augmented with the information that a performance simulator would require and which are normally excluded from UCMs as lower level detail. This includes the following:

- Ability to specify timestamp points along paths and specify response time requirements between these points
- Ability to specify choices between scenarios at or forks in UCMs
- Ability to specify hardware resources used by responsibilities
- Ability to specify global data stores accesses by responsibilities
- Ability to specify time distributions of the arrival of triggering events for scenarios along paths
- Ability to specify the characteristics of the hardware resources used by a system

3.3 The Use Case Map Navigator

This section will describe the Use Case Map Navigator and its features. A description of the operation of the editor can be found in the user's manual [11]. A discussion of how they match the requirements specified in the previous section will follow.

Application of Use Case Maps to System Design with Tool Support

The following graphic Figure 4: The Use Case Map Navigator shows the main window of the UCM Navigator and Figure 5: Elements of UCM Navigator Main Window displays the same main window with the major areas of the editor highlighted. As can be seen the editor possesses a main editing area, tools to create paths, components, and select elements, and areas for viewing and editing the characteristics of both responsibilities and conditions/events. The window also contains a display for the decomposition level of the map shown in the editing area. The map shown is actually a plug-in for a stub in the root or top level map of this design. The UCM Navigator allows complex multilevel maps to be created and navigated.

The popup menu shown in figure 4 shows the possible UCM transformations for the selected element, in this case, an empty point. The UCM Navigator allows users to construct use case maps by performing transformations on existing maps assuring that all maps are always valid UCMs with valid syntax. Control over the positioning of elements, the shape of paths and the appearance of components is available to the user, however all changes to the structure are always managed by the underlying hypergraph model assuring the validity of the maps. This satisfies the major requirement of a UCM editor specified in the previous section. In addition the design which is presented in chapter 4 will show that there is a clear separation of concerns between graphical manipulation and display and internal data structures.

Figure 4: The Use Case Map Navigator

The transformations that are applicable depend on the map element or elements that were selected as they are the transformations that are possible for that element or pair of elements. Transformations on empty points as shown in figure 4 are significant as that is how new elements are added to existing maps. There is at least one empty point between every two map elements. Therefore new elements can be placed anywhere on the map. As the additions are performed by precoded transformations on the maps they are always valid. For example adding an and/or fork not only adds the fork but also the stub of the second output path so the fork has no dangling ends. Similarly deleting the second last path connected to a fork or join deletes the fork or join as it is now redundant. Cutting a path at an arbitrary point ensures that the proper starting and terminating UCM elements are added to the recently cut path ends so that there are no dangling ends. The UCM Navi-

Application of Use Case Maps to System Design with Tool Support

gator therefore allows users to draw any valid UCM and position its elements as they please but performs all map modifications itself in response to user choices so as to ensure the integrity of maps drawn with the tool.

Figure 5: Elements of UCM Navigator Main Window

. The editor is also capable of creating software components and binding all map elements to these components. The graphic Figure 6:Responsibility and Component Characteristics depicts the dialogs used to edit the characteristics of responsibilities as well as those of components. Responsibilities have the basic characteristics of name and description with optional notation expressing component structural dynamics.

Figure 6: Responsibility and Component Characteristics

For the performance prediction extensions the characteristics of execution sequence (where resource functions are described) and referenced data stores have been

added. Components may have various characteristics aside from type including replication, dynamic/fixed, anchored/non-anchored etc.

The UCM Navigator also supports the generation of a textual linear form as will be explained in detail in chapter 5. The ability to generate postscript output existed from the prototype and was not a contribution of the author. All of the use case maps shown in this thesis were drawn with the UCM Navigator and imported as Encapsulated PostScript files into this document.

3.3.1 Capabilities to support multi-level maps

The UCM Navigator supports the creation of multilevel maps through its support for the concept of stubbing in UCMs. A stub can be drawn in a map representing an abstraction of some part of a system. Then plug-in maps (either static or dynamic) for that stub can be drawn and linked with the stub definition in the top level map. A binding between the input and output points of the stub and those of the plug-in map can be made which is then recorded in the generated linear form of the design.

Figure 7:Selection and Binding of Submaps shows the process of selecting a sub-map for either display or for editing the bindings between the input and output points of the top level map and those of the plug-in. During the binding process the input and output branches of the stub are given temporary labels which appear in the binding dialog. Users can select pairs of inputs or outputs and have the bindings recorded in the stub's internal data structures which are exported to the linear form. Both the navigation of multilevel maps through stub expansions and the bindings of stubs to their plug-ins are requirements for a UCM editing tool that have been satisfied.

Figure 7: Selection and Binding of Submaps

3.3.2 Capabilities to Support Performance Prediction

The UCM Navigator has been extended to include the necessary information for performance simulations described previously. This information includes the specification of timestamp points and response time requirements between them, resource function characteristics of responsibilities, specification of choices at or forks, time distributions of scenario triggering events, specification of hardware devices, binding of components to hardware processors, and the specification of global data stores as well as the access to them by responsibilities.

Figure 8: Extensions for Performance Prediction shows some of the mechanisms by which the performance information can be added to models designed using the UCM Nav-

Application of Use Case Maps to System Design with Tool Support

igator. The main window shows a design using timestamp points along the use case paths. These represent markers at which the execution time will be recorded during an actual simulation. The UCM Navigator has the ability to specify response time requirements between any two timestamp points (that can be reached from each other) so performance simulations can determine whether the design meets the desired requirements. The window at the top right displays the collection of response time requirements for the design and provides means for editing them.

Access by responsibilities to global data stores may be specified in the window in the bottom right of figure 8. The layered data stores that are accessed by the responsibility as well as the access modes are specified in this window. This dialog is invoked by the Specify Data Stores button in the responsibility editing dialog shown in figure 6. The responsibility dialog also contains an area for specifying the resource function of a responsibility, needed for simulation purposes.

The specification of the time distribution for arrival of triggering events for scenarios along paths is specified with the dialog at the bottom left of figure 8. Various statistical processes are available.

The specification of hardware devices, data stores and scenario selection at Or Forks are performed in other dialogs not shown in this thesis. The specification of hardware processor for certain software components can be seen in the component characteristics dialog at the bottom left of figure 6. This can only be specified for processes and ISRs.

Figure 8: Extensions for Performance Prediction

4. Design and Implementation of the UCM Navigator

4.1 Introduction

This chapter will describe the design of the UCM Navigator. The major classes will be described and use case maps of common system scenarios will be given. As well the design issues that led to the given design will be discussed and the design choices described. As this project began as a rewriting of an unstable prototype with a desirable user interface the desired characteristics of the design will be determined by comparison with the prototype.

4.2 Determination of Desired Characteristics for a UCM Editor

This tool began as an extension to a prototype UCM editor written by Jeromy Carriere which explored many user interface issues in the design of a UCM editor. As this project began as a rewriting of an existing prototype rather than the creation of an editor from scratch it is worthwhile to analyze the prototype to determine its desirable features which should be duplicated in the new version as well as its undesirable properties which need to be eliminated.

The prototype UCM editor, called `ucmedit`, possessed many desirable user interface features but was unstable as it was implemented without a proper underlying foundation in terms of the data structures, or lack thereof, used to represent use case paths. The interface used to draw paths was an intuitive point and click interface where paths could be drawn simply by the user moving the mouse pointer to where they desired map points to exist and clicking. Software components could be drawn simply by dragging out rectangles and then changing component characteristics in dialog boxes. Use case path elements were automatically bound to software components by being drawn within their boundaries. This binding meant that the map elements would move with the component when it was moved or resized giving a quick sketching capability to the editor. All of these UI features were intuitive and very well received by all who were given demonstrations of `ucmedit`. All of these features are desirable and were included in the design of UCM Navigator. In addition `ucmedit` used spline curves to draw naturally rounded lines to represent the use case paths. This is also a very desirable property, which was kept in the new ver-

sion, although it possesses performance implications which will be described later.

4.2.1 Deficiencies of UCM editor prototype

The deficiencies of ucmedit were quite obvious, the main being the program's instability. This was mainly due to the fact that the only data structures present in the editor were graphical in nature, representing the path points and splines that were drawn on screen. Ucmedit used four primary types of objects points, representing path points, segments representing path segments, which are collections of points, paths, which are collections of segments and components representing software components. Operations on paths, such as adding responsibilities or forks/joins were performed by attempting to manipulate these objects to achieve the desired effect on screen. The editor had no knowledge of the meaning of a set of paths in use case map terms and as such would often perform operations on paths in inappropriate manners resulting in corrupted data structures and thus program crashes.

Another deficiency of ucmedit was its poor performance in dealing with large maps containing many paths and especially long paths. This was due to the fact that the use of spline curves to display path segments has performance implications as interpolating a spline given a set of points is a computationally intensive task. As ucmedit uses a spline to represent each path segment it only reinterpolates the spline whenever one of its constituent points has changed. When no point has changed position the spline can be redrawn using the previously calculated intermediate points.

One of the reasons for this poor performance with large maps was that a bug existed which caused a single spline to possibly be reinterpolated several times between screen redraws. Although this bug was easily fixed the poor performance that was previously visible with large maps reinforced the notion that any new version of the editor would need to minimize spline calculations.

Another deficiency of ucmedit was the tool's inability to handle scale properly in the manner that the UCM methodology handles large scale systems. With UCM's stubbing of path segments is used to hide details from a main map. Stubbing can be used recursively allowing for multilevel designs. Ucmedit had some support for stubbing but it was limited to the concept of collapsing path segments into stubs. There was no support for expanding stubs into separate maps meaning that the paths in the stub could not cross

Application of Use Case Maps to System Design with Tool Support

components outside the stub boundaries, which is allowed in the UCM methodology. Also without the expansion of stubs into separate maps the ability to support dynamic plug-ins for stubs does not exist.

4.2.2 Summary of Desired Characteristics

This section determined the desired characteristics for a UCM editor by comparison with the prototype. These characteristics are formed from both the positive characteristics of the prototype and from its deficiencies which would need to be corrected in a newer version. The positive characteristics of the prototype that should be preserved are as follows:

- Intuitive point and click interface
- Automatic binding of path elements to software components
- Use of spline curves to display use case paths

The deficiencies of the prototype that need to be corrected are:

- Instability of execution
- Poor performance in the drawing of large maps
- Lack of proper support for both static and especially dynamic stubs

In addition to these characteristics new capabilities are desirable for a UCM editor such as the ability to generate a textual BNF of a UCM design and its use in the area of performance prediction.

4.3 Relation of Graphical Objects to Underlying Hypergraphs

The following sections will describe the objects used in the design of the UCM Navigator and how the chosen design fits the desired principles for a UCM editor. This section will provide necessary background through graphical examples of how the objects in the design relate to either those that are visible on the screen or act as underlying data structures.

A simple UCM drawn with the UCM Navigator is shown in figure 9. The visible objects on screen are identified by object type by being labelled with class names. The Figures are all subclasses of the abstract Figure class which as will be explained manage

the graphical display of a hyperedge representing a UCM element. They form part of the splines that make up the use case paths. Occasionally it is necessary to have elements of splines that do not relate to map elements as placeholders when paths enter or leave forks or joins. The example UCM contains two NullFigures which form the start of the splines leaving the branches of the or fork. These are not bound to any hypergraph element but are rather resources of the fork or join. They are necessary as it is desired that paths start and end at their corresponding input/output branches of a fork or join and not at the central coordinate of the figure representing the fork or join.

Figure 9: Example UCM and its visible objects

The following diagram, figure 10, shows the internal data structures for the example UCM. All of the UCM elements and empty points are represented by hyperedges which are connected by nodes to form the hypergraphs of use case paths. These hyperedges are related to the Figure objects which are displayed on the screen, as seen in figure 9, by a one-to-one mapping. Path label objects manage the user entered scenario labels that exist for each distinct path segment. The diagram shows the components A and B ref-

erencing the hyperedges contained inside its boundaries, that are in effect bound to it. In practice the components reference the Figure objects but due to the mapping the effect is the same.

Figure 10: Underlying Hypergraph of Example UCM

4.4 Design Issues

Software often needs to be designed as much for practical concerns such as performance and manageability of code as it does for minimization of necessary classes or complexity of code and this is the case with the design of the UCM Navigator. To provide solid data structures for the editor to represent UCM's the use of hypergraphs was chosen. This was motivated by the fact that the lack of underlying data structures made the prototype unstable and its operation harder to analyze. However hypergraphs themselves do not deal with the graphical aspects of a UCM editor such as the fact that each hyperedge that corresponds to a use case map element which has to be displayed on screen with a specific appearance at a certain position and that these elements need to be joined by smooth curves. This graphical functionality and information is separate from the hypergraph model but needs to be provided somehow. The issue is whether to allocate these responsibilities to the hyperedges themselves or create additional objects which manage the graph-

ical output.

The decision was made to separate the concerns of managing the internal data structures and managing the display. This was done for two main reasons. The first is in the area of performance. As the tool uses spline curves to display paths it must minimize the number of spline reinterpolations that need to be done. This can only be done if splines, which represent path segments, can be cached and only recalculated when necessary, that is when one of its elements has changed. This of course can only be done if the information which maps a UCM stored in memory as a hypergraph to a collection of paths segments which are drawn on the screen as splines is made permanent between redraws, that is, is stored in objects which manage these elements. The second reason for separating the concerns of internal data structures and graphical display was that doing so would result in the program structure being more understandable and modifiable.

This separation of concerns between the hypergraph internal model and the graphical display can be seen from the class hierarchy diagram as there are two main trees, that of the Hyperedges and that of the Figures. The Figure subclasses implement the graphical operations required such as methods to draw themselves, support hit detection and notify dependent objects when they are moved, for each Hyperedge which corresponds to a UCM element. The Figure objects are contained inside the Hyperedge objects and there is always a 1-1 mapping between hyperedges and figures. In addition the editor maintains lists of path objects (which reference lists of figures) for all of the path segments in a given use case map to provide a means for interpolating and drawing splines. This main design principle of the editor of having separate data structures for the use case maps in terms of hypergraphs and for the graphical information in terms of figures, paths, as well as software components stems from the aforementioned needs to keep the performance of the tool acceptable and a desire for separation of concerns to keep the code more manageable. Both of these issues will be explored further in the following sections.

4.5 Performance Issues

The issue of performance comes from the fact that cubic splines are used to draw smooth curves for path segments. Interpolation of these splines is very floating point intensive and time consuming. If there were no separate lists of paths maintained during map editing it would require the spline curves to be reconstructed every time the screen is

Application of Use Case Maps to System Design with Tool Support

redrawn (every 0.1s during motion) . Reconstructing the splines would mean the hypergraph would have to be scanned to determine the separate path segments, the graphical positions of the use case map elements would be added in the correct order to a spline representing each path segment and the spline would be interpolated, which as stated previously is an expensive operation. By maintaining separate lists of paths, (corresponding to path segments in UCM's) , these splines can be cached in that they can be reinterpolated only when needed, when one or more of their elements has changed position since the last screen refresh. This minimizes the number of time-consuming spline interpolations that need to be performed and requires separate path pools to be maintained for map objects.

The following illustrations show the graphical objects involved in the UCM Navigator display. An example UCM is shown in figure 11 along with all of its graphical elements. The top illustration shows the example path drawn with the UCM Navigator. The middle illustration shows the Figure objects that are visible on the screen. These graphical objects relate to the hyperedges which provide the underlying data structures for the map. The figures are labelled by their concrete Figure subclass and correspond to the Figure objects in figure 12 which shows the relationships of the graphical objects in the example map. The notation used in figure 12 for object containment and referencing relationships is from the Unified Modelling Language [12] . The bottom illustration of figure 11 displays only the spline curves which interpolate the visible objects in the middle illustration. They are labelled with spline numbers and are meant to provide context for figure 12.

Figure 11: Graphical Objects in UCM Navigator Display

ure objects which represent them. The use of the aforementioned NullFigures from figure 9 is visible here. These objects are placeholders for the starts and ends of paths and do not each correspond to an underlying UCM element as do HyperedgeFigure subclass objects but rather are contained by the Figure object representing a fork or join. They are dependent objects of their parent HyperedgeFigure objects in that when the parent object is moved all of the path segments that are connected to that element are marked so that they are reinterpolated during the next screen redraw. As can be seen from figure 11 and figure 12 NullFigures are the starting points of splines that begin at forks and the terminating figures of splines that end at joins.

The effects on performance are therefore clear as the use of separate graphical objects to cache splines results in object structures that perform the minimum reinterpolation. When any single input single output (SISO) UCM element is moved only that spline is recalculated. The exception is with path compositions such as the asynchronous triggering of wait 1 in the example map of figure 11 where there is a dependent figure relationship between the triggering empty and the wait as can be seen from the referencing relationship visible in figure 12 between the PointFigure objects representing the empty and wait. The PointFigure subclass handles the graphical responsibilities of many SISO UCM elements such as start points, empty points, result points, wait points, and timers. When a multipath UCM element such as a fork or join is moved all of its connected paths are recalculated both by the Figure object representing the UCM element moving which flags the path as modified and by messages that are sent to each of the element's NullFigures which are endpoints of the paths entering or leaving the fork/join that flag the splines of which they are an element as modified and in need of recalculation. In any event the number of spline recalculations is minimized. If separate graphical data structures such as paths which represent the splines were not kept all of the splines would have to be recomposed by scanning the hypergraph data structures and recalculated at each screen redrawing which would make the performance of the tool unacceptable for maps of any complexity and the UCM Navigator would be unusable.

4.6 Achieving Separation of Concerns

The use of separate classes to manage the graphical display also provides a separation of concerns which improves the understandability of the code. The management of

Application of Use Case Maps to System Design with Tool Support

the graphical display is quite complex. During use case map transformations, map elements needed to be added to and removed from various splines. In the case of multipath elements such as Or forks/joins and And fork/joins splines may need to be cut or joined. Also there is a need to install placeholder figures (which do not relate to a hyperedge) which are the aforementioned NullFigures from figure 9 and figure 12 so splines of path segments entering or leaving one of the many paths of a fork or join begin or end at the correct place. Even during normal display elements need to draw themselves and notify dependent figures when they are moved.

This graphical management should not be performed by the hypergraph elements directly as the code would become less manageable. They do however need to be initiated by the hypergraph elements as they are responsible for performing all UCM transformations. This is achieved by making calls to the utility object ConnectionManager (CM) which manages the graphical display on screen and performs all of the necessary path operations on the paths of the current map. The CM then performs the necessary path manipulations using the figures of the hyperedges involved in the transformation.

Therefore the concerns are separated as follows. It is the responsibility of the hypergraph to store the structure of the use case paths and of each individual hyperedge to perform the transformations relating to that part of the map. For example an and fork is responsible for adding a branch to itself. The graphical part of these transformations is passed on to the CM. It is the responsibility of the graphical classes which constitute the Figure subclasses, the Path Class and the utility CM singleton object to provide a correct graphical display for the use case map whose logical structure is stored inside the hypergraph and to manage this display when the map is redrawn, resized, and elements are moved.

4.7 UCM Navigator Classes

The following diagrams show the classes in the design of the UCM Navigator. The inheritance hierarchy is shown in figure 13. The classes with an asterisk are those that existed in the prototype. Other relationships among the classes in the design, excluding inheritance, specifically containment and referencing are shown in figure 14. Unified Modelling Language notation [12] is used for figure 13 and figure 14.

Figure 13: Class Hierarchy Diagram of UCM Navigator

Figure 14: Class Relationship diagram(excluding inheritance)

The component context diagram of the ConnectionManager, the object which manages the graphical display is shown in figure 15 and that of the Hyperedge object is shown in figure 16.

Figure 15: CCD of ConnectionManager

Figure 16: Hyperedge CCD

4.8 Description of Classes in the UCM Navigator

The previous class hierarchy diagram and class relationship diagram show the inheritance and other relationships among the main classes in the UCM Navigator. The two main hierarchies in the design are that of the Hyperedge subclasses and that of the Figure subclasses. Hyperedges and Figures form one-to-one object relationships with each other as hyperedges represent the use case map elements and are organized into hypergraphs representing use case maps in memory and figures are responsible for handling the graphical part of the display. They have methods to draw themselves, to detect user selection of their image and form part of the splines which are used to draw smooth curves for the use case paths. They also are bound to software components.

The remaining classes either support the hypergraph model or graphical display such as Hypergraph, Node, Label, Path, and Component or are one of several manager classes which are singleton objects which contain the code for managing a specific part of the editor. A detailed description of all of the classes is given in Appendix A.

4.9 Use Case Maps of UCM Navigator Operation

This section describes the operation of the UCM Navigator in its own terms. It shows the main behaviour signatures through the editor in terms of causal sequences of responsibilities. The decomposition of system design and visualization achieved through stubbing is shown by expanding common operations into stubs which can then be included in any number of higher level maps for scenarios which may perform these operations although showing the operations explicitly would make the maps too complex.

The first three UCM's are in fact stub expansions for the common editor operations of selecting an element from a map, drawing the screen and validating the applicability of the possible transformations. These operations occur during all UCM transformations performed by the users of the tool.

The following UCM shows the scenario for the creation of an And Fork at a certain point in a path. The stubs for Select And Validate are present at the start of the path as those functions are performed prior to all UCM transformations. Note that the creation of all of the necessary hyperedges/nodes needed for this operation and their linking into the hypergraph are responsibilities of the hyperedge of the current figure slot. It is always the

Application of Use Case Maps to System Design with Tool Support

case that the responsibility for modifying the hypergraph lies with the selected hyperedge, i.e. the selected UCM element. Note also that the responsibilities of manipulating splines lies with the ConnectionManager itself.

The following UCM shows a scenario in which a stub with a new submap is created. It essentially shows two separate operations, the creation of the stub, and the creation of the submap but is assumed to be continuous as the precondition is that the user chooses to edit the submap immediately.

Figure 18: Draw Screen Stub Expansion UCM

Figure 19: Validate Transformation Stub Expansion UCM

Figure 20: And Fork Creation UCM

Figure 21: Stub Expansion UCM

5. Generation of UCM Linear Form

5.1 Introduction

The linear form for use case maps was created by the research team on UCM's as a way of providing context-free text (EBNF) descriptions of UCM's that may be used as input to other tools. It is one of the contributions of this thesis to include in the UCM Navigator graphical editor the ability to generate this textual description of the structure of the UCM's that are drawn in the editor. This chapter will describe the procedures and algorithms used to translate the structure of the UCMs drawn by the users, which are internally stored as hypergraphs, into the UCM EBNF form.

5.2 UCM Linear Form Structure

An extended description of the UCM linear form created by the UCM researchers can be found in [13]. A discussion of its general structure is necessary before the translation can be described.

The linear form is designed to be able to describe complex designs which may contain many top level maps and submaps which are plug-ins for stubs in either top level maps or other submaps, thereby allowing many levels of recursion. At the top level the linear form describes designs by listing the descriptions of all of the maps in the design in one of two categories, root maps or plug-ins. Connections between a stub in a higher level map and its plug-in terms of the binding of the inputs and outputs of the stub to the start and end points of the submap are made by the user of the tool by matching names of input/output points of the stub to those in the submap. These bindings are included in the linear form description of the stubs and form the basis whereby multilevel UCM designs are logically connected.

The linear form description of each use case map is given the keyword **model**. Apart from characteristics such as name, title, and description the description of a **model** is given by the three specifications of its component structure, its path structure, and the specification of the responsibilities in the map.

The structure specification describes the hierarchy of software components in a given map. The characteristics of each component are given which include all possible

child components, the responsibilities which are allocated to this component, as well as all other UCM elements which lie within its boundaries.

The specification of use case paths is the most complex specification in the linear form. It is broken into four sub-specifications, **path atoms**, **semi-paths**, **paths**, and **path composition**. This separation is necessary as a map may consist of many different end to end scenarios with elements belonging to several scenarios. The term **path atom** refers to the smallest components of paths namely UCM elements (responsibility, fork, join ...) and their specification comprises a list of all of the elements in the map with generated input and output identifiers. The term **semi-path** refers to the partial paths between multi-path UCM elements. They are composed of path atoms and are specified by listing the connections between the generated input/output identifiers. The **path** specification refers to specifying the various end to end scenarios in the map as determined by the user labelling. **Paths** are composed of component **semi-paths** and are specified by listing the connections of their component semi-paths. **Path composition** refers to specifying the relationships whereby one path triggers another.

The responsibility specification is simply a listing of all of the responsibilities in the map and a description of their full characteristics, including their description and any possible structural dynamic characteristics.

5.3 Example of Linear Form Generation

The following pages contain an example of the generation of the linear form from a simple UCM . This example UCM contains many elements of UCMs such as multiple scenarios, path composition, binding of responsibilities and nested components. The example UCM itself is shown in figure 22. The following diagram, figure 23, shows the underlying hypergraph structure of the map from which the linear form is generated. The diagram which follows, figure 24, shows the steps involved in the process of translating the internal object structure into the linear form. This diagram is not meant to explain the entire process but rather to show which of the objects that comprise the internal data structures of the example UCM shown in figure 23 are involved at each stage of the translation. The actual generated EBNF follows in section 5.3.1. The following sections explain the process of generation in detail.

Figure 22: Example UCM to illustrate linear form generation

The above map is a sample UCM that was drawn to demonstrate the generation of the linear form output from a given map. The underlying hypergraph and object structure is as follows in figure 23.

Figure 23: Data Structures Present in Example UCM

Figure 24: Object Structure to Linear Form Transformation

Figure 24: Object Structure to Linear Form ... (continued)

Figure 24: Object Structure to Linear Form ... (continued)

5.3.1 UCM Linear Form Generated From Example Map

The following is the linear form that is generated by the UCM Navigator for the example map shown in figure 22. As can be seen by comparison with the map the structure is retained in the textual form. The map elements become path atoms which are connected into semi-paths. Varying collections of these semi-paths then form the paths of the end to end scenarios A, B, and C. The interpath connection visible as path C triggers wait1 asynchronously in the map is recorded as path compositions between C and A, B. The component structure follows directly from the map.

design linear-form

root maps {

model root

/* Example of linear form generation */

structure

components {

component A

is a Team

included structure

components {

component B

is a Object

responsibility references { resp B ; resp C ; }

atoms references { f1; j1; }

actual ;

component C

is a Object

responsibility references { resp D ; }

actual ;

}

pools { }

actual ;

}

pools { }

path specification

path atoms {

start s1

part of paths { A ; B ; }

out os1 ;

responsibility reference resp A

in is1

Application of Use Case Maps to System Design with Tool Support

```
    part of paths { A ; B ; }
    out os2 ;
wait wait-1
    in segments { is2 ; }
    part of paths { A ; B ; }
    out segments { os3 ; } ;
fork f1
    in is3
    part of paths { A ; B ; }
    out segments { os5 ; os6 ; } ;
responsibility reference resp B
    in is4
    part of paths { A ; }
    out os7 ;
join j1
    in segments { is5 ; is6 ; }
    part of paths { A ; B ; }
    out os8 ;
responsibility reference resp D
    in is7
    part of paths { A ; B ; }
    out os9 ;
end e1
    in is8
    part of paths { A ; B ; } ;
responsibility reference resp C
    in is9
    part of paths { B ; }
    out os10 ;
start s2
    part of paths { C ; }
    out os11 ;
responsibility reference resp E
    in is10
    part of paths { C ; }
    out os12 ;
empty empty1
    in is11
    part of paths { C ; }
    out os13 ;
end e2
    in is12
    part of paths { C ; } ;
```

Application of Use Case Maps to System Design with Tool Support

```
}
semi paths {
  semi path sp1
    links {
      link os1 with is1 ;
      link os2 with is2 ;
      link os3 with is3 ;
    };
  semi path sp2
    links {
      link os5 with is4 ;
      link os7 with is5 ;
    };
  semi path sp3
    links {
      link os8 with is7 ;
      link os9 with is8 ;
    };
  semi path sp4
    links {
      link os6 with is9 ;
      link os10 with is6 ;
    };
  semi path sp5
    links {
      link os11 with is10 ;
      link os12 with is11 ;
      link os13 with is12 ;
    };
}
paths {
  path C sp5;
  path B
    connections {
      connect sp1 with sp4;
      connect sp4 with sp3;
    };
  path A
    connections {
      connect sp1 with sp2;
      connect sp2 with sp3;
    };
}
```

```
    path compositions {
        join path C and A through empty empty1 and wait wait 1;
        join path C and B through empty empty1 and wait wait 1;
    }
    responsibilities {
        responsibility resp A ;
        responsibility resp B ;
        responsibility resp C ;
        responsibility resp D ;
        responsibility resp E ;
    }
}
```

5.4 The LinearGenerator Linear Form Generation object

The generation of the linear form is managed by a singleton object in the UCM Navigator which is named LinearGenerator (LG). It is responsible for controlling the generation by calling appropriate Generate() method of the proper objects, for opening and closing the file used, and for storing all temporary data structures and partially generated output. It contains the main Generate() method to generate linear form output for a given UCM design. It contains interface functions which allow the generation methods of individual hyperedges, which are called when the hypergraph is traversed in a depth first manner, to notify the LG of structural elements in the map, for example when new semi-paths are to be begun or when sections need to be added to the current semi-path. The LG is responsible for managing all information relating to the linear generation including generated identifiers, and temporary buffers but is controlled by the generation methods of the hyperedges and hypergraph as that is where the UCM structure is stored and thus where the generation should be controlled.

5.5 Generation of Component Structure Specification

The generation of the component structure specification is straightforward as all components contain a list of child components and a list of all the graphical figures that lie within its boundaries (this is used for graphically binding paths to components). All Component objects contain a Generate() method which outputs all of the desired informa-

tion which includes the components characteristics, that of its child components and the responsibilities/other atoms that lie within its boundaries. This method returns immediately for components which are child components of some parent to prevent components from being included multiple times. The lists of responsibilities and other atoms are generated from the list of graphical figures simply by querying each figure as to whether it is a responsibility or other atom. As many UCM elements are not given names by the user but are given generated identifiers prior to generation of component structure information all unnamed hyperedges in map are given these identifiers which are then used throughout the linear form specification of a map.

5.6 Generation of Path Structure Specification

The generation of the path structure from the hypergraph describing the UCM is the most complex part of the translation. It is the responsibility of the hypergraph object which generates the information either directly as the hypergraph is traversed or indirectly by storing information in temporary data structures for later generation of linear form code. These temporary data structures are stored in the LinearGenerator singleton object.

5.6.1 Hypergraph Traversal

In order to transfer the structure of the hypergraph in memory to the linear form output the hypergraph needs to be traversed in some manner. As the path specification is meant to show the connection of path elements to their neighbors the best way is to simply follow the path from beginning to end. A depth-first traversal is made from the start point of the map and which follows the map element by element. When a fork is reached each branch is followed to its end before the next branch is traversed. The polymorphic Generate() method that all hyperedges define is called for each hyperedge encountered. This allows the information necessary for the generation of path atoms, semi-paths and paths to be recorded, as will be explained in the following sections.

As there may be several paths in a given map a depth first traversal is made for each Start hyperedge in the map. As different start points may synchronize later at a path that has already been traversed to avoid multiple traversals of the same sections there is a counter kept in each hyperedge and in the LinearGenerator object of the number of times a map has had the linear form generated. These counters are compared at the beginning of

the Generate() method. If they are equal the section has already been traversed and the traversal is stopped.

5.6.2 Generation of Path Atoms

Path atoms form the smallest elements of the path description and correspond to individual map elements. Each visible UCM element outputs its characteristics including its identifier (either user given or generated) and the connections it has to other path atoms. The identifiers that atoms use for their input and output segments are obtained from the Linear Generator (LG) which determines them with simple counters. The output identifiers are stored in all of the following hyperedges so that each element knows the identifier of its predecessor, which is necessary for depth first traversals. The Generate() method also calls a LG function with its path label as a parameter so that the list of all scenarios of which the atom is part can be included in the path atom description.

The possible exception to every hyperedge generating a path atom output is the empty hyperedge which as stated previously is a placeholder in the UCM model. It is included in the linear form output only if it contains useful information such as pre/post-conditions, if it triggers another path asynchronously, or if it contains characteristics such as being a failure point of a path or denoting a shared responsibility along a path. Barring any of this useful information the Generate() method of an Empty returns without outputting anything to the linear form. It does however install the output segment identifier of the previous hyperedge into the next hyperedge so the next meaningful path atom will always know the identifier of the previous meaningful element allowing any number of unnecessary empty hyperedges to be skipped during the generation.

5.6.3 Generation of Semi-Paths

The generation of semi-path information is done concurrently with the generation of path atoms when the depth first traversals of the hypergraph are made. This is possible by noting that semi-paths are the unambiguous partial paths that lie between ambiguous atoms (atoms with more than one input or output) and that the individual atoms (hyperedges) know that they should end or begin semi-paths. For example a Start hyperedge always starts a new semi-path while a Result hyperedge always terminates the semi-path of which it is a part. Ambiguous or multipath hyperedges such as forks and joins also con-

tain this knowledge. A fork, either Or or And, terminates the semi path of which it is part and begins new ones for each output branch. A join, either Or or And, terminates the current semi path for each input branch and begins a new semi-path for its output branch.

The Generate() methods of the hyperedges perform this generation of semi-paths by calling functions of the LinearGenerator (LG). It contains interface functions for starting a new semi-path, for adding a new link to an existing path, and for terminating the current semi-paths. The identifiers of these semi-paths are simply generated numbers. The parameters to the functions for adding links and terminating semi-paths are the integer identifiers of the input and output segments involved in the link. The function to create a new semi-path needs no such parameters as the first link is created by the second atom in a semi-path.

The LG generates the linear form output based on the parameters it is given. As the specification of semi-paths occurs after that of path atoms in the linear form, in order for the generation to be performed concurrently, the output is added to a large character buffer for temporary storage. When the generation of path atom information is completed a LG method is called which outputs this temporary buffer to the file.

5.6.4 Generation of Paths

The generation of the specification of paths, which are the end to end scenarios thorough a map, is performed partly during the specification of paths and semi-paths as the hypergraph is traversed and partly after it is completed. Paths are specified by connections of semi-paths. As these connections are made at ambiguous atoms all information for most paths (other than those consisting of a single semi-path) can be obtained by searching through the hypergraph for these hyperedges and storing the information for later generation. The information is best stored as opposed to being used to generate the path specification immediately as all of the information for each scenario needs to be outputted prior to another that of another scenario. As there may be many scenarios involved at the ends of multipath hyperedges direct generation would require the hypergraph to be searched multiple times, one for each unique scenario.

These interpath connections are recorded by iterating through the hyperedge pool and sending the GeneratePathConnections() polymorphic message to each. Simple hyperedges simply ignore this call while multipath hyperedges provide an implementation

which calls a LG method which records each connection between every input and every output. The parameters of this method are the path labels of the input and output paths of the multipath hyperedge. The path labels are used as the LG method which creates a new semi-path during the traversal of the hypergraph stores the semi-path identifier in the path Label object thereby providing each hyperedge with knowledge of its semi-path identifier.

Once all of these connections have been stored in temporary data structures inside the LG the generation of path specification begins. The pool of Label objects for the current Map object is searched to find the set of scenario labels. The path specifications are then generated by, for each unique scenario label, searching these data structures for those that contain the label in both their input and output. In that case both semi-paths form part of the path specification for that scenario and a new connection is added to the specification with both semi-path identifiers specified.

5.6.5 Generation of Path Composition Specification

The generation of path composition information, referring to the triggering of paths by other paths, is performed at the end of the path structure generation. It is performed simply by scanning the hypergraph for Connect hyperedges which denote inter-path connections. For each Connect found a LG method is called with the parameters being the two hyperedges involved in the connection. This method uses the path labels of the two hyperedges to determine all of the possible combinations of scenarios synchronously or asynchronously triggering other scenarios. All of these individual path compositions are then output to the linear form.

5.7 Summary of Linear Form Generation Algorithm

The following pseudocode provides a summary of the major steps involved in translating a UCM design created with UCM Navigator into linear form output.

```
for all maps in design
  generate atom identifiers for unnamed atoms
  generate component structure specification
  determine set of start points in map
  for each start point
    perform depth-first traversal of hypergraph
    while traversal not finished
      while current path not finished
        call Generate() for current hyperedge
        if current hyperedge is ambiguous
          start new semi-path
          store semi-path id in label
        else
          add section to current semi-path
        endif
        advance to next hyperedge in graph
      endwhile
    endwhile
  endfor
  generate path specification
  generate path composition specification
  list responsibilities in map
endfor
```

6. Extensions for Performance Prediction

6.1 Motivation for using UCMs for Performance Prediction

This chapter describes the work done as part of a Nortel ISRP done with Craig Scratchley relating to the use of the use case map tool as a front end for performance simulations[14]. This work was motivated by the desire to use UCMs in the area of performance prediction. It was noted that UCMs were quite similar in many respects to Activity Sequence Networks (ASN) which is the notation used to describe systems for performance purposes. If somehow the UCM Navigator could be extended with the appropriate performance annotations then it could be used as a graphical front end for these simulations.

This chapter examines the extensions that were made both to the tool and to some traditional ways of thinking about UCMs that were necessary so that models could be described in terms of UCMs with appropriate performance annotations and then simulated.

This part of the study of the use of UCMs with tool support is important in that it shows a different application of UCMs as opposed to the traditional requirement engineering description of a software system. The requirements for performance modeling are quite different than that of requirements engineering and thus many facets of UCMs will have to be re-examined and possibly changed, at least in the context of using UCMs for performance modeling, as will be shown later.

6.2 Comparison - UCMs and Activity Sequence Networks

As was mentioned in the previous chapter the motivation for the ISRP for the use of the tool in this study as a front end for performance simulations was both the desire to use the UCM methodology and tool for performance prediction and the realization that UCMs were very similar to the Activity Sequence Networks (ASN) that were used on the performance side of this project, to describe the behaviour of software. If somehow the performance information that was present in the ASNs could be added to the UCM Navigator then the UCM tool could be used as a front end for performance simulations by transforming the UCM structure in terms of hypergraphs into ASNs. ASNs were neces-

Application of Use Case Maps to System Design with Tool Support

sary for the performance simulations as they were already programmed to use ASNs as the description of the software behaviour.

The following illustration, Figure 25: Comparison between UCM and ASN, compares the structure of UCMs and ASNs. An example UCM is drawn at the top of the diagram and its equivalent ASN appears at the bottom. Labels and arrows show the correspondence between UCM and ASN elements. Start and Result points in UCMs correspond to Input and Output Points in ASNs, Responsibilities in UCMs correspond to Activities, OrForks and OrJoins correspond to Decision and Merge Points, and AndForks and AndJoins correspond to Split and Join Points. What this diagram is meant to show is that there is a simple correspondence between UCMs and ASNs and so therefore UCMs can, with necessary additions, be used as a notation to describe the performance model of a system. What the illustration does not show is that the differences between the two notations in terms of path interpretation and the extra performance oriented data that is attached to ASNs needs to be accounted for.

The purpose of this section is not to discuss the specifics of the implementation of the performance simulations but rather to provide a foundation for this chapter in terms of determining the changes that are necessary to UCM's, both to the tool and to the standard interpretation of UCMs, for their use in performance prediction. As ASNs are used to model the behaviour of software execution for performance analysis they contain the information necessary for a simulator to simulate the execution of a system. As UCMs were created mainly for the high level scenario based design and requirements analysis they do not have this information. A comparison of UCMs to ASNs of their similarities and more importantly their differences will show what is needed to add and what restrictions need to change about UCMs so that they can describe the behaviour of an executing system to a simulation environment as well as ASNs. In fact once the required extensions are added to the UCM concepts and correspondingly to the tool performance simulations could theoretically be performed directly from the information contained in the UCMs.

Use Case Maps and ASNs are similar in that both are path notations [15] that show causal sequences of processing that must be performed by software, which are termed responsibilities in UCMs and activities in ASN. UCMs show these sequences of actions as responsibilities that are performed along paths and that may be bound to components. ASNs show sequences of actions as a series of ASN activities which are connected by arcs, split and join points and decision points to form "paths" of activities. They also both

Application of Use Case Maps to System Design with Tool Support

show synchronization of execution paths, divergent execution paths and the triggering of one path by another.

Figure 25: Comparison between UCM and ASN

The main differences between the two lie in the fact that UCMs were created for requirements analysis while ASNs were created to provide a means for software designers to describe their work to performance modellers and as such attempt to describe how software actually executes. These differences concern both the interpretation of paths and the hiding of lower level detail as well as the existence of components.

6.2.1 Differences in Path Interpretation between UCM and ASN

The interpretation of a path differs with UCMs and ASNs. With UCMs a scenario is viewed as an end to end path through a system that occurs whenever certain preconditions are met. Any forks in a map are simply the effects of path superposition as no decisions about which branch to take are made. The scenario is specified completely from beginning to end. Although decisions are often taken in the real life software systems the UCMs describe about which branch to take, at the UCM level such decisions are considered lower level detail. The interpretation with ASNs is quite different. As ASNs attempt to describe how software actually runs for simulation purposes, the decision points decide on the path to take based on user data. This can be a simple probability or a function that decides the path to be taken based either on the data that it is sent or global data.

6.2.2 Differences in use of data between UCM and ASN

Another major difference between UCMs and ASNs concerns the hiding of detail. As UCMs attempt to show scenarios above the level of inter-object messages and data low level details such as data are ignored. As ASNs must describe the running behaviour of software they cannot ignore data as actual program execution depends on data. For example the branch to take when encountering a decision point in an ASN is often made based on data. The execution time of ASN activities may depend on data. There are also important mutual exclusion implications. Global data in an application would need to be protected by locks to avoid simultaneous updates by multiple threads. Activities which access these global data stores would need to acquire these locks. As they could be blocked on trying to obtain these locks the run-time behaviour of the system would be affected. Therefore any accurate simulation of such a system must simulate this behaviour. In short as actual program execution depends on data a simulator would need this information and ASNs or any other notation that attempted to describe software behaviour for simulation purposes would need to provide it.

ASNs provide for the specification of two types of data, token data which are passed from one activity to another and data stores which are global to an ASN and can be shared among activities and decision points.

With UCMs data is often considered to be lower level detail. There is always the possibility of creating data objects with responsibilities such as get value and store value.

Application of Use Case Maps to System Design with Tool Support

However this option is awkward and inadequate for showing access to global data stores. It is desired to specify the global data stores accessed by a responsibility as part of the responsibility while the use of data objects would mean that accessing the data would be separate responsibilities. Also a responsibility could access many data stores and since of course data stores would be shared it would be very difficult to express such data access visually.

The illustration, Figure 26: Differences between UCMs and ASNs, repeats the basic example UCM and its corresponding ASN from figure 25 with captions explaining the different interpretations of the two diagramming notations and the different information that is managed by each such as the arrival process information in the ASN input point, the hardware resource demands that are specified in the ASN activity resource function and the functions which choose the path to take at an ASN decision point.

6.3 Extensions necessary to UCMs for Performance Prediction

The previous chapter described some of the differences between ASN's and UCM's in their description of the running behaviour of software. This section will describe the extensions that need to be made to the UCM descriptions so that UCMs can be used for performance simulations.

In order to simulate software and determine whether the software meets performance requirements the UCM model needs to:

- Specify choices between scenarios
- Provide a means to determine whether desired response time requirements are met
- Provide a means of specifying input events
- Provide a means of specifying the resources used by a responsibility as well as any global data stores that are referenced

The above elements allow the UCM model to specify how software executes in practice, which hardware resources are used, how access to shared data is controlled and how events are generated which is the information a simulation environment needs to accurately simulate a system. The ability to determine whether a system performs well or not in terms of satisfying maximum execution time requirements between certain events is important as the main reason to perform simulations is to determine if performance is adequate.

Figure 26: Differences between UCMs and ASNs

6.3.1 Specification of choices between alternate scenarios

Software often makes decisions about future execution paths based on current data such as when an or fork is encountered. The UCM methodology is designed to show end to end scenarios through systems and considers or forks as artifacts of path superposition where once a scenario is started it is followed to the end. In terms of the actual program execution it is not known which scenario is executing as the path taken depends on run time decisions at the or fork points. This is the conflict between the UCM way of thinking and the actual execution of software.

This is also one of the two areas where UCMs simply need to be interpreted differently for their use as performance models as opposed to their use in requirements specification. The other area is in the area of the specification of global data stores where restrictions on the hiding of data through layering need to be relaxed. This will be discussed later. As a performance simulation would always be more accurate if the model that the simulator simulates behaves exactly as the real software would. Therefore the UCM model should specify how run time decisions are made regarding branch selection at or forks.

What was discovered in this part of the study was that all of the methods of preserving the end to end scenario concept where no actual choices are made would result in either an unscalable and awkward method for specifying the starting of scenarios, less accurate simulation results, or both. If the choice of the branch selected were made based on a relative frequency for each branch as might be used as a first approximation in a performance simulation it would be possible to calculate the percentage of occurrence of each end to end scenario. These percentages could then be used to specify an arrival process for each individual scenario. However as the number of forks increases the number of scenarios increases exponentially making such an approach impractical. However as the choice of the branch does depend on run time data in actual systems not allowing the user of the tool to specify a run time branching choice would make simulations much less accurate.

6.3.2 Verification of Response Times

As the main reason for performing a performance simulation would be to predict performance and determine if system response times are adequate. The ASNs that are used

Application of Use Case Maps to System Design with Tool Support

for the simulations have the added feature of timestamp points along arcs where tokens which pass can be timestamped with the current time. This allows the elapsed time between any two points on an ASN to be determined and thus compared to the desired response time. This extension needs to be added to use case maps so that the time between any two points reachable from each other can be recorded during simulations. It is also desirable to be able to specify requirements on response times so that the simulator can determine whether these requirements are met.

6.3.3 Specification of Input Events

Any simulation needs to specify how input events are generated so that simulations may begin and the rate of input events being created in the system mirrors the situation with the actual software. A scenario in a UCM is said to begin when its preconditions are met. For performance simulations this is of course inadequate and statistical information needs to be given so that a simulator can properly generate starting events for scenarios according to the expected distribution.

6.3.4 Specification of Responsibility Characteristics

In order to properly simulate a software system the simulation environment must be aware of the resources (processing time, disk usage) that activities require and also the global data stores which are accessed as there are mutual exclusion implications. With ASNs resource functions [15] are used to specify the amount of each resource used by each activity. These functions use token data as their input and calculate the resources needed based either on data or random variables. Access to global data stores is shown graphically by dashed lines emanating from activities to the data stores which are represented as oval boxes in an ASN.

The UCM model needs to be extended to allow these characteristics to be specified for responsibilities. The responsibility specification currently allows for name, description and optional structural dynamics arrows. This needs to be expanded to include the characteristics of the resource functions as well as the list of data stores accessed and their access modes(read, write, read/write).

The specification of these global data stores is another area in which traditional UCM rigidity needs to be relaxed. Much of the data that takes the form of global data

would be considered lower level detail in the UCM methodology and would be ignored. If UCMs are to be used for performance simulation such data cannot be ignored as it has important mutual exclusion properties. The use of data objects to represent this data is unsatisfactory for the reasons presented previously. The simplest and most powerful solution is to allow responsibilities to refer to the data stores even if they are normally part of a lower layer.

6.4 Extensions to UCM Navigator for Performance Prediction

The previous section specified the extensions that needed to be added to the UCM model so that performance simulations were possible. This section describes the extensions to the UCM Navigator that were made to allow performance annotations to be attached to UCM models. Many of the extensions are simply performance annotations, often simple text strings, which are passed on to a simulation environment. Others require management of information across a design, such as response times and data stores as will be explained.

Not all features of the UCM Navigator implemented in this thesis are usable by performance simulations by the extensions that are described here. The exception is that there is no current support for the use of dynamic stubs with multiple plug-ins by performance simulations. While static stubs used for decomposition are supported (and are used in the case study in Section 7.5: Case Study B - Performance Prediction Project) the specification of a dynamic stub with multiple plug-ins is not usable by a performance simulation as no support has been provided for criteria for run-time plug-in selection by the simulator. Similar to the fact that the concept in traditional UCM interpretation of scenarios logically beginning when their preconditions are met is not adequate for performance simulations and is replaced by a time distribution in the performance extensions the use of preconditions for plug-ins to determine which plug-in should be selected at simulation time is not adequate. Information similar to that entered by designers to specify choices between scenarios at or forks needs to be provided to specify choices between the various plug-in maps of a dynamic stub at run-time. This was not done as the current designs used in the performance project are fairly simple and do not need this powerful feature but nevertheless the support of dynamic stubs by performance simulations is very feasible.

6.4.1 Timestamps

In order for a simulator to measure the execution time between points on a path timestamps were added to the UCM model. The **Timestamp** class is a type of hyperedge which can be placed anywhere along a UCM path. They have the characteristics of a user given name as well as a user defined flag which allows users to specify whether the timestamp denotes the completion of the previous map element or the start of the next element. They appear on the screen as triangles pointing into the path. Initially they are drawn in outline but appear filled when they are involved in a response time requirement as mentioned below. Timestamps provide the simulation environment with path markers where the elapsed time can be recorded allowing elapsed times between path points to be calculated.

6.4.2 Response Time Requirements

The UCM Navigator was also extended to allow users to specify response time requirements between any two timestamps that can be reached from each other. The class **ResponseTime** encapsulates this information. The collection of response time requirements is global to a UCM design regardless of how many levels the design is composed and is managed by the utility object **ResponseTimeManager**. Requirements can be created by selecting any two timestamps reachable from each other and entering the time requirement, the percentage of time that this requirement must be met, along with a name for the requirement. Once created the list of requirements for either the entire design or a specific timestamp can be viewed and edited.

6.4.3 Arrival Processes for Scenarios

In order to allow specification of input events for paths the editor was extended to allow arrival processes to be specified for start points of paths. The user, through a dialog, can specify the arrival process type (exponential, uniform etc.) as well as the numerical characteristics of the distribution. This information as is all of the performance information is passed on to the simulation environment.

6.4.4 Specification of Choices between Scenarios

As discussed in previous sections it was decided necessary to allow the user to specify how choices between scenarios at or forks are made. As the number of possibilities is large, ranging from simple probabilities to logical conditions the decision was made to allow the user to specify a simple text string which could be parsed and interpreted in an appropriate manner by a simulation environment.

6.4.5 Specification of Resource Functions for Responsibilities

As performance simulations require knowledge of which hardware resources are used by activities the description of UCM responsibilities has been expanded to allow users to specify resource functions as simple text strings. These text strings are passed to the simulation environment.

6.4.6 Mutual Exclusion of Data Stores

The definition of UCM responsibilities has been expanded to allow the specification of a set of global data stores which are in fact layered data in the strict UCM interpretation. They are specified through a dialog which lists the data stores accessed and their access modes. The user chooses from lists of all of the available data stores in the design as well as a list of all available access modes.

The lists of data stores and access modes for a particular design are created separately by a dialog invoked from the main window. The list of available data stores is managed by the utility singleton object **DataStoreDirectory**. The list of data stores referenced by each responsibility is in fact a list of references to indexes in the directory.

6.4.7 Specification of Hardware Devices

As performance simulations must be aware of the hardware demands of activities they must be provided with the characteristics of hardware devices such as processor speeds and disk access times. The UCM Navigator has been extended with a dialog invoked from the main window allowing users to specify characteristics of classes of devices (processors, disks, DSP ...) as text strings. The singleton object **DeviceDirectory** contains all of the descriptions of the hardware devices supplied by the user.

6.5 Linking of UCM Navigator to Simulation Environment

The previous sections described the annotations that users can make to UCM models for their use as performance descriptions. This section will describe the linking between the tool and the simulation environment.

There are several possibilities for linking the two environments. The linear form whose generation was described in the previous chapter could be extended with the performance annotations and then parsed by the simulator. While this is the most robust long term solution, which may be performed in the future, it was not chosen as parsing the linear form and generating the ASN models from the abstract syntax tree would be a daunting task. As the purpose of this ISRP is the proof of concept that UCMs can be used for performance simulations a simpler alternative was employed.

The fact that UCMs and ASNs was taken advantage of by reusing the UCM Navigator code for file storage so that the UCM output file could be used to recreate the hypergraph in memory and generate the ASN from its structure.

6.6 Conclusion

This chapter discussed the issues involved in using UCMs as a means of providing the software description to a simulation environment. At the least this would require that the tool capture performance information that could then be passed on to a simulation environment. What also was required was a re-examination of some of the concepts of use case maps. What was discovered was that there were clashes between how UCMs and other notations such as ASNs represented software behaviour. This resulted directly from the different purposes of UCMs and other methodologies. Use Case Maps were designed for requirements engineering to show end to end scenarios through systems and to abstract away from lower level detail. Notations that are useful to performance modelling attempt to describe the exact run-time behaviour of software.

These differences occurred in two main areas, the specification of selection of alternate paths at forks and the hiding of lower level detail. It was discovered that, at least for performance modelling these facets of UCMs need to be changed so that software systems can be described with enough detail so that accurate simulations can be performed.

7. Case Studies

7.1 Introduction

In order to provide examples of both projects for which the UCM Navigator was used there will be two case studies, one describing work for the Agent Project and one describing the use of a design specified in UCM terms as the front end for performance simulations as an example of the work done for the UCM Performance Prediction project.

The two case studies are necessary to demonstrate different aspects of the usability of the tool. The case study for the Agent project involves the use of the tool to specify the operation of call processing in which alternative features are expressed as alternative plug-ins. This demonstrates the use of the tool to describe dynamically pluggable behaviour signatures. It also demonstrates the use of the tool as a front end for other programs as the purpose of entering this design into the UCM Navigator was to generate the linear form of the structure which then would be read as input to the Agent tools which were written in Java for the project. These tools would assist in the process in generating code from the behaviour structures in the maps. The case study for the Performance project describes a simple example system described in terms of UCMs that has been annotated with performance information as a means of showing the applicability of the tool as a front end for performance simulations. There was a need for two case studies as no example application makes use of both the performance annotations and the substitution of dynamic behaviour signatures. The Agent project is not concerned with performance and performance simulations do not make use of dynamic stubs with alternative plug-ins.

7.2 Case Study A - Agent Project

This case study describes a UCM design describing a simple agent based telephony application with several features described. This was published as part of a paper on feature interaction for the PAAM conference[16]. The paper describes the feature interaction problem between two common features, Originating Call Screening (OCS) and Call Forwarding (CF). The feature interaction results when a caller uses the call forwarding feature of a second telephone (or even the same telephone) to call a number that is on the prohibited calling list for OCS from their location. The paper describes how this and many

Application of Use Case Maps to System Design with Tool Support

types of feature interactions commonly found in communication systems can be solved by using agent-based techniques to implement such systems which are based on the notion of competing rule engines.

The UCM Navigator enters into this process as the paper describes a semi-automated process for generating these rule engines from UCMs. The process is as follows. First the design is entered into the UCM Navigator with any alternative behaviour signatures being expressed as alternative plug-ins for a dynamic stub. The linear form exports these alternative plug-ins along with the pre- and post-conditions of the paths in the plug-ins. The linear form is read in agent table-based tools to create tables for all alternative plug-ins which relate to alternative agent plans for implementing goals. These tables contain the name of the plug-in, its pre/postconditions, and its responsibilities. Information of lower level that is not present in the UCMs is added to the tables by designers. The tables are then used in later stages for code generation.

Use Case Maps then enter into the process of constructing agent based systems by providing a means by which complex agent interactions can be visualized and designed. These complex interactions involve alternative behaviours which depend on run-time conditions. As these can be described by dynamic stubs with alternative plug-ins with varying pre/postconditions the structure of this complex behaviour will be exported in the linear form. This structure can then be read in by other tools to build tables with designer assistance describing the dynamic behaviour of the system. Therefore the graphical, intuitive nature of UCMs can be used to design agent systems which involve the cooperation of many agents, allowing the visualization of such interaction, and be used as a front end to generate information for later tabular stages of the development process.

It should be emphasized that the UCM Navigator does not detect any feature interactions. It simply provides a means for visualizing feature interactions by means of alternative behaviour signatures which may be selected at run-time. By providing a means to visualize different possible end-to-end behaviour and exporting a textual description of a design which includes the conditions (pre/postconditions) for the choice of the various plug-ins the UCM Navigator may be used to both visualize system behaviour and provide support for agent tools which may help in detecting feature interactions.

The design which appeared in the paper and which was entered into the tool is shown in the following diagrams. The design consists of a root map consisting of objects representing a caller and answerer as well as agents for each of the two. There is actually

only one type of agent, a user agent which fills the role of a Call Side agent or an Answerer Side agent depending on whether its site is initiating or receiving a call. There are four plug-in maps representing alternative behaviour signatures as each agent has two alternative behaviour signatures, one for each feature. These behaviour signatures correspond to different rule engines which compete in a tuple space, which is the communication medium for agents, to determine which pattern will ultimately be chosen.

Figure 27 depicts the root map of the design. There is an object representing the caller and an agent that operates on its behalf. At the receiving side there are stacks representing the possible answerers and the agents operating on their behalf. Stacks, which represent multiplicity in UCMs, are used at the answering side to denote that one caller could potentially signal many answerers. The path across this map shows the causal sequences for a connection request. The scenario is begun when the caller attempts a call. The agent uses one of the plans in the CSP (Call Side Processing) stub act on behalf of the caller. The two features available are the default originating process which simply forwards the call request and Originating Call Screening. At the receiving side the user agent plays the Answerer-Side role and its ASP (Answerer Side Processing) stub contains two alternative plug-ins, one for the default Terminating feature which simply allows a connection to be made and one for the call forwarding feature which forwards the request to another agent.

The feedback path from the ASP stub back to the CSP stub in the Call Side agent denotes the action in the case of the call forwarding stub being selected in which case the request to forward the call is passed back to the Call Side agent to determine if the forwarding number is on a prohibited list. This map is in fact the solution UCM to the feature interaction problem as the feedback path shown solves the feature interaction. The original map in the paper which suffered from the feature interaction problem simply possessed a feedback path from the ASP stub to another ASP stub in another Answerer agent. As it did not check whether the forwarding number was on the prohibited list it suffered from feature interactions between the OCS and CF features. The use of UCMs to visualize end-to-end behaviour of systems of agents helps to visualize potential feature interactions and in some cases, such as this one, suggest a solution.

The paths back from the CSP and ASP stubs to the Caller object denote the notification to the caller that the call has either been accepted, the answerer is busy, or the call has been rejected by the OCS feature.

Figure 27: Root Map - Telephony Feature Interaction Example

Figure 28: Originating - Default Plug-in for CSP

Figure 28 shows the very simple plug-in for the default Originating feature. It and all plug-in maps are bound to their parent stub by the binding process described in Section 3.3.1: Capabilities to support multi-level maps and in [11]. It simply sends a request to an answerer side agent. The screen captures show the responsibilities for the map as well as the preconditions and resulting events for this path which are interpreted as the conditions

Application of Use Case Maps to System Design with Tool Support

for the plug-in as a whole. The sole responsibility `snd-req` specifies that a call request is to be made to the answer side agent. The precondition for this plug-in to be selected is that a number is selected and the resulting event is that a request is sent to an answerer agent. The plug-in map specified in figure 28 does not contain any postconditions for the path or starting events for the path as the designer chose not to include them. The UCM Navigator allows any number of appropriate conditions to be specified including preconditions and triggering events for path start points, waiting places and timers, pre/postconditions for stubs, and postconditions and resulting events for path end points. However it is up to the designer to include any or all of them. Similarly the plug-in map shown in figure 29 does not contain any triggering and resulting events as the designer chose not to include them.

Figure 29: OCS: Originating Call Screening for CSP

Figure 29 shows the plug-in for the call screening feature. A number is checked by the OCSlist component to see if it is in the restricted list. If so the connection is refused, otherwise the call request is sent to the answerer side agent. The responsibilities in the

map are shown in the screen capture and correspond to the actions of checking whether is in the OCS list and correspondingly sending the request or refusing the connection. The preconditions for this path and hence for the plug-in are that a number is collected and an outgoing call is requested. The postcondition for path O is that the call is permitted and that of path P is that the call is rejected.

Figure 30: Terminating: Default Plug-in for ASP

Figure 30 shows the plug-in for the default Terminating feature which returns to the caller whether the phone is busy or available and ringing. The two responsibilities for the map perform those two basic functions. The precondition for the path and hence the plug-in is that there is an incoming call. One of the postconditions for path T1, the caller not busy path is that the answerer is notified. The postcondition for both of the paths is that the caller is notified in any event, either by a ringback or a busy signal.

Figure 31: CF: Call Forwarding for ASP

Figure 31 shows the very simple plug-in for the call forwarding feature which simply in the mind of the observer forwards the request to another answerer agent. The forwarding is not visible by this simple plug-in but is seen by examining the top level map figure 27. As the responsibility involves forwarding, the diversion path in the ASP stub is taken resulting in the forwarding of the request to the other agent. The sole responsibility fwd-req performs this function. The preconditions for this plug-in are that the call forwarding feature is activated and that there is an incoming call. The postcondition for this path is that the caller is notified of a new destination.

The plug-ins collectively show the behaviour that may result in the end to end system depending on run time conditions. The purpose of entering high level designs of agent systems into the UCM Navigator is both to provide visualization of intended behaviour and to capture all of the desired behaviour as well as textual conditions (pre/postconditions, triggering/resulting events) in a format which can be used by other tools. The top level maps provide a behaviour description of the end to end system with dynamic characteristics specified by stubs. The plug-ins for those stubs specify the behaviour in terms of sequences of responsibilities to be taken under certain conditions. As the textual conditions specify the selection of these dynamically pluggable behaviour signatures the linear form output of such a design contains enough information to begin developing agent systems. The next section will explain how the information captured by the tool is used in the building of agent systems.

7.3 Transformation of UCM Design into Agent Systems

The behaviour signatures shown in the plug-ins are implemented in the prototype system as separate rule engines which communicate via the tuple space, as described in the paper. The end-to-end paths through a system that occur are based on the decisions the rule engines make in collaboration with each other. The actual prototype implementation uses a Java implementation of the CLIPS expert system as the rule engine for the implementation of agent logic and Java code to perform the actions of the agent. The agents communicate through the Micmac (an internal agent coordination environment of Mitel based on tuplespaces) blackboard with each other and with Java interface threads for MediaPath, a telephony simulator.

Figure 32: Agent Communication - Tuple Space (following [16])

Figure 32 shows an extract from the paper in which the sequence of tuple space operations on the Micmac blackboard is shown. The sequence shows how the feature interaction between the Call Forwarding and Originating Call Screening features are resolved. Agents communicate with each other by performing operations on the tuple space which may consist of either posting the desire to perform an action or commenting on a previous posting. By these mechanisms the feature for OCS is able to comment on the call forwarding feature's desire to forward the call to X which is on the OCS prohibited list. The Originating feature which is responsible for initiating calls responds to the com-

Application of Use Case Maps to System Design with Tool Support

ment by the OCS rule engine by withdrawing its request.

The competing rule engines which communicate by posting their intentions to the tuple space for conflict resolution thereby controlling the behaviour of the system relate to the designs present in the UCMs as each rule engine corresponds to a dynamic plug-in for a stub. Information from the UCM design in terms of preconditions and postconditions as well as the responsibilities along the path segments in the plug-ins is used to guide the development of the rule engines as will be explained in the next section.

It is not the purpose of this chapter to explain in detail the operating environment or implementation details of the agent prototypes but rather give an overview of the entire prototyping process from designs specified in terms of UCMs to the actual executing code. Specifically the process of transformation from the maps drawn in the tool, to the internal models of the agents (which will be described later) to the declarative CLIPS rules which form the agent behaviour will be discussed. An explanation of this process will show how the use of Use Case Maps with dynamically selectable behaviour signatures can not only be used to visualize complex inter-agent behaviour but to capture information for semi-automated implementation of the logic of those agents.

7.3.1 Agent Models

In order to understand how agent systems can be built with information from UCM design diagrams it is necessary to explain some of the theory of the functioning of agents and the models used in their construction. Only then can the role of UCM information in this design process be explained.

The agents in the prototype are based on the BDI model in which agents form decisions based on the beliefs they currently have, the desires to perform certain functions and their intentions to perform certain actions. Desires relate to goals of agents, beliefs relate to system conditions (such as run-time pre/postconditions), and intentions relate to a specific plug-in whose purpose is to execute actions to realize a certain goal. The set of all possible actions or plug-ins, before selection of one to execute, are called plans.

There are many models that can be used to describe agents. Two of these will be described in this section as they were used in the prototype implementation. The Agent Internal model describes the mental state of the agents. The Conversational model presents the communication and coordination among agents.

7.3.2 Agent Internal Model

The agent internal model describes agents in terms of their goals, beliefs, and tasks. Goals relate to activities the agents wish to accomplish such as sending a call request, or receiving a call request. Beliefs relate to the facts of which an agent is aware, which affect its current and future decisions. Tasks relate to all of the agent tasks that must be performed. These may refer to sequences of responsibilities in a UCM or to complex tasks which are in fact subgoals.

The following table, extracted from the paper, shows the agent internal model for the example in this case study. Much of the information in this table is extracted from the linear form outputted by the UCM Navigator and processed by the Java agent tools.

.Table 1: Agent Internal Model (following [16])

	Goal	Precondition	Postcondition	Task	Comment
1	Process originating call	Number is collected	Request sent to answerer	send_request	ORIGINATING
2	Process originating call	Outgoing call connection requested	Call permitted or rejected	check_list doPermit doReject	OCS
3	Process call request	There is an incoming call	Caller and/or answerer are notified	ring notify_caller	TERMINATING
4	Process call request	CF is on. There is an incoming call	Caller notified of a new destination	doForward	CF

The goal column lists the goals that the agent wishes to accomplish such as processing originating calls and terminations of calls, in this case. As there may be several methods of accomplishing goals, as with different plug-ins for a dynamic stub a goal may occupy many rows in the table. Each plug-in for the dynamic stubs in this example (Figure 28 to Figure 31) represents a plan in agent terms of how a goal may be accomplished and is implemented in the prototype as a separate CLIPS engine which competes with the CLIPS engines for the other plans.

The precondition column lists the beliefs (facts) that must hold for a goal to be executed. The postcondition column lists the effects of successful goal execution on an agent's beliefs. These are extracted from the linear form as the preconditions and postcon-

Application of Use Case Maps to System Design with Tool Support

ditions of the paths in the plug-in maps that represent the individual features (which are seen as alternate plans to accomplish the same goal) as can be seen from (Figure 28 to Figure 31) .

The task column lists the tasks that may be performed to accomplish the goal. These, in this case, are simply the responsibilities that exist in the plug-ins. However as was mentioned earlier these tasks may in general contain complex tasks represented by subgoals, each with their own set of plans for accomplishing that goal. Each plan would then appear as rows in this table with the goal in the goal column being the subgoal and all other information extracted from the plug-in maps for the dynamic stub representing the subgoal.

The following illustration is an extract from the paper which illustrates the relationships between agent concepts such as goals, beliefs and tasks to the concepts of UCMs namely dynamic stubs, pre/postconditions, responsibilities and static stubs.

Figure 33: Mappings between UCM-agent concepts (following [16])

Path segments which traverse a component, specifically an agent, represent a goal that the agent need accomplish. Stubs are interpreted differently depending on whether they are static or dynamic. Static stubs are interpreted simply as sets of tasks for the agent to perform (a simple decomposition view of stubbing) while dynamic stubs are interpreted as subgoals where each plug-in map which provides alternate behaviour signatures for the stub is seen as alternative plans to implement that goal. The set of pre/postconditions for the plug-ins help in forming the belief set for the plan, which constitutes the set of facts that must hold true for the agent to select that particular plan at run-time. Responsibilities as stated previously are simply seen as high level tasks for agents to perform.

Static stubs as stated previously are simply sets of these tasks. However as complex tasks may be represented by subgoals and hence by dynamic stubs static stubs which are simply used for decomposition may contain dynamic stubs to represent subgoals along the paths.

7.3.3 Conversational Model

The agent internal model described in the previous section described how the mental state of the agents was described with the help of information extracted from UCMs. However the internal model does not specify how agents communicate with each other. The conversational model is used to determine the messages agents send to each other. As with the internal model information from UCMs is used to formulate the sequences of messages.

The visualization of the transfers of control that are present in UCM diagrams suggest the messages that need to be sent. For example in figure 28: Originating - Default Plug-in for CSP the simple plug-in for the Originating feature includes the responsibility **snd-req** which sends the request to the answer side agent. This is accomplished in this prototyping environment by the call side agent placing a proposal for connection in the tuple space so that other features can comment on it. As the precondition for the originating path is simply that a number is collected this feature does not need to wait for messages from other agents (i.e. wait on the tuple space with a blocking query operation) to begin. It simply needs to place its proposal for connection in the tuple space.

Similarly in figure 31: CF: Call Forwarding for ASP the simple plug-in for the call forwarding feature contains the responsibility **fwd-req** which simply forwards the request to the new answerer agent but the plug-in map itself gives no indication of how to accomplish this feat. The path structures in the top level map suggest the appropriate communication. As can be seen from figure 27: Root Map - Telephony Feature Interaction Example the only path from one answerer side agent to another is the feedback path back to the call side agent which can then forward the request to the new answerer side agent. This is also how the feature interaction between the originating call screening feature and the call forwarding feature is resolved. If the number to which the call should be forwarded is on the originating screening list it will be detected when the call side agent attempts to connect to the new number. The pre- and postconditions for the call forwarding feature plug-in map also suggest appropriate communication. The preconditions are that the CF feature is on (

Application of Use Case Maps to System Design with Tool Support

which simply results in the CF plug-in being selected at run-time instead of the default terminating plug-in) and that there is an incoming call. The postcondition of this plug-in is that the caller is notified of a new destination. As the plug-in scenario can only begin when a call connection is proposed by a call side agent and the effect of the plug-in is that this same call side agent is notified of the new number appropriate communication between the two agents can be inferred to be waiting for a call connection proposal and then replying to that same agent which sent the proposal with a forwarding request. This is implemented in the tuple space environment by first performing a blocking query on call connection proposals to the answering agents number and then responding to the proposal by placing a counter proposal in the tuple space to propose forwarding the call to the new number. This counter proposal is intended at the originating agent which reads the counter proposal from the tuple space and proceeds accordingly with the new information in hand.

The following table is an extract from the paper which shows the inter-agent conversational model for the telephony example in this case study. There are four types of messages in the conversational model, proposal (Prop), counter proposal (CProp), ACCEPT, and REJECT. These four message types implement a general agent negotiation protocol. The messages in the conversational model are used to implement the tuple space operations displayed in figure 32: Agent Communication - Tuple Space (following [16]).

Table 2: Conversational Model (following [16])

	Received	Sent	Comment
1		Prop(:connectFrom a :connectTo b)	ORIGINATING
2	Prop(:connectFrom a :connectTo b)	ACCEPT REJECT	OCS
3	Prop(:connectFrom a :connectTo b)	ACCEPT REJECT	TERMINATING
4	Prop(:connectFrom a :connectTo b)	CProp(:connectFrom a :connectTo f)	CF
5	CProp(:connectFrom a :connectTo f)	Prop(:connectFrom a :connectTo f)	ORIGINATING

In summary the conversation necessary between agents is partially determined by the path structure of the UCMs describing the system and partially by the pre/postconditions in these UCMs (which are part of the agent internal model). The use of UCMs to model the behaviour of agent systems provides a visual framework to view desired end-to-end behaviour which suggests appropriate communication pathways. As UCMs are not a detailed methodology the transformation from the maps to the agent internal and conversational models to the declarative rules (in this case CLIPS) for the expert system that

performs the reasoning for the agents is not linear and needs much human input. However the general behaviour structure provided by the use of UCMs provides a view of operation at a high level of abstraction which can then be expanded by designer input.

7.4 Summary of Agent Case Study

This case study has shown how the use of UCMs specifically with the feature of dynamic stubs with dynamically pluggable behaviour signatures can be used both to visualize the interaction among agents in a system and to provide design information for a semi-automated agent prototyping process in which the behaviour and communication signatures of the agents in the UCMs are translated to logic rules in an expert system. If the entities representing the individual agents are given a communication medium with which they can negotiate such as the Micmac blackboard utilizing a tuple space that is described in the paper there is a path from designs specified in UCM terms to executable implementations which are far superior in many respects, specifically the ability to dynamically resolve conflicts between features, than standard communication software implementations.

As UCMs provide visualizations of high level behaviour but not detailed descriptions of behaviour the process requires much designer input to fill in details. The steps in the process are:

- Utilize Use Case Maps for the high level design of agent systems as a means of discovering agent behaviour. Use the UCM concepts of dynamic stubs with differing plug-in maps as a means of describing alternate behaviour signatures which may be chosen at run-time.
- Use back end tools that parse the linear form generated by the tool to help construct, with designer input, a simple internal model for the agents from the pre/post-conditions, responsibilities and plug-in maps in the UCM design
- Derive a conversational model for communication between the agents from the coordination signatures in the UCMs, and partly from the designer's input.
- Utilize both the internal model and the conversational model to implement independent expert system (in this case CLIPS) engines where each agent plan which corresponds to a plug-in for a dynamic stub is implemented as a separate engine. The appropriate inter-agent communication messages are inferred by designers from the

conversational model.

7.5 Case Study B - Performance Prediction Project

The second case study of this chapter is a simple example demonstrating the use of UCMs for performance prediction. As the Performance project is only at a preliminary proof of concept stage there have been no attempts to use the UCM Navigator to predict the performance of meaningful systems. Simple examples have been used to test the process of simulating a system from a UCM using the performance annotations that have been added to the UCM Navigator and which were described in chapter 6.

The example described in this section is of a top secret video conference system that enables users to participate in encrypted video conferences. Each location transmits a local audio/video signal which is compressed and encrypted as well as receives signals from the other conference participants. The remote A/V streams are decrypted and uncompressed and all video signals, including the local signal are displayed on the screen. All audio signals are mixed and played on the local speaker. In addition each station has a console at which users can add and remove participants and adjust the audio volume of each participant.

Figure 34 shows the root map for the system. There are three main system activities described in this map, the processing of local output from the local video camera and microphone, the processing and display of A/V streams from remote conference participants and the processing of console input to add/remove participants or adjust volumes. The paths starting with SaudHW and SvidHW are started when audio and video frames arrive and have responsibilities for merging A/V, compression, encryption and file storage.

The processing of remote streams is shown by the series of paths which are synchronized in the AudioP process which synchronizes the audio frames of all sources for play on the sound card. The processing of these streams is done in the ProcN stubs which are assigned to separate hardware processors. Although the processing is similar in all of these paths separate paths were needed as combining all of these paths into a single one requires capabilities not yet existing in the tool. These specifically are the specification of separate processors for processes in different instances of the same stub and support for logical path synchronization.

The processing of keyboard input is done by the stack of processes labelled

Application of Use Case Maps to System Design with Tool Support

Process 7.5. A stack is used as the processing of remote streams is done by separate processors and the binding of responsibilities to the hardware processor on which they execute is performed through the enclosing component of the responsibility. As the console functions relate to the processing of remote streams it is required to have the operator functions of add/remove participant and adjust volume execute on the appropriate processor.

7.5.1 Specification of performance annotations for root map

In order to perform a performance simulation on a design specified in terms of UCMs performance annotations need to be added to the responsibilities, paths, and components in the design. These annotations include:

- Specification of the resource functions for the responsibilities which specify the sequence of hardware resource usage by the responsibility
- Specification of timestamp points along paths and response time requirements between these timestamp points
- Specification of the layered data stores accessed by the responsibility needed for mutual exclusion considerations in the simulation
- Specification of the time distributions of the arrival events down paths
- Specification of the choices made at or forks
- Specification of the hardware processor on which a component is run

This section will describe these annotations for the top level map. The following section will describe the same annotations for the sole plug-in map for this design. In addition to the above annotations which need to be made to each responsibility, path starting point, or fork, and component in the design there are several specifications which are global to a design. These include the specification of response time requirements, of hardware device characteristics and of layered data stores. These will be given in a later section.

Figure 34: Video Conference System UCM

Figure 35: Resource Functions for A/V functions in Main Map

Figure 35 shows the responsibility dialogs which show the execution sequences describing the resource functions for the audio/video manipulation responsibilities in the top level map, that is the responsibilities relating to processing local and remote video.

Figure 36 shows the resource functions for the responsibilities relating to keyboard processing to add/remove participants and adjust volumes. The responsibility AdjustVolume in addition to using a small amount of processing time also accesses a layered data store in which the volume levels are stored. The specification of access to layered data

stores is important for mutual exclusion considerations in the simulation.

Figure 36: Resource Functions for Keyboard Processing

The resource functions are currently specified in a rather cryptic manner which

corresponds to the Objective-C methods that are used to actually simulate the usage of resources. This can be simplified in the future.

The phrase a puServDist: [Det : 38] signifies the use of processor resources with a service time distribution of deterministic with a value of 38 microseconds. Similarly the phrase a resource: \$Disk3 servDist: [Det : 2] relates to hardware resource usage, in this case, the use of a hard disk labelled Disk3 (specified using the hardware device dialog box in the performance menu) with a deterministic service usage of 2 disk accesses.

Figure 37: Specification of Arrival Event Time Distributions

Figure 37 specifies the time distributions of the arrival of starting events for scenarios down paths. This must be specified as the simulation environment would need to generate starting events for scenarios.

As there is only one or fork in the top level map figure 38 specifies the branch selection characteristics of the or fork following the parse responsibility where different branches are taken depending on the user input. The specification of selection characteristics is needed to simulate the choice of branches. In this case the characteristics are specified as relative frequencies but in general may just as well be percentages or logical functions which depend on a global variable such as a layered data store. As the output

branches of an or fork are currently not labelled the different branches are identified by the first responsibility along them.

Figure 38: Specification of Branch Choices

The specification of hardware processors for software components is shown in figure 39. This is necessary as in general it may be desired to simulate the performance of a multiprocessor system and it would be necessary to know the hardware device on which the responsibilities inside a software component consume processor cycles. That is definitely the case with the example in this case study as the video compression/decompression and encryption/decryption require heavy processing power and are implemented on a multiprocessor system.

Figure 39: Specification of component to processor allocation

The specification of the hardware processor on which each component runs is made in the component attributes dialog box where there exists a field Processor Name which is valid and enabled for certain component types (Process, Team)

Component Name	Hardware Processor
Process6	PU6
Store6P	PU6
MultiM6P	PU7
DisplayP	PU7
Process7.5	PU7
AudioP	PU7
Remote	PU0

7.5.2 Specification of performance annotations for plug-in map

This section will specify for the plug-in map all of the performance annotations specified in the previous section for the root level map.

Figure 40 shows the sole plug-in for the example which displays the processing that must be done for each remote video stream including decryption, decompression and display. It involves the same functions as for the local stream, except in reverse order. The decryption and storage processes in each of these stubs are each bound to a different hardware processor as they are computationally intensive.

Figure 40: Remote Stream Processing Stub

Figure 41: Resource Functions and Layered Data Store Access

The resource functions for the Decrypt and Uncompress responsibilities simply use large amounts of processing time with fixed service times of 11 000 and 28 000 microseconds respectively. The store to file responsibility uses a small amount of processor time (38 microseconds and accesses one of the three disks (the disk varies for the five plug-in maps) for two disk accesses

The resource functions for the Mix, SeparateAfromV, and Display responsibilities use processor time of 380, 38, and 841 microseconds respectively. The Mix responsibility additionally accesses both of the layered data stores that are defined in this design (as visible in the dialog at bottom left, which is invoked from the Mix responsibility dialog) the AudioAccumulator with mode RW read/write and VolumeArray in the read only mode RO.

Application of Use Case Maps to System Design with Tool Support

The resource functions and access to layered data stores by this submap are shown in figure 41. The heavy use of processor time for the decryption and uncompression responsibilities leads to them being assigned to different hardware processors for the five versions of the plug-in map. The Mix responsibility which performs the mixing of the five audio channels accesses the global data stores for audio output.

As mentioned previously the plug-in map shown is duplicated five times once for each stub Proc1-Proc5 in the root map. This is done as the tool currently does not support the concept of variation of a plug-in map's parameters, such as the processor name attribute for a component, with different instantiations of the plug-in map. The map shown in figure 40 is duplicated five times for the stubs Proc1-Proc5 the only differences being that the decryption/uncompression process and the storage process are differentially labelled ProcessN and StoreNP where N is the number of the parent stub. These two processes are also bound to the hardware processor of the same number, that is for example stub Proc2 in the root map has processes labelled Process3 and Store3P which are bound to processor PU3. The other two processes the audio and display processes AudioP and DisplayP are labelled identically in all five plug-in maps are all allocated to processor PU7, which handles the audio and video display in this design.

In terms of time distributions for arrival processes there is only one path start point in the plug-in maps, labelled FromRoot, which does not need a scenario starting process as its stimulus comes from the top level map. There is no or fork in the plug-in maps and so there are no branch selection characteristics in any of the plug-in maps.

7.5.3 Global Specifications to UCM design

As stated previously there are several types of performance specifications that are global to a design represented with UCMs rather than annotations to a specific responsibility, path start or choice point, or component. These include the following:

- The set of response time requirements for a design
- The specification of the characteristics of hardware devices
- The specification of layered data stores

Response time requirements are designer specified performance requirements that state the desired time limit for processing that occurs between two timestamp points which are located along paths. They are used to determine if the specified system can meet its

Application of Use Case Maps to System Design with Tool Support

performance requirements as the simulation environment is capable of reporting whether or not these requirements were met during the simulation run. Timestamp points are placed along use case paths to delimit intervals of paths where it is desired to measure the execution time along this interval. They are created by selecting a pair of timestamp points, invoking the Create Response Time requirement transformation and entering the appropriate information into the resulting dialog box which includes the desired response time in microseconds, the percentage of time that this requirement must be met, as well as a name for the requirement.

Figure 42 shows the collection of response time requirements for this design. Once a response time requirement is created as described above it appears in this list of requirements for the entire design where it can be viewed, edited or even deleted. Requirements are global to a design rather than being local to a map as response time requirements may cross map boundaries. That is they may start in a higher level map and end in a lower level plug-in map. There are requirements for all major processing functions such as video processing, audio synchronization as well as for console operations.

Figure 42: TSVC Response Time Requirements

Name	Start	End	Response Time	Percentage
IncomingAud1	T1.1	T7	250000	95
AdjustVol	T8	T8	500000	95
AddP	T8	T10	2000000	95
RemPart	T8	T11	2000000	95
OutgoingM4	T3	T4	150000	95
LocalVid	T3	T12	100000	95
IncomingAud5	T15	T7	250000	95
IncomingAud2	T12	T7	250000	95
IncomingAud3	T13	T7	250000	95
IncomingAud4	T14	T7	250000	95
R1.3-2.3	T13	T2.3	250000	95
R1.2-2.2	T12	T2.2	250000	95
R1.1-2.1	T1.1	T2.1	250000	95
R1.4-2.4	T1.4	T2.4	250000	95
R1.5-2.5	T1.5	T2.5	250000	95

Selected Response Time Requirement

Response Time (us): Percentage:

Name:

Accept Cancel Delete

Remove Dialog

The specification of hardware devices includes devices such as processors or hard disks so that a simulation environment can gauge the amount of time that would pass for a given processor operation or disk access. The Device Characteristics dialog invoked from the performance menu allows designers to specify the names and characteristics such as speeds and access times of the hardware devices in a particular execution environment. Responsibilities specify the hardware devices on which they execute by a number of means. As shown certain component types (process, team) can specify the hardware component on which they execute which binds all responsibilities bound to that component to its hardware processor. Access to other hardware devices such as hard disks is specified in the Execution Sequence field which specifies the resource function of the responsibility.

The hardware device characteristics for the two basic device types specified in this example (processor, disk) are shown in figure 43. The eight processors are given the simple characteristic 1 which specifies that they are all of equal speed. In general there may be different speeds for different processors as well as different integer and floating point speeds for a single processor. The three hard disks also are given a very simple characteristic. In general the speed of a disk may be specified in actual values for seek time, block

transfer rate and any other characteristics of which the simulation environment can make use.

Figure 43: Hardware Device Characteristics

The specification of the layered data stores is shown in figure 44. There are only two data stores one for the volume array which stores the user set volumes of the individual conference participants, and one for the audio accumulator used when mixing audio input. Two access modes of read only and read/write are also specified. As shown previously responsibilities specify which layered data stores they access through a separate dialog box invoked from the responsibility dialog. This information is then used by the simulation environment as accessing global data stores would have performance implications in an actual executing system.

Figure 44: Layered Data Store Characteristics

7.6 Process of System Simulation from UCM designs

The previous sections described the annotations that need to be made to a design specified with the UCM Navigator so that performance simulations could be performed. This consisted of the inclusion of timestamp points along paths, the specification of resource functions, executing processors, and access to global layered data stores for responsibilities, the specification of scenario starting and branching for use case paths as well as information that is global to a design such as response time requirements, hardware device characteristics and global data stores.

This section will explain how a use case map design specified with the above information is simulated. The simulation results of the example specified in detail in this case study will be given.

Design information is currently passed from the UCM Navigator to the simulation environment through the tool's file format. It will eventually be done through the linear form when the linear form has been expanded to include performance information. The simulation environment prompts the user for the name of a UCM file to simulate as well as

Application of Use Case Maps to System Design with Tool Support

a random number seed and the length of the simulation. The simulation environment then reads in the file using the file loading code from the UCM Navigator, creating the appropriate objects. These objects are then traversed and scanned to create the appropriate ASN objects upon which the simulation environment is based as described in Section 6.5: Linking of UCM Navigator to Simulation Environment. The environment then performs a Parasol simulation based on the ASN structures and outputs the results.

7.6.1 Simulation Results

The following is the output of the simulation run for the example in this case study. As can be seen the name of the ucm file is given as well as information required by Parasol, namely a random number seed and a simulation duration. The results consist of two basic statements, the utilization of hardware devices, and the results for each response time requirement. The utilization of processors is specified currently with a Parasol task number which corresponds to the Process components in the UCM design. The mapping between these numbers and the component names can be seen from the following lines which are extracts of the tracing done during the simulation. The complete tracing was not included for reasons of length. For each response time requirement specified with the tool there is the actual execution times with means and confidence intervals, as well as output specifying whether the requirements were met.

Sample tracing output:

Time: 1.59025E+06; Node: 9; Task 9 (AudioP) receives message 2075 on port 72.

Time: 1.59067E+06; Node: 9; Task 21 (DisplayP) executing.

Time: 1.589E+06; Node: 14; Task 29 (Store1P) ready.

Time: 1.589E+06; Node: 11; Task 24 (Process5) sending message 2072 to task 14 via port 87.

Simulation output:

>ucmnavPF

```
*****
**
*                                     *
*           P A R A S O L (Version 3.1)           *
*                                     *
*****
**
```

Application of Use Case Maps to System Design with Tool Support

Enter random number seed: 2

Enter positive simulation duration: 600000000

Enter name of .ucm file to simulate: tsvc4e.ucm

Blocked simulation statistics for time = 6E+08.

Name	Mean	95% Interval	99% Interval
Disk (cpu 0) Utilization	0.719904	0.000259	0.00043
Disk (cpu 0) Utilization	0.719934	0.000176	0.000292
Disk (cpu 0) Utilization	0.719934	0.000175	0.000291
overall-within-req't	1	0	0
IncomingAud.1 time	41548.464944	0.097315	0.161397
IncomingAud.1 within req't	1	0	0
AdjustVol time	76	0	0
AdjustVol within req't	1	0	0
AddP time	351.6	597.700972	991.287923
AddP within req't	0.4	0.679978	1.127745
RemPart time	351.6	597.700972	991.287923
RemPart within req't	0.4	0.679978	1.127745
OutgoingMM time	38878	0	0
OutgoingMM within req't	1	0	0
LocalVid time	840	0	0
LocalVid within req't	1	0	0
IncomingAud.5 time	41548.464944	0.097315	0.161397
IncomingAud.5 within req't	1	0	0
IncomingAud.2 time	41548.464944	0.097315	0.161397
IncomingAud.2 within req't	1	0	0
IncomingAud.3 time	41548.464944	0.097315	0.161397
IncomingAud.3 within req't	1	0	0
IncomingAud.4 time	41548.464944	0.097315	0.161397
IncomingAud.4 within req't	1	0	0
R1.3-2.3 time	39836	0	0
R1.3-2.3 within req't	1	0	0
R1.2-2.2 time	41532.036432	0.101133	0.167729
R1.2-2.2 within req't	1	0	0
R1.1-2.1 time	39835.965152	0.096739	0.160442
R1.1-2.1 within req't	1	0	0
R1.4-2.4 time	39038	0	0
R1.4-2.4 within req't	1	0	0
R1.5-2.5 time	39038	0	0
R1.5-2.5 within req't	1	0	0

Application of Use Case Maps to System Design with Tool Support

PU0 task 2 Utilization	5.10004E-06	6.62166E-06	1.0982E-05
PU3 task 5 Utilization	0.779941	0.000157	0.000261
PU7 task 8 Utilization	0.016819	3.89105E-06	6.45332E-06
PU6 task 22 Utilization	0.759943	0.000158	0.000262
PU5 task 24 Utilization	0.00076	1.75814E-07	2.91589E-07
PU4 task 26 Utilization	0.00076	1.75814E-07	2.91589E-07
PU1 task 29 Utilization	0.00076	1.75814E-07	2.91589E-07
PU2 task 32 Utilization	0.779941	0.000157	0.000261
PU2 task 33 Utilization	0.00076	1.75814E-07	2.91589E-07
PU1 task 30 Utilization	0.779941	0.000157	0.000261
PU4 task 27 Utilization	0.779941	0.000157	0.000261
PU5 task 25 Utilization	0.779941	0.000157	0.000261
PU6 task 23 Utilization	0.00076	1.75814E-07	2.91589E-07
PU7 task 9 Utilization	0.016819	3.89105E-06	6.45332E-06
PU7 task 10 Utilization	0.016819	3.89105E-06	6.45332E-06
PU7 task 11 Utilization	0.008359	1.93396E-06	3.20748E-06
PU7 task 12 Utilization	0.016819	3.89105E-06	6.45332E-06
PU7 task 13 Utilization	0.008359	1.93396E-06	3.20748E-06
PU7 task 14 Utilization	2.02668E-06	1.81438E-06	3.00916E-06
PU7 task 15 Utilization	0.008359	1.93396E-06	3.20748E-06
PU7 task 16 Utilization	0.008359	1.93396E-06	3.20748E-06
PU7 task 17 Utilization	0.00076	1.75814E-07	2.91589E-07
PU7 task 18 Utilization	0.016799	3.88643E-06	6.44565E-06
PU7 task 19 Utilization	0.008359	1.93396E-06	3.20748E-06
PU7 task 20 Utilization	0.00076	1.75814E-07	2.91589E-07
PU7 task 21 Utilization	0.016819	3.89105E-06	6.45332E-06
PU3 task 6 Utilization	0.00076	1.75814E-07	2.91589E-07

Simulation Complete

7.7 Summary of Performance Prediction Case Study

This case study explained the process of annotating a system design specified with the UCM Navigator so that performance simulations could be made. Full detail was given of the performance annotations and the reasons for those annotations. The purpose of this study was to show the relative ease for which fairly complex designs could be simulated. The use of the UCM Navigator as a front end for performance simulations both gives an added dimension to the use of UCMs for high level system designs as well as providing a graphical interface for simulation environments the design to be simulated is clearly visi-

Application of Use Case Maps to System Design with Tool Support

ble as opposed to the case where hand edited textual input files are used as inputs to simulation environments.

8. Discussion

8.1 Introduction

The previous chapters described the requirements, design, and implementation of the UCM Navigator, a prototype tool for the high level design of systems using UCMs. This chapter will explore issues related to the tool such as the ease of use of its interface, the performance of its display, its applicability to large designs, and possible extensions which would make the tool work with other applications and design models. Many of these issues can be handled simply with more implementation effort, such as display performance while others require more investigation such as improvements to the user interface and complex visualization techniques for large designs.

8.2 User Interface Issues

As one of the purposes of the UCM Navigator is to provide a tool by which designers can analyze and design systems at a high level the user friendliness and intuitiveness of its interface are important issues. A tool that is meant to edit Use Case Maps should allow users to do so in a manner that is natural and intuitive.

The user interface of the UCM Navigator began by imitating that of the C++ prototype in that both had the same point-and-click interface for drawing paths and the same bindings of path elements to components. The major difference arises in that as this editor is built on hypergraphs and the concept of hypergraph transformations as a means of editing paths the contents of the popup transformation menus varies with the element or elements selected. The interface has been quite well received. Feedback has been obtained as to improvements in the interface as well as additional desired editing operations. This section will describe many possible improvements to the interface.

8.2.1 Addition of UCM Notation and Desired Features

The UCM Navigator allows users to use most elements of UCM notation. Some elements have not yet been added. These include the following, relatively minor additions:

- The ability to trigger failure paths through the use of the failure notation to link a

Application of Use Case Maps to System Design with Tool Support

RAISE responsibility on a main path to a HANDLE responsibility on a failure path. Also the ability to specify timeout paths for timed waiting places is necessary as these are a form of failure path.

- The cardinality of synchronizations such as 2:2 for actual path concurrency or 1:N for logical concurrency. The tool needs to recognize and record path compositions which may have one path in and one path out physically (or vice versa) but may logically have numerous identical paths.
- Instance values for plug-ins that include formal components. The plug-in map itself would include formal components which could be instantiated by means of user given instance information attached to the plug-in binding.
- The ability to specify path continuity across stubs. The tool would collect information as to the fixed mappings of input paths to output paths of stubs
- The ability to specify element types for pools which would be either component, with a component type or plug-in with a list of references
- The addition of a browser to scan the map hierarchy of an entire design
- The ability to orient paths in any direction, this involves making or/and forks/joins orientable in the four main directions
- The ability to produce various outputs such as MIF or HTML files as well as various options for the postscript output

Another desirable feature would be the ability to validate the completeness of the specification of a design. For a design without performance annotations this would mean verifying whether all path starts and ends as well as other elements(waits, timers, stubs) have all of the appropriate conditions(pre/post) and events(triggering, resulting) specified, whether all paths segments have appropriate scenario labels and whether all plug-in maps are bound to their parent stubs. For designs with performance annotations it would also mean verifying whether all timestamp points are a part of a response time relationship, whether all active components have a processor specified, whether all or forks have branch choices specified, and whether all responsibilities have execution sequences defined.

8.2.2 Advanced Editing Features

The UCM Navigator currently supports all basic editing features that are necessary to draw UCMs into multilevel hierarchies through the use of stubs. However the editing transformations that were created were a minimal set in order to create a working editor in the time available. Many more operations which would be quite useful in the manipulations of large designs can be performed. There is currently no support for undoing operations, or for copying sections of paths between maps. This last feature would be particularly useful in moving sections of paths between a top level map and a stub such as when a designer would want to move path sections from a map into newly created stubs along the paths of the map as the map grows in complexity. Other advanced editing features include the ability to rotate and scale sections of maps. These modifications range in complexity from relatively minor, for copying sections of paths to major for undo operations and rotating, scaling of maps as complex data structures would need to be transformed for these operations.

These editing features would make the tool a more powerful one for designers to use as the ease of manipulations of large designs would be increased. Any improvements which make the tool more powerful in its ability to manipulate large designs enhance the usability of the UCM methodology in the design of large systems.

8.2.3 Manipulation of Display Properties

The UCM Navigator currently has a fixed size map editing area. Useful extensions to the tool are to make this area scrollable, resizeable and possibly zoomable. A scrollable editing area would allow the use of virtual editing screens which are much larger than the available display area. The visible editing screen would be simply the section of the virtual area selected by the scrollbars which can fit within a given display and would resize as the UCM Navigator window is resized. The ability to zoom the display, that is change the scale of the display would allow small map detail to be easily viewed. A larger editing area would allow much larger maps to be created and manipulated. Similarly maps could be reduced in scale to allow more information to fit into available screen space. This would greatly extend the usefulness of the tool. The complexity ranges from relatively minor for implementing a scrollable and resizeable drawing area to major for implementing the zooming functionality, as complex display transformations would need to be performed.

8.2.4 Enhanced visibility of transformations

One of the difficulties in using the tool is that the possible transformations are all hidden in popup menus which must be explicitly invoked by the user. New users are not sure of how to perform an operation such as creating an and join or merging two paths. It is possible to change this by augmenting the operation of the menus with a row of graphical buttons corresponding to all of the possible transformations. These controls could be used to perform transformations in place of the popup menus. There would be one control for each possible transformation, each with a graphical image denoting its operation. They would be visible at all times but would be either active or inactive and grayed-out depending on the element/s selected. This would be a relatively minor modification.

8.3 Design Scalability Issues

One of the design principles of the UCM Navigator has been to support the design of complex multilevel maps to any degree of complexity. The provision of support for stubbing with plug-in maps for the contents of the stub provides this capability. However other scalability issues remain.

One of these issues is the cross-referencing between different map levels as an end-to-end scenario is followed through the system. Lower levels of path detail can be viewed by progressively expanding stubs and viewing the plug-in maps. However at some point the user loses context and the behaviour flow through the system becomes less clear. One possible solution to this, the use of fisheye transformations, will be explained in the next section. Other simpler solutions are the ability to display multiple maps on screen simultaneously or have multiple editing windows along with advanced cross-referencing between levels. This cross-referencing may take the form of the ability to view stub binding information on screen, that is the input and output paths of a stub will be marked with the labels of the corresponding path start and end points in the plug-in map (for static stubs) and the path starts and ends in the plug-in map will be marked with the labels of the corresponding paths in the higher level map. Currently stub binding information appears only in the generated linear form. The ability to display multiple maps with cross-referencing information is a relatively major modification as the tool would need to work with multiple display and editing windows.

Other means of expanding support for large designs include support for layering in UCMs, support for the use of standard behaviour patterns, the ability to support multiple root maps in a design and to selectively view paths and components through the use of transparency overlays.

8.3.1 Fisheye Viewing of Multilevel Maps

The UCM Navigator supports the design of multi-level maps through the concept of stubbing and the ability to specify the plug-ins for stubs as separate maps. Multi-level maps to any level of decomposition could be constructed. However with increased decomposition comes a loss of context as the user loses context for the position of the behaviour described in the stub plug-in in terms of the whole design. A solution to this is to allow multiple views to be displayed on screen through multiple editing windows. For example one editing window would show a top level map with a stub highlighted. A second window could show the plug-in map for that stub that was selected by the user (in the case of a dynamic stub). A stub in the second window could be highlighted and its chosen plug-in map displayed in a third editing window. This expansion could be extended to any number of levels.

However the ability to view the elements of a decomposition does not necessarily provide the visual context that is desired, it only allows the elements of the decomposition to be shown simultaneously. It does not reduce the amount of screen space needed to show the context as ideally all levels should be displayed simultaneously to provide context.

Previous work on visualization of large software systems has been performed in a M.Sc. thesis by Charles Cui in the area of layered fisheye view techniques [17]. This thesis involved building a tool which uses nonlinear fisheye viewing techniques to view several layers of a software design concurrently. This tool assumes that a software design is specified as a hierarchically structured set of components where the macroscopic behaviour signatures are shown by timethreads (the previous incarnation of Use Case Maps). The display would initially show the top level of a component structure with timethreads traversing the components. Internal components of those present in the top level would exist in the design but be hidden in the display as would the paths that cross the hidden components. A component could be selected and expanded. This would result in a nonlinear transformation being performed on the top level diagram so that both the top level diagram

Application of Use Case Maps to System Design with Tool Support

and the expanded view of the component showing previously hidden internal components and paths would be visible on the screen simultaneously. A central section of the screen would be assigned to the view of the expanded component. This view would be nondistorted. The view of the top level diagram of which the component is part would still be visible on screen but it would be nonlinearly transformed so that it would fit in the area surrounding the central section describing the expanded component. Such an arrangement would make good use of limited screen space while preserving contextual information. This technique could be expanded recursively in that one of the internal components of the expanded component could be itself expanded. The display on screen would then consist of a central nondistorted section displaying this component's internals with the contents of its enclosing component and the top level diagram both being transformed to be compressed to fit into the remaining area of the display. These expansions could proceed indefinitely up to the maximum decomposition level of a design however the display of the higher levels would be compressed to such an extent as to be unintelligible.

This technique could be applied to the UCM Navigator to be able to simultaneously view a higher level map including a stub and the chosen plug-in map for that stub. The major difference between the use of the fisheye transformation using UCM's and using hierarchically structured components would be that the unit of decomposition in the UCM methodology is a plug-in map whose internal paths are not limited to remaining inside the boundaries of its source stub. Paths in a plug-in may cross components that are external to the source stub, including components in another part of the design as well as components that are anchored in the main map. This may result in confusing displays on screen as the user may interpret components inside a plug-in as being hierarchically contained inside the stub. Some adequate means of differentiating contained components from external components must be given and users must be familiar with the concepts of UCMs in any event. Fisheye displays of UCMs will not be as intuitive as those of hierarchical component structures as the concept of expanding components recursively is far more visually intuitive.

When constructing fisheye displays of hierarchical components the boundary between the higher level diagram and the display of the expanded component is simple. It is the boundary of the expanded component. The boundary between a stub in a higher level map and the display of its plug-in map is not so simple. Some distinctive boundary would need to be created (such as thick dashed lines or a series of thinner dashed lines,

Application of Use Case Maps to System Design with Tool Support

possibly in a different color) to create a recognizable boundary for the expansion of a plug-in map. Path continuity would be shown across this expansion boundary by having the input and output paths to the stub be connected to their corresponding path start and end points in the plug-in map.

The fact that such transformations could be performed is not in doubt. Multilevel UCM designs are stored in the UCM Navigator as separate Map objects for each top level map and plug-in map. Map objects contain the entire structure of maps including the graphical Figure, Path, and Component objects needed to draw the map on screen as well as the hypergraph describing the path structure. Information from several Map objects could be used to construct fisheye displays of UCM decomposition on screen in which the higher level map would be distorted and the expansion of a stub, a plug-in map, would be nondistorted and occupy the central region of the screen. The one restriction that should be made is that only nondistorted maps should be editable as it would be difficult to perform transformations on visually distorted maps. As the purpose of fisheye transformations is simply the simultaneous viewing of multiple levels of decomposition this restriction is not too constraining. The implementation of fisheye viewing would be a major modification to the tool as first of all a proper method of displaying the many levels across stub boundaries would have to be determined and secondly a complex graphical transformation would need to be applied to the tool in a manner that would still leave transformed maps as visually understandable.

8.3.2 Multiple Root Maps and Selective Viewing of Paths

Another scalability issue is the ability for a design in the UCM Navigator to have many root maps and to have the ability to selectively view paths and their associated components. Currently the tool allows multi-level designs to be used to specify end to end scenarios through the use of a single root map with many plug-in maps to describe the lower levels. However as designs become complex the possibility of the same plug-in maps appearing in many different scenarios is likely leading to the desirability of having designs in the tool to have many root maps so the network of plug-in maps can be reused.

The ability to show many different end to end scenarios in a single design can be accomplished in many different ways. The first is to allow users to draw different maps and specify them as being root maps. This is not a major extension as the tool currently

Application of Use Case Maps to System Design with Tool Support

allows an unlimited number of different maps to be drawn, it is just that only one of these is interpreted as the root map of a design and the rest are interpreted as plug-ins. Support simply needs to be added to allow many different maps to be recognized as root maps allowing users to display their root map of choice and begin the navigation of the scenario through its many plug-ins. The ability to support multiple root maps is a minor modification.

Another way to accomplish the creation of many different end to end scenarios is to recognize that different scenarios may use the same component context diagram and provide a way of overlaying many set of paths over the same components. The tool of course allows multiple paths to be drawn over the same components in the same map however they are all displayed simultaneously which may result in very cluttered maps. What is needed is the ability to create transparency overlays in which the user has the option of selecting which paths or sets of paths to display on a map. Paths which are not selected will not be displayed although they will remain in the data structures for the map. Groups of paths could be selected by the user based on their path labels and named. These would correspond to the relevant paths necessary to describe a particular use case. This named grouping of paths would form a transparency overlay which could be overlaid over other such overlays on a map so that one or many different scenarios could be viewed simultaneously as the user chooses. There would be many possible views of paths for a given map depending on which transparency overlays were selected.

It is also possible to selectively display components in that components which are crossed by paths which were selected by the user to be displayed will be displayed and those that are not crossed by visible paths will be hidden although they will remain in the map's data structures as would hidden paths. Alternatively groups of components from the Component Context Diagram of a system could be selected and formed into named groups so that only the desired set of components out of the set of all background components in a map would be visible. This would further reduce visual clutter and display on screen simply what is required for the user to understand the selected scenarios. The transparency overlays could be used for any map including plug-in maps. The implementation of transparency overlays would be a relatively minor modification.

An advanced editing technique could possibly perform automated compositions of paths based on common sections. For example if a user draws a path B that contains responsibilities R1 and R2 and a preexisting but separate path A already contains those

Application of Use Case Maps to System Design with Tool Support

responsibilities the tool could compose the two common sections into one using or forks and joins. The common section would be labelled A+B as expected and the other sections of paths A and B would be labelled the same but enter and leave the common section through an or join followed by the common section then by an or fork. The implementation of automated composition would be a major addition to the tool as intelligent operation would have to be programmed as well as complex manipulations of the hypergraph structures, the complexity of which are not yet performed in the tool.

8.3.3 Substitution of Behaviour Patterns from Standard Libraries

The UCM Navigator as described allows users to create stubs and plug-in maps for these stubs. It allows these plug-in maps to be shared among many stubs. It is a simple extension to allow individual maps for plug-ins to be saved to and loaded from files separately from entire designs. This allows map definitions to be transferred from one design to another. This could be used to create libraries of standard behaviour patterns which could be then plugged in to designs as desired. The tool would be aware of the location of the library maps and the insertion of the standard patterns into stubs would be a standard editor operation.

The ability to load and save maps to separate files could also be used to export maps and even entire sections of designs between files for general editor operation. An operation to export a map would also export its submaps. Similarly an import map operation would import a map along with its plug-in maps. Successively more complex designs could be built on preexisting maps and multilevel designs already entered in the tool. All of these are relatively minor modifications.

8.3.4 Support for layering

The tool does not currently support the concept of layering in the UCM methodology in which lower levels of behaviour in a design are considered as detail and are hidden. With such support the tool would be capable of managing several layers which each would consist of numerous maps of multilevel designs. There would need to be suitable cross-referencing between layers such as service access points in the ROOM methodology [5] so users would be able to explore the interconnections between layers to understand a multi-layer design. There is currently no such cross-referencing capability in the UCM method-

ology. Support for layering would also allow detail which would normally be hidden in a layer of design to be visible with the proper visual cues to identify the elements as layered detail. Adding support for layering would be a major modification to the tool as first appropriate cross-referencing which is necessary between the layers would have to be determined as it does not exist in the methodology, and second, the tool would be to support new visual elements such as this cross-referencing information as well as standard layering elements such as layered data stores.

8.4 Performance Issues

The redrawing of the display in the UCM Navigator was implemented in a very simple manner and as such results in noticeably slow redraws when dealing with large maps, even with fast hardware. This is due to the fact that the display is implemented in the most simple manner possible, with the entire screen being redrawn at each redrawing time. This results in delays as potentially thousands of low level graphics calls need to be made and the graphics buffer needs to be completely rebuilt at each redraw. It should be noted that the minimization of spline recalculation as mentioned in chapter 4 improves the performance of the display greatly over what it would be otherwise as the time to draw the hundreds of short line segments which comprise the splines of the use case paths is much less than it would be to recalculate and redraw them at each interval.

A much more time efficient approach and the approach used in commercial software is to use either separate graphics buffers or drawing modes such as exclusive or, in which an incremental change can be made to a background and reversed later by performing the same operation, to draw the minimum at each screen refresh. This would work as follows. A drawing tool would keep track of what elements are in the background and do not change and which elements would need to change, for example as they were being dragged across the screen, as in a drawing editor. The background elements would then be placed in a graphics buffer by being "drawn" in that buffer. Then some overlay technique would be used to draw the non-constant elements and overlay them on top of the static background elements. The advantage of this is that only the non-constant elements would need to be redrawn at each screen refresh. Screen refreshes in the UCM Navigator have been rather arbitrarily set to occur every 0.1s during motion of the display in order to balance the need for smooth motion with the need for an acceptably long refresh interval that

Application of Use Case Maps to System Design with Tool Support

would not overload a processor, in that with too small of an interval, the display could not be completely drawn before the next refresh interval resulting in continuous drawing operations which would overload the processor. The redrawing of only non-constant graphical elements could involve an exclusive or technique in which new pixels are OR-ed onto the frame buffer of the background in a temporary manner such that no information in the background is lost. It could also involve using separate buffers for the non-constant and the background elements and overlaying them at each refresh. Both techniques have the same effect, only the details are different.

These graphical display techniques could easily be used in the UCM Navigator. It is simply a matter of time and effort. It would require re-implementing the display to keep track of which elements are static and which are not, in each update operation. For example when a component was being moved all child components, enclosed elements and the splines connected to those elements would be marked as changing and drawn in an exclusive-or mode over a background consisting of all the other elements in the map. To speed things up further when complex sections are being moved they could be drawn in outline only. Splines could be drawn as straight line segments during such moves so as to avoid any unnecessary spline recalculations. For simple operations such as selecting map elements or components where highlighting needs to be done or handles need to be drawn the new elements can simply be drawn in XOR mode. The use of XOR mode for simple redraws such as when elements are selected and deselected would be a minor modification. If all such techniques were used the display performance of the tool could increase by orders of magnitude enabling the UCM Navigator to be used on maps as complex as one would desire with excellent display performance. Full implementation of the techniques described would be a relatively major modification, not so much for the complexity of the graphics operations but for the complexity of tracking, through all editor functions, which elements are static background elements and which are changing.

The issue of the performance of the tool only relates to the speed of the display updating. All other operations of the tool perform satisfactorily. It should be noted that the use of multilevel maps does not slow down the tool at all as the number of maps in a design only potentially would affect file saving and loading. The speed of the tool only relates to the complexity of the map being currently displayed. The tool as it exists could be used for maps with hundreds of plug-ins.

8.5 Integrating the UCM Navigator With Other Methodologies

The UCM Navigator is simply designed to allow users to edit UCMs to describe the behaviour of systems. However UCMs were never intended to be the sole methodology used for design. UCMs are intended to express high level behaviour signatures above the levels of inter-object messages and small scale objects. However, in any design, once implementation is begun lower levels need to be designed and documented.

Widely accepted OO methods [2] [3] for describing the detailed design of a system include class diagrams such as class hierarchy diagrams and class relationship diagrams for describing the relationships among classes and messaging diagrams [1] [4] such as message sequence charts and collaboration graphs, where numbered sequences of messages are shown within the context of a component diagram. These traditional methodologies describe the object structure and detailed object behaviour of a design. UCMs were created as a means of filling a gap in the traditional specification of software systems, the gap being the high level end to end descriptions of scenarios occurring through a system. Therefore they are intended to complete traditional software descriptions rather than compete with them.

Future work could therefore involve integrating other traditional software descriptions into the UCM Navigator. For example, as a method for describing the class structure of an application editors for class hierarchy diagrams and class relationship diagrams (showing relationships such aggregation and usage) could be added to the tool enabling a set of classes to be described which could then be used in UCMs. The tool would support cross-referencing between the software components appearing in Use Case Maps and those in the relationship diagrams. In order to describe the detailed behaviour of a responsibility in terms of the actual exchange of messages an editor for message sequence charts could be added to the tool in which users could specify the set of messages sent and received by the objects involved. This MSC would then be indexed into the responsibility appearing in a UCM so that the user could view the detailed behaviour of a responsibility. Alternately collaboration graphs could be used which would show numbered messages sent between components which could possibly be the same component context diagram that appeared in the corresponding UCM. The object types appearing in these MSCs or collaboration graphs could be indexed into the class hierarchy/relationship diagrams described above.

Application of Use Case Maps to System Design with Tool Support

Integrating standard methodologies for design description into the UCM Navigator would not only provide a standard repository of design information for system designers but doing so would explore many issues involved with the integration of UCMs with traditional methodologies. However implementing these additions to the tool would be a very major effort as essentially the UCM Navigator would be transformed from a graphical editor for UCMs to one for all popular software engineering notation. The resulting program could easily be double to triple the current size of the tool.

8.6 Integration of the UCM Navigator With Other Applications

The previous sections described issues relating to the use of the UCM Navigator as a tool for the editing of UCMs possibly with integration with other methodologies. This section describes how the tool could be integrated tightly with other applications, specifically the applications for which it is used as a front end. This includes performance simulation environments for which the tool allows the collection of performance related data and such tools as agent code generators which make use of the tool's ability to specify dynamically selectable behaviour signatures (through the use of plug-in maps for stubs) along with their respective pre/postconditions.

There already exists forward integration between these applications and the UCM Navigator as the tool is capable of passing design information through the linear form (as with the agent tools) or through its own textual file format (currently used for the performance project). This section discusses how the integration can be two-way, that is how these applications can pass information back into the UCM Navigator. Examples of information being passed back into the tool include results from performance simulations, the ability to visualize executing scenarios from code generated using design information from the tool through the use of the linear form, and the ability to transform the designs entered in the tool through the substitution of concrete behaviour patterns for formal parameters by manipulation of the UCM file format.

8.6.1 Transformation of Linear Form into UCM File Format

This section discusses how the linear form which is currently a textual representation of the structure of a map or design and does not contain graphical information could be transformed into a replacement file format for the tool. The significance of this is that

Application of Use Case Maps to System Design with Tool Support

such a file format could be transformed more easily by other applications to add information or possibly change the structure of the map.

The UCM Navigator currently supports two file formats, its own textual file format and the linear form. The tool's own file format is simply a dump of all the information in the objects in a given design in a format that the tool's loading routines understand so as to rebuild the object structure. It was never designed to be readable or easy to manipulate and is not in any standard format. The linear form expresses the structure of the tool in a standard EBNF syntax which can be parsed by standard tools such as lex, yacc and the Java parser JCC which is currently used in the agent project. It was not intended at the time to be a replacement file format for the tool and as such does not contain any graphical information such as positions, orientations and colors. It does not even contain information for the performance extensions as the linear form was developed for the agent project which doesn't use the performance extensions.

It is a simple matter to include the missing information (graphical and performance) into the linear form so that it can assume the role of the tool file format, eliminating the need for the separate tool file format. Doing so would be advantageous both to the users and developers of the tool as users would be required to deal with only one file format and developers would only have to support one file format. It would also make the tool file format a much more standard one that could be manipulated by standard applications. If this new file format were written using XML (Extensible Markup Language) which is a standard for which much code support exists in the way of both parsers and tools which can transform an input file specified in XML format it would be even more powerful as an interface between the UCM Navigator and other tools. Replacing the current file format with a properly expanded linear form would be a major modification as all file loading code would need to be rewritten and complex object structures would need to be reconstructed.

8.6.2 Display of Feedback from Performance Simulations

The UCM Navigator, as described in chapter 6.:Extensions for Performance Prediction, has been extended for its use as a front end in the performance prediction of software systems. Performance information that is normally not associated with UCMs such as response time requirements, time distributions of the starting of scenarios, choices

Application of Use Case Maps to System Design with Tool Support

between scenarios, access to global data stores as well as hardware device information has been added to the tool through the use of numerous dialogs as well as additional map elements (timestamp points). This information is all gathered by the tool and exported in the file format. A simulation environment reads in the input file extracting all the path and component information as well as the performance specific information and constructs the objects necessary to simulate the system.

The transformation is currently only one way as graphical UCM models that are simulated by a simulation environment produce only textual output which can be cross-referenced to the originating map by means of labels for map elements and response time requirements. It is desired that the UCM Navigator be a complete simulation environment in which designs can be drawn in the editor, performance annotations can be made, optionally simulations can be invoked in the editor, and simulation output can be viewed in the editor by means of annotations to map elements and additional information in the current dialog boxes. Examples of information that could be returned from performance simulations includes information regarding whether the desired response time requirements were met, which could be displayed in the response time requirement dialogs, information about processor utilization, information about contention for global data stores, information about the use of hardware by software processes and by individual responsibilities and virtually any performance statistic that relates to the elements in the originating UCM. This information would be displayed in the appropriate manner, either in a dialog box relating to that information or possibly by annotating map elements themselves.

The method of transfer of simulation data from the simulator back into the tool would be the tool's output file. When the replacement of the tool's current file format with an XML format is done, optional elements in the format description could be added so that information could be passed back into the tool from simulation environments or possibly other tools. Information about a performance simulation could be added by the simulator which would read in the file to construct the objects to perform the simulation, run the simulation, and then output the simulation data back into the optional fields of the input file from which the simulation was performed. This is possible as the file format would be in the well defined XML format. This file could then be read back into the UCM Navigator to display the simulation results as described above. Implementing the feedback of performance data would vary in complexity of implementation from relatively minor to major

depending on the amount of information passed back and the desired complexity of the visual display in the tool.

8.6.3 Transformation of UCM designs by external applications

The previous section on the transformation of the linear form into a replacement file format using XML mentioned how data specified in XML format could be easily transformed by external applications. One possible application of this is the feedback of simulation results that was described in the previous section. However such feedback does not change the structure of the maps, it simply annotates elements.

It is possible through file format manipulation to manipulate the path structure of a UCM specified in an XML format. This opens up many possibilities. Variations of communication patterns could be automatically substituted by external applications. For example a design can be created in the tool which specifies the desire for certain object communication but does not specify details, such as providing stubs for such details but no plug-ins. An external application could fill in such details by transforming the XML described map by providing plug-ins for these unfilled stubs. Alternatively if it was not desired to clutter the main map with stubs at points at which specific patterns could be inserted empty points along paths could be marked and sections of paths which implement desired patterns could be inserted at these points during processing. The resulting design could then be simulated to determine its performance. Several design alternatives could be easily compared. Strictly speaking the ability to specify plug-in maps for stubs could accomplish the same task of the substitution of design details but manual substitution would need to be performed and stub bindings would need to be made manually which would be cumbersome and inefficient for large designs. Automated transformations allow numerous changes to be made simultaneously. The implementation of these abilities would be a major effort as the tool would have to properly display path structures that have no graphical information, in the sections of paths that are inserted by external programs.

8.6.4 Visualization of Scenarios

The UCM methodology is designed to allow designers to specify end-to-end behaviour signatures through systems at the requirements engineering phase. If these scenarios were used as a front end for generating code for the system they would also be

Application of Use Case Maps to System Design with Tool Support

descriptions of the systems' intended behaviour. As the code generation process might have errors it would be useful to compare the intended behaviour signatures with the actual generated code. The best way to do this is to visualize the actual behaviour signatures in the environment in which the system was created, that is in terms of the original UCM scenarios.

The UCM Navigator could be extended to allow scenarios to be traced on screen as a series of events occurring along paths. These events could be the execution of responsibilities, the starting and terminating of paths, or the choice of various scenarios. The visualization could take various forms. It could involve the highlighting of path elements, the displaying of icons representing tokens along paths or the displaying of text messages on screen.

This visualization would be possible by writing an API for this purpose. Code that was originally generated from the UCM diagrams (or possibly simply code that was hand written but implements a system described in its early stages using UCMs) would call the functions of this API and cause the progress through this code to be shown on screen so that the designers could compare the actual behaviour with the original intended behaviour. The implementation of scenario visualization would be a major undertaking as first appropriate visual highlighting API functions would need to be determined and implemented, and secondly the functions of this API would need to be used correctly in order to trace the execution of a scenario based on input, most likely from a file which would require the determination of a proper file format and implementation of the parsing of the file implemented in the tool.

9. Conclusions and Future Work

9.1 Conclusions

This thesis involved the building of a prototype Use Case Map editor as a research tool to explore aspects of using UCMs in the high level design of large systems. The research involved many facets of a UCM development environment such as user interface issues, support for scalability of large designs, as well as the use of such an editor as a front end for other software work such as agent development and performance simulations of systems specified with UCMs. In summary this thesis explored many issues including the following:

- The use of a UCM editor as a research tool for the application of the UCM methodology to large designs both as a means of visualization and documentation as well as design
- Support for scalability of large designs through the use of stubs along use case paths with associated submaps allowing complex designs to be decomposed into many levels
- Explicit support for dynamically selectable behaviour signatures through the use of dynamic stubs with multiple plug-in maps defined
- Support for generation of a linear textual output of the structure of maps that can be used as an interface for other software engineering applications that would use UCMs as a means of specifying a high level design and which would perform further processing on the design
- The exploration of the use of UCMs and the UCM tool as a means of specifying the execution characteristics of a system such that performance simulations could be made from a UCM design which had been properly annotated with execution information. This involved discovering the proper characteristics and how they must be applied to UCMs

9.1.1 UCM Editor Issues

The basic editor development provided a platform for the use of UCMs for the visualization and design of large systems. The ability to manipulate UCMs in a graphical

Application of Use Case Maps to System Design with Tool Support

editor as UCMs and not as graphical objects in a standard graphical editor provided both validation of entered designs and much greater ease of manipulation and maintenance. The existence of such an editor made the use of the UCM methodology more practical and attracted much interest in the local UCM community.

The concept of decomposition through the use of stubs along use case paths was considered first simply a method of increasing the scalability of designs as stubs could have submaps defined leading to arbitrarily complex multilevel maps. The visualization aspects of UCMs led the researchers in the agent project to use UCMs to visualize complex inter-agent interactions and led to the desire to use the UCM concept of stubbing to specify dynamically selectable behaviour signatures, as for example alternate behaviour of an agent depending on system conditions, where many alternate plug-in submaps could be defined and the specific submap would be chosen at run-time. The concept of dynamic stubs with multiple plug-ins was added to the tool.

Interest in the use of the UCM editor as a front end for other applications specifically agent tools custom built for the agent project led to the development of the linear form by the UCM research team, a textual description of the structural form of a UCM design which could be used as input to numerous other tools. The generation of the UCM linear form from graphical designs entered in the tool was added.

With the generation of linear form available and the concept of dynamically pluggable behaviour signatures added the tool became the front end for the agent project as described in the agent case study in chapter 7.

9.1.2 Extensions for Performance Prediction

The use of UCMs as a means of specifying the performance characteristics necessary to provide input for a simulation environment involved providing execution detail that would need to be provided, such as the hardware resources used by responsibilities, the statistical processes by which scenarios begin, as well as the characteristics of the hardware devices in a design. It also involved realizing where the UCM methodology, in its main use as part of the requirements engineering process, was too strict in that it could not specify detail that would be needed in an execution environment. In order to quantify the results of a simulation extensions to use case paths were created to delimit intervals which could have performance requirements attached to them.

Application of Use Case Maps to System Design with Tool Support

The provision of execution detail involved the ability to specify resource functions for responsibilities as well as the ability to specify time distributions for arrival events at path start points. The ability was also provided to mark places along use case paths with timestamp points which could then be grouped to delimit path segments to which response time requirements could be specified. These requirements were then verified by the simulation environment. Provision was also provided for the specification of hardware devices and global data stores.

It was also discovered that the methodology was too strict in two main areas, the ability to specify choices at or forks at run time, and the ability to specify global data that is accessed by responsibilities which an execution environment would need for mutual exclusion implications. The concept of or forks as superposition of paths where no choice was taken as the entire end-to-end path was predetermined by the path preconditions did not allow the specification of alternate execution based on run-time data. The concept in UCMs of data being of a lower layer and invisible at higher layers was restrictive in the specification of access to global data.

9.2 Thesis Contributions

The main work in this study was to develop tool support for the manipulation of Use Case Maps for their use in the high level design of systems. The main results of this work were:

- The representation of system structures and behaviours at high levels of abstraction through the use of complex multilevel maps by tool support
- The generation of a linear textual output, constructed from a context-free grammar from graphical designs entered in the tool
- The provision of support for the definition of dynamically changing behaviour signatures through the use of alternative plug-ins
- The provision of support for the attachment of performance information to designs specified in terms of UCMs enabling the tool's use as a front end for performance simulation of systems
- Provided an environment in which to test the UCM methodology with industry-posed problems. The ability to specify dynamic behaviour patterns was very useful to the agent system project of a local company

9.3 Future Work

The work in this thesis involved the UCM Navigator, a prototype UCM tool that allows designers to edit UCMs in an environment with syntax checking and which generates a textual form for maps allowing the tool to be used as a front end for other tools. The tool has also been extended to collect data needed for performance simulations so that the tool can be used as a front end for performance prediction of systems. However there are many directions in which the development of the UCM Navigator can be extended.

9.3.1 Advanced Editing and Display Properties

The editor currently supports a basic set of operations necessary to create UCMs and multilevel designs with UCMs. However many operations may be added which would increase the editing power of the tool. These include support for undoing operations, for copying or moving sections of maps between maps (useful for moving functionality from root maps into submaps) and for rotating and scaling sections of maps as well as verifying the completeness of a design such as the existence of all appropriate scenario labels, conditions/events, stub bindings as well as performance annotations.

There also exist improvements that could be made to the tool's map editing window. The tool currently has a fixed size editing window with constant element sizes. The editing area could be made scrollable so that virtual editing areas could be used to allow the creation of much larger maps than the available screen space. In addition the editing area could be made zoomable so that map sections could be enlarged to show fine detail or shrunk so that more information could fit within the available space.

Improvements could also be made to the user interface to increase user-friendliness. The tool currently displays the map transformations that are possible with the selected elements by way of popup menus. This is not necessarily desirable as the possible operations are hidden. An optional way of implementing the display and invocation of transformations is with rows of graphical controls, one for each operation. The controls would be grayed out and unavailable if the operation was not applicable at the time.

9.3.2 Design Scalability Support

The UCM Navigator provides support for the design of complex multilevel maps

Application of Use Case Maps to System Design with Tool Support

through the use of stubs with submaps defined. This allows complex designs to be created. However there is much room for improvement in the ability of the tool to manage these many levels in terms of enabling users to simultaneously view maps of different level and, to provide cross-referencing between these levels.

One of the problems with decomposition is that context of the end to end flow through a system is lost as successive levels are expanded. One of the means for remedying this problem is to allow multiple maps to be visible simultaneously by having many editing areas which could simultaneously display separate levels of a design. Cross-referencing information would allow the user to see the connections between the levels such as the ability to display stub binding information on the screen.

Another more ambitious way of displaying information from many levels simultaneously is the use of fisheye transformations as described in the thesis by Charles Cui. Fisheye transformations are nonlinear transformations which allow many levels of a graphical design to be visible in a given area. The process works by nonlinearly distorting the display of a higher level when it is desired to show the internal composition of a lower level. The thesis research was based on the concept of hierarchical component structures where a given level would show the components at that level but hide the contents of those components. When it was desired to show the internal contents of that component it would be expanded in a central window on the screen with the previous component level being nonlinearly distorted so that it could fit outside the boundaries of the newly expanded component. The process of recursive decomposition could theoretically be continued indefinitely with successive components being expanded and all levels being visible on screen through the use of nonlinear distortions on the higher levels. The only limit would be screen space and resolution limitations.

With respect to the UCM Navigator fisheye transformations could be used to display the map hierarchy from the root map to the currently displayed plug-in map. The major difference is that with UCMs the unit of decomposition is a stub with a plug-in map as opposed to a hierarchical component structure. A visual notation would need to be created to signify the expansion of the submap of a stub. It would then be possible to display nonlinear transformations of all of the higher level maps of the currently displayed map in order to provide context.

Support for the specification of large designs with the tool would be helped by formal support for the UCM concept of layering in which lower levels of behaviour in a

Application of Use Case Maps to System Design with Tool Support

design are considered as detail and are hidden at higher layers of higher level of abstraction. With such support the tool would be capable of managing a multi-layered design where each layer could consist of complex multi-level designs. What would need to be created is a method of cross-referencing information between layers so that the connections between layers were made visible when desired.

Other future work that could be done to improve the support for large designs includes support for the importation of standard design patterns from known libraries into the tool, support for designs with multiple top level (root level) maps, and support for the concept of transparency overlays in which it would be possible to display only selected paths and the components that cross them. This last feature would allow a normally complex and cluttered UCM to be decomposed into overlays of simpler maps in which sets of paths which correspond to the paths used to specify a certain use case could be grouped and selected for display by the user. This would allow the specification of more complex maps than is now practical.

9.3.3 Performance Improvements

The redrawing of the display of the UCM Navigator is currently implemented in a very simple manner with the entire screen being cleared and redrawn at each screen refresh. Thus results in noticeably slow screen refreshes with large maps even with fast hardware due to the large number of low level graphics calls. It is possible to improve this greatly through the use of complex redrawing operations in which only elements which need to be changed are erased and redrawn. This requires the use of a complex drawing mode such as exclusive-or in which new pixels are drawn on a background in such a manner that the operation can be reversed. This would allow the tool to keep track of which elements need to be changed and redrawn with each refresh and which could remain in the background.

9.3.4 Integration with other methodologies and applications

The UCM methodology was never intended to be a stand-alone design methodology but rather a complement to traditional design specification techniques such as class hierarchy and relationship diagrams, message sequence charts and state machines. Possible future work for the tool would include integrating these standard methodologies. This

Application of Use Case Maps to System Design with Tool Support

has many possibilities. For example editors for class hierarchy and relationship diagrams could be added to the tool to express relationships between the classes in a design. These classes could then be used to create instances in the UCMs of the design with cross-referencing capabilities provided. Another possible extension is the capability to specify message sequence charts (MSC) as a means of specifying the actions of a responsibility.

In addition to the UCM methodology being a supplement to other methodologies the UCM Navigator was always intended to be a front end for other applications. The prime examples described in this thesis are the generation of linear form for the agent project and the use of the tool as a front end for performance simulations. However these uses of the tool are one way only, that is the tool transfers information to these environments but cannot receive information in return. True integration with other applications would mean two way communication with these applications. It would be desired for other applications to return information to the UCM tool as well as possibly transform designs that were created using the tool.

Examples of information being returned from applications includes feedback of results from performance simulations as well as visualization of the execution of scenarios which depict the execution of code generated with information from the tool. This latter extension would require an appropriate series of animation operations to be added to the tool. An example of the transformation of designs entered into the tool by external applications is an application which substitutes various standard detail communication patterns at specific points in a design as a means of evaluating design alternatives. All of the alternatives could then be simulated to find the best design.

In order for both the feedback of information and possible transformation of UCM designs to be performed by external applications it would be necessary to replace the current file format with a much more understandable format which could be parsed and transformed by external applications. The use of a standard syntax such as XML (Extensible Markup Language) for which much tool support exists would enable the current file format to be replaced by a new linear form which assumes the functions of both formats. Such a format could be more easily transformed by adding information such as results of performance simulations or possibly even transformed so that external applications may transform use case paths that were entered in the tool.

References

- [1] ITU, "Recommendation Z.120: Message Sequence Charts (MSC), Geneva, 1996
- [2] Grady Booch, "Object Oriented Design with Applications", Benjamin/Cummings, 1994
- [3] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object-Oriented Modeling and Design", Prentice-Hall, 1991
- [4] I. Jacobson et. al., "Object-Oriented Software Engineering (A Use Case Driven Approach)", ACM Press, Addison-Wesley, 1992
- [5] B. Selic, G. Gullickson, and P.T. Ward, "Real-time Object-Oriented Modelling", Wiley, 1994
- [6] Buhr, R.J.A. "Use Case Maps as Architectural Entities for Complex Systems", to appear in a forthcoming IEEE Transactions on Software Engineering special issue on Scenario Management, 1998
- [7] Buhr, R.J.A. and Casselman, R.S. "Use Case Maps for Object-Oriented Systems", Prentice-Hall, 1995
- [8] Michel Locas, "Use of Hypergraphs to Represent Use Case Maps", incomplete M.Eng. thesis, unpublished
- [9] G. Farnsworth and W.Kok, "A Simple Use Case Map Navigator", Fourth year project report, Dept. of Systems and Computer Engineering, Carleton University, 1996, unpublished
- [10] J. Carriere, "The Design of a Use Case Map Editor", Course project report, Dept. of Systems and Computer Engineering, Carleton University, 1996, unpublished
- [11] Chris Witham, "Use Case Map Navigator User's Manual", Dept. of Systems and Computer Engineering, Carleton University, 1998, <http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/ucmnav>
- [12] UML Notation Guide, version 1.1, Object Management Group, September 1997
- [13] "Annotated UCM Grammar", <http://www.sce.carleton.ca/rads/agents>
- [14] C. Scratchley and A. Miga, "A Use-Case Map Editing Tool with Performance Pre-

- diction Capabilities”, ObjecTime Workshop on Research in Real-Time Object-Oriented Modelling, January 13-14, 1998, Kanata, Ontario, Canada
- [15] C.Scratchley, “Automated Identification of Hardware Efficient Skeletal Designs”, Ph.D. thesis proposal, Dept. of Systems and Computer Engineering, Carleton University, 1997, unpublished
- [16] R.J.A. Buhr, D.Amyot, M.Elammari, D.Quesnel, T.Gray, S.Mankovski, “High Level Multi-Agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example”, Third International Conference and Exhibition on the Practical Applications of Intelligent Agents and Multi-Agents (PAAM’98) 23-27 March 1998, London, UK
- [17] C. Cui, “Fisheye Visualization Techniques for Large Software Designs”, M.C.S. thesis, Faculty of Computer Science, Carleton University, 1993

Appendix - Class Descriptions and Interfaces

Hyperedge subclasses

Concrete classes:

Connect	Empty	OrFork
OrJoin	Responsibility	Result
Start	Stub	Synchronization
Timer	Timestamp	Wait

Abstract classes:

Hyperedge	MultipathEdge	WaitingPlace
-----------	---------------	--------------

The hyperedge classes form the main classes in the design of the UCM Navigator. They represent the main UCM elements and are responsible for all of the operations on those elements, that is they contain the code for the operations. For example a Responsibility is responsible for editing its description as well as deleting itself. A Synchronization hyperedge is responsible for adding or deleting branches of and forks or and joins and for deleting itself in the hypergraph when it is no longer needed. All hyperedges are responsible for generating their information in the linear form output. The hyperedges are described as follows:

Class Hyperedge - Abstract

```
class Hyperedge {  
  
    friend class FileManager;  
  
public:  
  
    Hyperedge( boolean collections );  
        // constructor, collections flag allows different behaviour  
    virtual ~Hyperedge();  
  
    virtual void AttachSource( Node *new_node );  
        // polymorphic method for attaching source nodes to this  
        hyperedge
```

Application of Use Case Maps to System Design with Tool Support

```
virtual void AttachTarget( Node *new_node );
    // polymorphic method for attaching target nodes to this
    hyperedge
virtual void DetachSource( Node *node );
    // polymorphic method for detaching source nodes
virtual void DetachTarget( Node *node );
    // polymorphic method for detaching target nodes

Cltn<Node *> *SourceSet() { return ( source ); }
    // access method for source node list
Cltn<Node *> *TargetSet() { return ( target ); }
    // access method for target node list
HyperedgeFigure *GetFigure() { return ( figure ); }
    // access method for figure object

boolean HasSourceColour( nodeColour colour );
    // returns flag as to the existence of a source node of a
    certain colour
boolean HasTargetColour( nodeColour colour );
    // returns flag as to whether a target node of a certain
    colour exists
Cltn<Node *> *SourceOfColour( nodeColour colour );
    // returns subset of source node list matching colour
Cltn<Node *> *TargetOfColour( nodeColour colour );
    // returns subset of target node list matching colour

void TransferConnectTo( Hyperedge *edge );
    // transfers node colour D connections to edge
void DeletePath();
    // prompts user for confirmation, deletes entire path

void Transform( transformation trans );
    // actually performs the transformation trans
void TransformDouble( transformation trans, Hyperedge *edge );
    // actually performs the double selection transformation
    trans with edge
boolean Validate( transformation trans );
    // validates whether transformation trans is valid
boolean ValidateDouble( transformation trans, Hyperedge *edge
    ); // validates whether double selection transformation
    transformation trans is valid
virtual boolean Perform( transformation trans, execution_flag
    execute ); // polymorphic method, validates or performs
    transformation trans
// polymorphic method, validates or performs double selection
    transformation trans
```

Application of Use Case Maps to System Design with Tool Support

```
virtual boolean PerformDouble( transformation trans, Hyperedge
    *edge, execution_flag execute );

virtual edge_type EdgeType()=0; // polymorphic method which
    returns integer flag denoting subclass type

virtual void EdgeSelected(); // polymorphic method which is
    called when the map element corresponding to this hyperedge
    is selected
virtual char * LabelBalanced(); // polymorphic method which
    returns dialog prompt for path label dialog if label is obso-
    lete
virtual void Save( FILE *fp ); // polymorphic method which
    saves hyperedge's data to file
virtual boolean Generate( ofstream &linear, Hyperedge
    *previous_edge )=0; // polymorphic method which generates
    hyperedge's data to linear form file
virtual char * AtomIdentifier()=0; // polymorphic method which
    returns identifier for hyperedge
virtual void GenerateIdentifier(); // polymorphic method which
    generates hyperedge identifier
virtual boolean IsAtomReference(); // polymorphic method which
    returns whether a hyperedge is a responsibility or other
    atom
virtual void GeneratePathConnections(); // polymorphic method
    which generates path connections to linear form output for
    connect hyperedges

int GetNumber()    { return( hyperedge_number ); } // returns
    the internal integer identifier for this hyperedge

Label *PathLabel() { return( label ); } // returns the hyper-
    edge's path label object
void SetLabel( Label *new_label ); // allows label object to
    be changed
void PreviousOutput( int output ) { previous_output = output;
    } // returns identifier of previous hyperedge, used in lin-
    ear form generation
void Output(); // debugging method which outputs data for
    hyperedge

static void ResetHyperedgeCount() { number_hyperedges = 0; } /
    / resets global count of hyperedges, used when editing new
    files
```

protected:

Application of Use Case Maps to System Design with Tool Support

```
void DeletePathSection(); // deletes the path containing this
                           hyperedge

Cltn<Node *> *source, *target; // lists of source and target
                           nodes
HyperedgeFigure *figure; // figure object which manages graph-
                           ical operation of map element
Label *label; // path label object

int hyperedge_number, load_number, generation_count,
  previous_output; // various integer counters
static int number_hyperedges;

};
```

Class MultipathEdge - Abstract

```
class MultipathEdge : public Hyperedge {

public:

    MultipathEdge( boolean collections ) : Hyperedge( collections )
    {}
    ~MultipathEdge() {}
    virtual char * AtomIdentifier() { return( identifier ); } //
        polymorphic method which returns identifier for multipath
        edge

    virtual void Delete()=0; // polymorphic method which deletes
        multipath edge when it becomes redundant
    virtual int PathCount()=0; // polymorphic method which returns
        number of inputs or outputs, or maximum if both exist

protected:

    char identifier[7]; // text identifier used in linear form
        output

};
```

Class WaitingPlace - Abstract

```
class WaitingPlace : public Hyperedge {
```

Application of Use Case Maps to System Design with Tool Support

public:

```
WaitingPlace( boolean collections );
~WaitingPlace();

virtual void AttachTarget( Node *new_node ); // implements
specific behaviour for class for attaching target nodes
virtual void EdgeSelected(); // implementation of polymorphic
method which displays conditions in main window
virtual void Save( FILE *fp ); // implementation of Save
method for waiting places
virtual char * AtomIdentifier() { return( identifier ); } //
returns textual identifier

boolean ConnectPath( Hyperedge *trigger_edge, execution_flag
execute ); // hypergraph transformation which attaches a
triggering element to waiting place
boolean DisconnectPath( Hyperedge *trigger_edge, execution_flag
execute ); // hypergraph transformation which detaches
triggering elements if they exist

virtual boolean Generate( ofstream &linear, Hyperedge
*previous_edge ); // implements linear form generation
method for this subclass
static void ResetCounter() { number_waits = 1; } // resets
global counter for class
void WaitingName( char *new_name ); // sets new identifier
```

protected:

```
void DeleteWaiting(); // deletes the waiting place map element
from map, called by subclasses
boolean EditWaitingName( execution_flag execute ); // allows
user to edit identifier of element

ConditionManager condition_manager; // instance of object
which manages textual conditions
char identifier[20]; // user given textual identifier

static int number_waits; // global count of instances of sub-
class
```

private:

```
void BreakConnections(); // breaks triggering connections to
```



```
        this element if they exist
};
```

Class Connect - Concrete

The **Connect** hyperedge is the only hyperedge that does not have a graphical representation. It represents the connections made between paths such as between the Result and Start of separate paths or between a Result /Empty and Timer/Wait. Connect hyperedges are placed between the elements of different paths as a means of marking connections. Connect hyperedges accept source and target nodes of different colors to distinguish interpath connections from normal hypergraph connections.

```
class Connect : public Hyperedge {
public:
    Connect();
    ~Connect() {}

    void AttachSource( Node *newNode ); // implement specific
        behaviour for Connect hyperedges
    void AttachTarget( Node *newNode );

    virtual edge_type EdgeType() { return( CONNECTION ); } //
        returns flag identifying hyperedge type
    virtual boolean Generate( ofstream &, Hyperedge * ) { return
        QUIT; } // default method which stops traversal of hyper-
        graph during linear form generation
    virtual char * AtomIdentifier() { return( NULL ); } // default
        method as Connect edges need no identifier
};
```

Class Empty - Concrete

The **Empty** hyperedge does not correspond to any use case map element but nevertheless is very much necessary in the design of the editor. It acts as a placeholder along paths allowing all other map elements to be inserted at that point in the hypergraph. It is responsible for inserting all of the other hyperedges into use case paths. It also is capable of being given path characteristics such as preconditions and postconditions and graphical notations representing failure points of paths or shared responsibilities between compo-

Application of Use Case Maps to System Design with Tool Support

nents.

```
class Empty : public Hyperedge {

public:

    Empty( boolean collections = TRUE );
    ~Empty();

    virtual edge_type EdgeType() { return( EMPTY ); } // returns
        identifier for subclass type
    virtual void EdgeSelected(); // called when element is
        selected, places pre/postconditions in main window display
    virtual void AttachSource( Node *newNode ); // implement spe-
        cific behaviour for empty hyperedges
    virtual void AttachTarget( Node *newNode );

    OrJoin *AddOrJoin(); // replaces itself with an or join at
        the same point of the path in the hypergraph structure
    Synchronization *AddAndJoin(); // replaces itself with an and
        join at the same point of the path in the hypergraph struc-
        ture

    boolean ConnectPath( Hyperedge *trigger_edge, execution_flag
        execute ); // transformation which connects this empty
        point as a trigger for a waiting place
    boolean DisconnectPath( Hyperedge *trigger_edge, execution_flag
        execute ); // transformation which disconnects a triggering
        connection if one exists
    boolean Cut( execution_flag execute ); // transformation which
        cuts the path into two sections at the location of this empty
    boolean AndCompose( Hyperedge *edge, execution_flag execute );
        // transformation which creates and and join with the path of
        edge at the current location
    boolean OrCompose( Hyperedge *edge, execution_flag execute );
        // transformation which creates and or join with the path of
        edge at the current location
    boolean DecomposeFromJoin( execution_flag execute ); // trans-
        formation which detaches the path of this empty point from an
        and/or join
    boolean DecomposeFromStub( execution_flag execute ); // trans-
        formation which detaches the path of this empty point from
        the input or output of a stub
    boolean DeleteBranch( execution_flag execute ); // transforma-
        tion which deletes the branch of an and/or fork on which this
        empty point resides
    boolean Perform( transformation trans, execution_flag execute
```

Application of Use Case Maps to System Design with Tool Support

```
); // executes or validates the transformation with code
trans

virtual void Save( FILE *fp ); // implements Save method for
this empty hyperedge
virtual boolean Generate( ofstream &linear, Hyperedge
*previous_edge ); // implements linear form generation
method for this empty hyperedge
virtual char * AtomIdentifier() { return( identifier ); } //
returns textual identifier for this element
virtual void GenerateIdentifier(); // generates a textual
identifier for this element
virtual boolean IsAtomReference(); // determines whether this
empty has useful characteristics so as to be included in the
linear form

static void ResetCounter() { number_emptyies = 1; } // resets
the global counter of empty hyperedges
void PreviousInput( int input ) { previous_input = input; } /
/ stores the identifier of the previous hyperedge, used in
linear form generation
int PreviousInput() { return( previous_input ); } // returns
the identifier of the previous hyperedge
void NewSemiPath() { new_semi_path = TRUE; } // sets a flag
used in linear form generation
void PathNumber( int new_number ) { path_number = new_number; }
// sets the indentfier of the semi path
int PathNumber() { return( path_number ); } // returns the
identifier of the semi path
void CopyConditions( ConditionManager &cm ); // allows pre/
post conditions to be copied to anther empty hyperedge when
this instance is deleted
```

private:

```
boolean AddResponsibility( execution_flag execute ); // trans-
formation which replaces this object with a responsibility
hyperedge at the current location
boolean AddAndFork( execution_flag execute ); // transformation
which replaces this point on the path with an and fork struc-
ture
boolean AddOrFork( execution_flag execute ); // transformation
which replaces this point on the path with an or fork struc-
ture
boolean AddStub( execution_flag execute ); // transformation
which replaces this object with a stub hyperedge at the cur-
```

Application of Use Case Maps to System Design with Tool Support

```
    rent location
boolean AddTimer( execution_flag execute ); // transformation
    which replaces this object with a timer hyperedge at the cur-
    rent location
boolean AddTimestamp( execution_flag execute ); // transforma-
    tion which replaces this object with a timestamp hyperedge
    at the current location
boolean AddWait( execution_flag execute ); // transformation
    which replaces this object with a wait hyperedge at the cur-
    rent location
boolean DeletePoint( execution_flag execute ); // transforma-
    tion which deletes this empty point on the path
boolean AddPoint( execution_flag execute ); // transformation
    which adds another empty point to the current path located to
    the right of this one
boolean DeletePath( execution_flag execute ); // transforma-
    tion which deletes the path on which this empty point is
    located at user's confirmation
boolean EditPathLabel( execution_flag execute ); // allows user
    to edit the scenario label of this path segment
boolean EnablePathLabel( execution_flag execute ); // user
    operation which causes the scenario label to be visible if
    the viewing of labels is disabled
boolean DisablePathLabel( execution_flag execute ); // user
    operation which cancels the visibility of the scenario label
boolean AddFailurePoint( execution_flag execute ); // user
    operations which add and remove the failure point notation
    to this empty point
boolean RemoveFailurePoint( execution_flag execute );
boolean AddSharedResponsibility( execution_flag execute ); //
    user operations which add and remove the failure point nota-
    tion to this empty point
boolean RemoveSharedResponsibility( execution_flag execute );
boolean EditBranchChoice( execution_flag execute ); // allows
    user to enter the selection criteria for this branch of an or
    fork, used in simulations
void AddSimple( Hyperedge *edge ); // private method which
    replaces this object with hyperedge edge, adds extra empty
    points if necessary
void DecomposeFrom( MultipathEdge *mp_edge ); // decomposes
    the path of this empty point from multipath edge edge

ConditionManager condition_manager; // object which manages
    the pre/postconditions
boolean display_label, new_semi_path; // flags representing
    displaying of scenario label and starting of new semi path in
```

Application of Use Case Maps to System Design with Tool Support

```
        linear form generation
int characteristics, // flag representing annotations to this
    empty point ( failure point / shared responsibility )
    previous_input, path_number; // variables used in linear
    form generation
char identifier[10]; // textual identifier for this hyperedge
static int number_empties; // global count of empty hyperedges

};
```

Class OrFork - Concrete

The OrFork hyperedge represents or forks in the UCM notation. It is responsible for adding and deleting output branches and deleting itself when no longer needed. An additional requirement of the performance extensions is that it is responsible for managing user input relating to branch selection characteristics for its output branches during simulations.

```
class ExitPoint { // data class used by OrFork

public:

    ExitPoint( Node *exit, char *choice = NULL )
    {
        exit_point = exit;
        branch_choice = choice;
    }

    ~ExitPoint() {}

    Node *exit_point; // an output node of the fork
    char *branch_choice; // text string representing branch selection mechanism

};

class OrFork : public MultipathEdge {

public:

    OrFork( boolean collections = TRUE );
    ~OrFork() {}

    edge_type EdgeType() { return( OR_FORK ); } // returns identi-
```

Application of Use Case Maps to System Design with Tool Support

```
fier for subclass type
boolean Perform( transformation trans, execution_flag execute
); // executes or validates transformations with code trans
void ExtendFork(); // creates second output branch or fork
void Delete(); // deletes the fork when it becomes unnecessary
void EditBranchChoice( Empty *empty ); // allows user to spec-
    ify branch choice string

virtual void AttachSource( Node *new_node ); // performs error
    checking behaviour
virtual void AttachTarget( Node *new_node ); // adds ExitPoint
    data object to exit points list for each new target node
virtual void DetachTarget( Node *node ); // removes corre-
    sponding ExitPoint from exit points list
int PathCount(); // returns number of output branches

virtual char * LabelBalanced(); // determines if path label
    needs to be updated by user
virtual boolean Generate( ofstream &linear, Hyperedge
    *previous_edge ); // implements linear form generation
    method
virtual void GenerateIdentifier() { sprintf( identifier, "f%d",
    number_forks++ ); } // generates a textual identifier
virtual void GeneratePathConnections(); // creates mappings of
    semi paths to paths in LinearGenerator object
virtual void Save( FILE *fp ); // implements Save method for
    this hyperedge

static void ResetCounter() { number_forks = 1; } // resets
    global counter of Or Forks
```

protected:

```
boolean AddBranch( flag execute ); // transformation which
    adds a branch to the fork
boolean RotatePaths( flag execute ); // user operation which
    rotates the positions of the or fork branches

Dcn<ExitPoint *> exit_points; // dictionary of branch selec-
    tion strings indexed on node pointers
boolean label_balanced; // flag used to determine whether
    relabelling of paths is necessary
static int number_forks; // global counter of number of or
    fork instances
char *previous_selection; // temporary storage for branch
    selection pointer
```

```
};
```

Class OrJoin - Concrete

The **OrJoin** hyperedge similarly represents or joins. It is responsible for adding and deleting input branches and for deleting itself when no longer needed.

```
class OrJoin : public MultipathEdge {

public:

    OrJoin( boolean collections = TRUE );
    ~OrJoin() {}

    virtual edge_type EdgeType() { return( OR_JOIN ); } // returns
        identifier for subclass type
    virtual void AttachTarget( Node *new_node ); // implements
        error checking behaviour
    virtual boolean Generate( ofstream &linear, Hyperedge
        *previous_edge ); // implements linear form generation
        method
    virtual void GenerateIdentifier() { sprintf( identifier, "j%d",
        number_joins++ ); } // generates a textual identifier
    virtual void GeneratePathConnections(); // creates mappings of
        semi paths to paths in LinearGenerator object

    boolean Perform( transformation trans, execution_flag execute
        ); // executes or validates transformations with code trans
    int PathCount(); // returns number of input branches

    void ConnectForkPath( Hyperedge *edge ); // connects the path
        ending in hyperedge edge as an input path to the or join
    void Delete(); // deletes the join when it becomes unnecessary
    static void ResetCounter() { number_joins = 1; } // resets
        global counter of Or Joins

protected:

    boolean RotatePaths( flag execute ); // user operation which
        rotates the positions of the or fork branches

    static int number_joins; // global counter of number of or
        join instances
```

```
};
```

Class Responsibility - Concrete

The **Responsibility** hyperedge represents a responsibility in the UCM notation. It is responsible for editing its description through a user dialog, for creating and deleting graphical arrows representing component movement and for deleting itself.

```
class DataReference { // data object used by Responsibility

public:

    DataReference( int data, int mode ) { data_store = data;
        access_mode = mode; }
    ~DataReference() {}

    int Data() { return( data_store ); }
    void Data( int data ) { data_store = data; }
    int Access() { return( access_mode ); }
    void Access( int mode ) { access_mode = mode; }

private:

    int data_store, // integer reference to data store stored in
        DataStoreDirectory object
        access_mode; // integer reference to access mode

};

class Responsibility : public Hyperedge {

public:

    Responsibility( boolean collections = TRUE ); // collections
        flag allows suppression of dialog box during file loading
    ~Responsibility();

    virtual edge_type EdgeType() { return( RESPONSIBILITY ); } //
        returns identifier for subclass type
    virtual boolean Generate( ofstream &linear, Hyperedge
        *previous_edge ); // implements linear form generation
        method
    virtual char * AtomIdentifier(); // returns name of responsi-
        bility which is used as an identifier
    virtual boolean IsAtomReference() { return( FALSE ); } //
```


Application of Use Case Maps to System Design with Tool Support

```
    returns fact that it is a responsibility
virtual void Save( FILE *fp ); // implements Save method for
    responsibilities

virtual void AttachSource( Node *newNode ); // perform error
    checking
virtual void AttachTarget( Node *newNode );

boolean DeleteResponsibility( execution_flag execute ); //
    transformation which deletes this responsibility and
    replaces it with an empty point
boolean EditResponsibility( execution_flag execute ); // allows
    user to edit responsibility characteristics through dialog
boolean Perform( transformation trans, execution_flag execute
    ); // executes or validates transformations with code trans

Cltn<DataReference *> * DataReferences() { return( data ); } /
    / returns list of data references
void AddDataReference( int data_id, int access_id ); // adds
    new data store refernce to list
void DeleteDataReference( DataReference *dr ) { data-
    >Delete(dr); delete dr; } // deletes data store reference
boolean ReferencesData( int data_id ); // returns whether a
    certain data store is referenced or not

private:

    Cltn<DataReference *> *data; // list of DataReference objects
    which store references to global data stores

};
```

Class Result - Concrete

The **Result** hyperedge represents the end of a path. It is responsible for connecting the ends of paths to and joins/multipath synchronizations, or joins and stubs. It is also responsible for managing its lists of postconditions and resulting events.

```
class Result : public Hyperedge {

public:

    Result( boolean install = TRUE, boolean collections = TRUE );
    ~Result();
```

Application of Use Case Maps to System Design with Tool Support

```
virtual edge_type EdgeType() { return( RESULT ); } // returns
    identifier for subclass type
virtual void EdgeSelected(); // called when element selected,
    installs textual conditions in main window
virtual void Save( FILE *fp ); // implements Save method for
    Results
virtual boolean Generate( ofstream &linear, Hyperedge
    *previous_edge ); // implements linear form generation
    method
virtual char * AtomIdentifier() { return( identifier ); } //
    returns textual identifier

void AttachSource( Node *newNode ); // perform error checking
void AttachTarget( Node *newNode );
boolean ConnectPath( Hyperedge *trigger_edge, flag execute ); /
    / transformation which attaches this path end as the trig-
    gering element of a waiting place
boolean DisconnectPath( Hyperedge *trigger_edge, flag execute
    ); // transformation which detaches a triggering connection
    if one exists
// transformations which attach the end of path to either an
    existing and/or join or to another path at an empty point, in
    which case an and/or join is created
boolean AndCompose( Hyperedge *edge, flag execute );
boolean OrCompose( Hyperedge *edge, flag execute );
boolean StubCompose( Hyperedge *edge, flag execute ); // trans-
    formation which attaches the end of the result's path as an
    input path to a stub
boolean Perform( transformation trans, execution_flag execute
    ); // executes or validates the transformation with code
    trans
boolean PerformDouble( transformation trans, Hyperedge *edge,
    execution_flag execute ); // executes or validates the dou-
    ble selection transformation with code trans
static void ResetCounter() { number_results = 1; } // resets
    global counter of Results
void SetBound() { bound = TRUE; } // operations on flag denot-
    ing whether path endpoint is bound to a stub output
void SetUnbound() { bound = FALSE; }
boolean IsBound() { return( bound == TRUE ); }
void PathEndName( char *new_name ); // set method for identi-
    fier
```

private:

```
boolean EditPathEndName( execution_flag execute ); // allows
```

Application of Use Case Maps to System Design with Tool Support

```
    user to enter identifier

    ConditionManager condition_manager; // object which manages the
        textual conditions
    char identifier[20]; // user given identifier
    static int number_results; // global count of result hyperedges
    boolean bound; // flag denoting whether endpoint is bound

};
```

Class Start - Concrete

The **Start** hyperedge represents the start point of a path. Its responsibilities include merging the start of a path with another path or a stub thus destroying itself in the process. It is also responsible for connecting a path to another in a triggering arrangement and for managing user input about the arrival process down a path, which is an extension made for the Nortel performance project. Starts are also responsible for managing the lists of its preconditions and triggering events.

```
class Start : public WaitingPlace {

public:

    Start( boolean install = TRUE, boolean collections = TRUE );
    ~Start() {}

    virtual edge_type EdgeType() { return( START ); } // returns
        identifier for subclass type
    virtual void Save( FILE *fp ); // implements Save method for
        Starts
    virtual boolean Generate( ofstream &linear, Hyperedge
        *previous_edge ); // implements linear form generation
        method

    void AttachSource( Node *newNode ); // perform error checking
    void AttachTarget( Node *newNode );
    boolean Merge( Hyperedge *edge, execution_flag execute ); //
        transformation which merges the start of this path with the
        end of another or the output of a stub
    boolean ConnectPath( Hyperedge *trigger_edge, execution_flag
        execute ); // connects start point to a triggering path ele-
        ment
    boolean DeletePath( execution_flag execute ); // deletes the
        entire path if user confirms operation
```

Application of Use Case Maps to System Design with Tool Support

```
void MergeWithResult( Hyperedge *result ); // merges two path
    together, destroying this start
void MergeWithStub( Hyperedge *stub ); // sets the start of
    this path as the output of a stub, destroying this start
boolean Perform( transformation trans, execution_flag execute
    ); // executes or validates the transformation with code
    trans
boolean PerformDouble( transformation trans, Hyperedge *edge,
    execution_flag execute ); // executes or validates the dou-
    ble selection transformation with code trans
static void ResetCounter() { number_starts = 1; } // resets
    global counter of Starts
void SetBound() { bound = TRUE; } // operations on flag denoting
    whether path starting point is bound to a stub input
void SetUnbound() { bound = FALSE; }
boolean IsBound() { return( bound == TRUE ); }
arrival_type ArrivalType() { return( arrival ); } // access
    methods for arrival type
void ArrivalType( arrival_type atype ) { arrival = atype; }
float Input( int index ) { return( input[index-1] ); } //
    access methods for distribution values
void Input( float value, int index ) { input[index-1] = value;
    }
char * ExpertInput() { return( expert ); } // access methods
    for expert input string
void ExpertInput( const char *new_process );
char *Identifier() { return( identifier ); } // returns user
    given identifier

private:

boolean EditPathInitiation( execution_flag execute ); // allows
    user to enter time distribution characteristics through dia-
    log
boolean SamePath( Hyperedge *first, Hyperedge *second ); //
    determines if two hyperedges are part of the same path

arrival_type arrival; // enumerated type for arrival time dis-
    tributions
float input[2]; // values for time distributions
char *expert; // string for expert mode on time distribution
static int number_starts; // global count of start hyperedges
boolean bound; // flag denoting whether endpoint is bound

};
```

Class Stub - Concrete

The **Stub** hyperedge represents both static and dynamic stubs in UCM's. It is responsible for managing its submaps, either a single one for static stubs or a collection of submaps for dynamic stubs. This includes installing plug-in maps as submaps of the stub, viewing the submaps in the editor window and allowing the user to specify bindings between a stub and its submaps. Stubs also manage lists of preconditions and postconditions.

```
typedef enum { STATIC, DYNAMIC } stub_type;
typedef enum { PATH_ENTRY, PATH_EXIT } end_type;

class StubBinding { // data class used by Stub to store bindings
    between input/output points of stubs and path endpoints in
    submap

public:

    StubBinding( Node *node, Hyperedge *endpoint = NULL )
    { boundary_node = node; path_endpoint = endpoint; }

    Node *boundary_node; // input or output node of stub
    Hyperedge *path_endpoint; // path start or end in stub
    int path_identifier; // integer identifier of stub inputs, out-
        puts

};

class PluginBinding { // data class used by Stub which stores the
    plug-in bindings for a single plug-in

public:

    PluginBinding( Stub *stub, Map *pi );
    ~PluginBinding();

    void AddNode( Node *new_node, end_type etype ); // adds a node
        to the bindings list
    void RemoveNode( Node *node, end_type etype ); // removes a
        node from the bindings list

    Map *plugin; // the plug-in map for which the bindings are
        made
    Dcn<StubBinding *> entry_bindings, exit_bindings; // list of
```

Application of Use Case Maps to System Design with Tool Support

```
        bindings for input, output points

};

class Stub : public WaitingPlace {

public:

    Stub( boolean collections = TRUE ); // constructor, flag
        allows suppression of user dialog during file loading
    ~Stub();

    virtual edge_type EdgeType() { return( STUB ); } // returns
        identifier for subclass type
    virtual void EdgeSelected(); // installs textual conditions in
        main window list
    virtual void Save( FILE *fp ); // implements polymorphic Save
        method for stubs
    virtual boolean Generate( ofstream &linear, Hyperedge
        *previous_edge ); // implements polymorphic linear form gen-
        eration method for stubs
    virtual void GeneratePathConnections(); // generates mappings
        of semi-paths to paths in Lineargenerator object
    virtual char * AtomIdentifier() { return( stub_label ); } //
        returns user given textual identifier for stub

    virtual void AttachSource( Node *new_node ); // adds new source
        node to plug-in bindings list
    virtual void AttachTarget( Node *new_node ); // adds new target
        node to plug-in bindings list
    virtual void DetachSource( Node *node ); // removes source node
        from plug-in bindings list
    virtual void DetachTarget( Node *node ); // removes target node
        from plug-in bindings list

    boolean DeleteStub( flag execute ); // deletes the stub from
        the path
    boolean ExpandStub( flag execute ); // allows user to create a
        new submap for stub
    boolean ViewSubmap( flag execute ); // displays one of the
        stub's submaps, displays selection dialog if necessary
    boolean EditStubLabel( flag execute ); // allows user to edit
        the stub label
    boolean InstallExistingPlugin( flag execute ); // installs a
        previously created plug-in as a submap of the stub
    boolean RemoveSubmap( flag execute ); // allows user to remove
```

Application of Use Case Maps to System Design with Tool Support

```
    one of the stub's submaps, displays selection dialog if necessary
boolean BindPlugin( flag execute ); // allows user to specify
    stub bindings through a dialogs, displays selection dialog
    if necessary
boolean CopyExistingPlugin( flag execute ); // installs a copy
    of a previously created plug-in as a submap of the stub

int PathCount() { return( source->Size() ); } // returns number
    of input paths
boolean HasSubmap() { return( submaps.Size() != 0 ); } //
    returns flag if any submaps exist
Cltn<Map *> * Submaps() { return( &submaps ); } // returns list
    of submaps
void ViewPlugin(); // displays a plug-in map
boolean Perform( transformation trans, execution_flag execute
    ); // validates or performs the single selection transforma-
    tion with code trans
boolean PerformDouble( transformation trans, Hyperedge *edge,
    execution_flag execute ); // validates or performs the dou-
    ble selection transformation with code trans
void Install(); // invokes ConnectionManager functions for
    installing a stub
void SetName( const char *new_label ); // set method for stub
    label
char * StubName() { return( stub_label ); } // returns user
    given stub label
void SetType( stub_type type ) { stype = type; } // set method
    for stub type( static/dynamic )
void SetSelectedMap( Map *selected_map ); // installs plug-in
    map selected by user in the editor window
void InstallNewSubmap( Map *new_map ); // installs user
    selected submap as a plug-in to stub if it is not already one
void InstallSubmapImage( Map *new_map ); //installs copy of
    user selected submap as a plug-in to stub
void RemovePlugin( Map *map ); // removes a map from the list
    of plug-ins of the stub
void InitiatePathBinding( Map *plugin ); // initiates process
    of binding entry/exit points for a given plug-in
void ConcludePathBinding(); // terminates process of binding
    entry/exit points for a given plug-in
void UpdateBindingsDisplay(); // updates the display of the
    current bindings in the bindings dialog
void BindEntryPoints( int stub_entry, int plugin_entry ); //
    records binding of stub input point and plug-in start point
void BindExitPoints( int stub_exit, int plugin_exit ); //
```

Application of Use Case Maps to System Design with Tool Support

```
    records binding of stub output point and plug-in end point
void BreakBinding( int binding ); // breaks the binding
    selected by the user
```

```
Start * SubmapPathStart( Empty *empty ); // returns the start
    point in the submap bound to the input point of the stub for
    static stubs, used in performance simulations
Result * SubmapPathEnd( Empty *empty ); // returns the path end
    point in the submap bound to the output point to the stub for
    static stubs, used in performance simulations
```

private:

```
boolean PathStartValid( Start *start ); // returns whether a
    start object stored in the bindings list still exists, error
    checking method
boolean PathEndValid( Result *result ); // returns whether a
    result object stored in the bindings list still exists,
    error checking method
void ValidatePathBindings( PluginBinding *pb ); // validates
    if the information stored in bindings list is still valid
boolean IgnoredEndingsExist(); // returns flag signifying if
    any start/end points in current submap are unbound

stub_type stype; // type of stub, static/dynamic
char stub_label[20]; // user given label for stub
Cltn<Map *> submaps; // list of submaps for this stub
ConditionManager condition_manager; // object which manages
    the textual conditions
Dcn<PluginBinding *> plugin_bindings; // list of plug-in bind-
    ings for all submaps
PluginBinding *current_binding; // the plug-in binding that is
    being edited

static Start *plugin_starts[20]; // global storage for stubs
    for start and end point pointers
static Result *plugin_results[20];
static int number_starts, number_results; // numbers of start
    and end points

};
```

Class Synchronization - Concrete

The **Synchronization** hyperedge is used to represent and forks, and joins and mul-

Application of Use Case Maps to System Design with Tool Support

tipath synchronizations. A Synchronization hyperedge acts as one of the above three elements depending on its connections. A single hyperedge was chosen to represent these UCM elements as the particular structure may change as connections are made and broken. A Synchronization hyperedge is responsible for adding and deleting branches to its input and output. It cannot be deleted by the user directly but is responsible for deleting itself when it has only one input and one output branch thereby becoming superfluous.

```
typedef enum { FORK, JOIN, MULTIPATH } synch_type;

class EntryPoint { // data class used by Synchronization to store
    timer and trigger path information

public:

    EntryPoint( Node *entry, boolean timer_exists = FALSE, Node
        *trigger = NULL )
    { entry_point = entry; timer = timer_exists; trigger_path =
        trigger; }

    boolean timer; // flag signifying presence of timer
    Node *entry_point; // input node of synchronization
    Node *trigger_path; // first node of triggered path

};

class Synchronization : public MultipathEdge {

public:

    Synchronization( synch_type type, boolean collections = TRUE );
    ~Synchronization() {}

    edge_type EdgeType() { return( SYNCHRONIZATION ); } //
        returns identifier of subclass type
    synch_type SynchronizationType() { return( stype ); } //
        returns type of synchronization

    virtual void AttachSource( Node *new_node ); // node methods
        keep track of current synchronization type
    virtual void AttachTarget( Node *new_node );
    virtual void DetachSource( Node *node );
    virtual void DetachTarget( Node *node );

    int PathCount(); // returns maximum of input/output counts
    int InputCount() { return( source->Size() ); } // returns num-
```

Application of Use Case Maps to System Design with Tool Support

```
    ber of input paths
int OutputCount() { return( target->Size() ); } // returns num-
    ber of output paths

void ConnectForkPath( Hyperedge *edge ); // connect a path as
    an input to this synchronization
void ExtendFork(); // creates second output branch of a fork
    when it is first placed in a path
void Delete(); // deletes the synchronization from the path
    when it is no longer needed

boolean Perform( transformation trans, execution_flag execute
    ); // validates or performs the single selection transfor-
    mation with code trans
boolean PerformDouble( transformation trans, Hyperedge *edge,
    execution_flag execute ); // validates or performs the dou-
    ble selection transformation with code trans

virtual char * LabelBalanced(); // determines if path label
    needs to be updated by user
virtual void Save( FILE *fp ); // implements Save method for
    this hyperedge
virtual boolean Generate( ofstream &linear, Hyperedge
    *previous_edge ); // implements linear form generation
    method
virtual void GenerateIdentifier() { sprintf( identifier,
    "syn%d", number_synchronizations++ ); } // generates a tex-
    tual identifier
virtual void GeneratePathConnections(); // creates mappings of
    semi paths to paths in LinearGenerator object
static void ResetCounter() { number_synchronizations = 1; } //
    resets global counter of Synchronizations
```

protected:

```
boolean AddBranch( flag execute ); // adds a new output branch
    to the synchronization
boolean RotateInputPaths( flag execute ); // allows user to
    rotate the position of the input paths
boolean RotateOutputPaths( flag execute ); // allows user to
    rotate the position of the output paths

boolean AddIncomingSynch( Hyperedge *edge, flag execute ); //
    allows user to add a new input path to the synchronization
boolean AddTimer( Hyperedge *edge, flag execute ); // allows
    user to add a timer on an input branch
```

Application of Use Case Maps to System Design with Tool Support

```
boolean RemoveTimer( Hyperedge *edge, flag execute ); // allows
    user to remove a timer from an input branch if it exists

Dcn<EntryPoint *> entry_points; // list of timer information
    for input points
synch_type stype; // type of synchronization
boolean label_balanced; // flag used to determine whether rela-
    belling of paths is necessary
static int number_synchronizations; // global counter of number
    of synchronization instances

};
```

Class Timer - Concrete

The **Timer** and **Wait** hyperedges represent timers and waiting places in UCM's. They are responsible for connecting triggering path elements of other paths(either the ends of paths or empty segments) to the path on which they reside and for deleting themselves when the user requests.

```
class Timer : public WaitingPlace {

public:

    Timer( boolean collections = TRUE );
    ~Timer() {}

    edge_type EdgeType() { return( TIMER ); } // returns identi-
        fier for hyperedge type
    boolean DeleteTimer( flag execute ); // transformation which
        deletes the timer
    boolean Perform( transformation trans, execution_flag execute
        ); // validates or performs the single selection transforma-
        tion with code trans
    boolean PerformDouble( transformation trans, Hyperedge *edge,
        execution_flag execute ); // validates or performs the dou-
        ble selection transformation with code trans

};
```

Class Timestamp - Concrete

The **Timestamp** hyperedge represents timestamp points in ASN's which are

Application of Use Case Maps to System Design with Tool Support

needed for the performance extensions to record places on UCM paths where the time is to be recorded during simulations. It is responsible for deleting itself.

```
typedef enum { PREVIOUS, NEXT } timestamp_reference; // enumer-
    ated data type which refers to whether timestamps references
    previous or next element

class Timestamp : public Hyperedge {

public:

    Timestamp( boolean collections = TRUE );
    ~Timestamp();

    virtual edge_type EdgeType() { return( TIMESTAMP ); } //
        returns identifier for hyperedge type
    virtual boolean Generate( ofstream &, Hyperedge * ) { return(
        CONTINUE ); } // default ( empty ) generation method
    virtual char * AtomIdentifier() { return( NULL ); } // default
        ( empty ) identifier access method
    boolean Perform( transformation trans, execution_flag execute
        );// validates or performs the single selection transforma-
        tion with code trans
    boolean PerformDouble( transformation trans, Hyperedge *edge,
        execution_flag execute ); // validates or performs the dou-
        ble selection transformation with code trans
    virtual void Save( FILE *fp ); // implements Save method for
        timestamp

    char *Name() { return( name ); } // access method for Name
    void Name( const char *new_name ); // set method for name
    timestamp_reference Reference() { return( reference ); } //
        access method for reference
    void Reference( timestamp_reference tr ) { reference = tr; } /
        / set method for reference

private:

    boolean DeleteTimestamp( execution_flag execute ); // transfor-
        mation which deletes the timestamp point
    boolean EditTimestamp( execution_flag execute ); // allows user
        to edit timestamp point characteristics
    boolean ViewResponseTimes( execution_flag execute ); // allows
        user to view all response time requirements involving this
        timestamp point
    boolean CreateResponseTime( Hyperedge *ts, flag execute ); //
```

Application of Use Case Maps to System Design with Tool Support

```
        allows user to create a response time requirement involving
        this timestamp point
    boolean EditResponseTime( Hyperedge *ts, flag execute ); //
        allows user to edit a response time requirement involving
        this timestamp point
    boolean DeleteResponseTime( Hyperedge *ts, flag execute ); //
        allows user to delete a response time requirement involving
        this timestamp point

    char name[20]; // user given name for timestamp
    timestamp_reference reference; // data type which refers to
        element referenced by timestamp

};
```

Class Wait - Concrete

```
class Wait : public WaitingPlace {

public:

    Wait( boolean collections = TRUE );
    ~Wait() {}

    edge_type EdgeType() { return( WAIT ); } // returns identifier
        for hyperedge type
    boolean DeleteWait( flag execute ); // transformation which
        deletes the wait
    boolean Perform( transformation trans, execution_flag execute
        ); // validates or performs the single selection transforma-
        tion with code trans
    boolean PerformDouble( transformation trans, Hyperedge *edge,
        execution_flag execute ); // validates or performs the dou-
        ble selection transformation with code trans

};
```

Figure subclasses

Concrete classes:

OrFigure	OrNullFigure	PointFigure
ResponsibilityFigure	StubFigure	

SynchronizationFigure SynchNullFigure
TimestampFigure

Abstract classes:

Figure HyperedgeFigure

As mentioned before the figure subclasses manage the graphical display of their corresponding hyperedges or are placeholders for spline interpolation as in the case of the two null figure classes which have no corresponding hyperedge. Class Figure - Abstract class Figure {

public:

```
Figure();  
virtual ~Figure();
```

```
Path *GetPath() { return( path ); } // access method for path  
object of which figure is part  
int GetNumber() { return( figure_number ); } // returns inte-  
ger identifier for object
```

```
virtual void GetPosition( float& f_rx, float& f_ry )=0; //  
polymorphic method which returns coordinates at which figure  
is drawn
```

```
virtual void SetPath( Path *new_path, boolean collections =  
TRUE ); // polymorphic method for changing the figure's path  
object
```

```
virtual void PathChanged(); // polymorphic method for notifying  
dependent splines that they are out of date
```

```
virtual void Save( FILE *fp )=0; // implements Save method for  
this figure type
```

```
static void ResetFigureCount() { number_figures = 0; } //  
resets global count of figure objects
```

protected:

```
int figure_number, load_number; // current and past integer  
identifiers for object
```

```
Path *path; // path object of which this figure is a part
```

```
static int number_figures; // global count of number of figure  
objects
```

```
};
```

Class HyperedgeFigure - Abstract

```
class HyperedgeFigure : public Figure {

#ifdef SOLARIS
    friend ostream& operator<<( ostream&, const HyperedgeFigure *
        );
#endif

public:

    HyperedgeFigure( Hyperedge *edge ); // constructor, stores cor-
        responding hyperedge object
    virtual ~HyperedgeFigure();

    Hyperedge *Edge() { return( dependent_edge ); } // returns
        corresponding hyperedge object
    HyperedgeFigure *DependentFigure() { return( dependent_figure
        ); } // access methods for dependent figure object
    void DependentFigure( HyperedgeFigure *new_dependent ) {
        dependent_figure = new_dependent; }

    void SetActualPosition( float f_nx, float f_ny ) { fX = f_nx;
        fY = f_ny; } // access method for internal position coordi-
        nates
    virtual void GetPosition( float& f_rx, float& f_ry ); //
        implements polymorphic position retrieval method for hyper-
        edge figures
    virtual void SetPosition( float f_nx, float f_ny, boolean limit
        = TRUE, boolean enclose = TRUE, // implements polymorphic
        position setting method
        boolean dependent_update = TRUE, boolean interpolate = TRUE
        ); // for hyperedge figures
    virtual void Draw( Presentation *ppr )=0; // polymorphic func-
        tion for drawing figure on screen or postscript file
    virtual boolean PointInside( float XCoord, float YCoord ); //
        polymorphic hit detection method for mouse clicks
    virtual boolean IsResponsibility(); // returns if figure is
        that of a responsibility

    void SetSelected() { selected = TRUE; } // access and reset
```

Application of Use Case Maps to System Design with Tool Support

```
    methods for selected flag
void ResetSelected() { selected = FALSE; }
boolean IsSelected() { return( selected == TRUE ); }

Component *GetEnclosingComponent() { return( component ); } /
/ access and reset methods for enclosing component
void SetEnclosingComponent( Component *encl_component ) { com-
    ponent = encl_component; }
void ResetEnclosingComponent();

protected:

float fX, fY; // internal position coordinates
Hyperedge *dependent_edge; // corresponding hyperedge object
HyperedgeFigure *dependent_figure; // dependent figure object
    for path element connections
Component *component; // the enclosing component of the figure
boolean selected; // flag signifying whether figure is
    selected or not

};
```

Class OrFigure - Concrete

The **OrFigure** class is used to display or forks and or joins. It is responsible for displaying itself in the appropriate manner depending on the number of input paths and output paths they may have and for giving appropriate coordinates to the placeholder null figures which mark the starts and ends of paths.

```
class OrFigure : public HyperedgeFigure {

public:

    OrFigure( Hyperedge *edge ); // constructor, contains pointer
        of dependent hyperedge
    ~OrFigure();

    virtual void SetPosition( float f_nx, float f_ny, boolean limit
        = TRUE, boolean enclose = TRUE, // implementation of poly-
        morphic position setting method
        boolean dependent_update = TRUE, boolean interpolate = TRUE
        ); // sets position and notifies dependent paths
    virtual void PathChanged(); // notifies dependent paths of
        position change
    void ChangeOrientation(); // changes the orientation from for-
```


Application of Use Case Maps to System Design with Tool Support

```
ward to backward and vice versa
virtual void Draw( Presentation *ppr ); // implementation of
Draw method for or figures
virtual void Save( FILE *fp ); // implementation of Save method

int AddDependent( OrNullFigure *new_null, boolean collections =
TRUE ); // adds a terminating null figure of a path as a
dependent, returns path order
void RemoveDependent( OrNullFigure *null ); // removes a null
figure as a dependent
void GetNullPosition( int ordinal, float& f_rx, float& f_ry );
// returns the position at which a null figure with path
ordering ordinal should be drawn
void RotatePaths(); // rotates the position of the input/output
paths

protected:

    Cltn<OrNullFigure *> *fork_paths; // list of dependent null
    figures ( terminations of paths entering or leaving join/
    fork )
    figure_orientation orientation; // the orientation of the fig-
    ure ( forward/backward )
    boolean deleted; // flag set when figure is deleted

};
```

Class OrNullFigure - Concrete

Objects of the Or NullFigure class are used as placeholders to mark the starts and ends of path segments entering and leaving or forks and or joins so that proper splines can be interpolated. They are registered as dependent objects of their corresponding OrFigure objects. They have no position coordinates but rather query their parent for coordinates. They are aware of their ordering in the fork/join, that is whether they are the first, second or nth path leaving or entering the fork/join.

```
class OrNullFigure : public Figure {

public:

    OrNullFigure( OrFigure *or_figure, Path *new_path, boolean col-
    lections = TRUE ); // constructor, includes pointer to par-
    ent figure
    ~OrNullFigure();
```

Application of Use Case Maps to System Design with Tool Support

```
virtual void GetPosition( float& f_rx, float& f_ry ); //
    returns position at which spline should begin/end, queries
    parent figure

int Ordinal() { return( ordinal ); } // access methods for path
    ordinal value
void Ordinal( int new_ordinal ) { ordinal = new_ordinal; }

virtual void Save( FILE *fp ); // implements Save method

private:

    OrFigure *parent_figure; // the parent OrFigure to which this
        null figure is registered
    int ordinal; // path ordinal value, i.e. index of path in fork/
        join

};
```

Class PointFigure - Concrete

The PointFigure object is used to display the graphical figures of most simple single input, single output hyperedges such as Start, Result, Wait, Timer, Empty. A single class is used to avoid class explosion of similar classes.

```
typedef enum { EMPTY_FIG, START_FIG, WAIT_FIG, RESULT_FIG,
    TIMER_FIG } point_type;

class PointFigure : public HyperedgeFigure {

public:

    PointFigure( Hyperedge *edge, point_type type ); // construc-
        tor, includes data type for represented element
    ~PointFigure();
    virtual void Draw( Presentation *ppr ); // implements Draw
        method
    virtual void Save( FILE *fp ); // implements Save method

private:

    point_type ptype; // the type of hyperedge the figure repre-
        sents
```

```
};
```

Class ResponsibilityFigure - Concrete

The **ResponsibilityFigure** class displays the graphical display of a responsibility along a path and uses the **DynamicArrow** class to display arrows relating to structural dynamics. It manages the characteristics of a responsibility such as name, description, execution sequence which are initially edited with a dialog box invoked by its constructor.

```
class ResponsibilityFigure : public HyperedgeFigure {  
  
public:  
  
    ResponsibilityFigure( Hyperedge *edge, boolean collections =  
        TRUE );  
    ~ResponsibilityFigure();  
  
    virtual void Draw( Presentation *ppr ); // implements Draw  
        method  
    virtual boolean IsResponsibility() { return( TRUE ); }  
    virtual void Generate( ofstream &linear ); // implements linear  
        form generation method  
    virtual void Save( FILE *fp ); // implements Save method  
  
    char * Name() { return( name ); } // access methods for name,  
        description, and execution sequence  
    void Name( const char *new_name );  
    char * Description() { return( description ); }  
    void Description( const char *new_description );  
    char * ExecutionSequence() { return( execution_sequence ); }  
    void ExecutionSequence( const char *new_es );  
  
    boolean HasDynarrow(); // returns flag signifying if dynamic  
        arrow present  
    void SetDynamicArrow( DynamicArrow *new_arrow ) { dynamic_arrow  
        = new_arrow; } // access methods for dynamic arrow  
    DynamicArrow * GetDynamicArrow() { return( dynamic_arrow ); }  
  
    void Highlight() { highlighted = TRUE; } // set, reset methods  
        for highlight flag  
    void Unhighlight() { highlighted = FALSE; }  
  
    void Direction( etResponsibility_direction erdNew_direction ) {  
        erdDirection = erdNew_direction; } // access methods for  
        drawing direction
```

Application of Use Case Maps to System Design with Tool Support

```
    etResponsibility_direction Direction() { return( erdDirection
    ); }

private:

    char name[32], *description, *execution_sequence; // responsi-
        bility name, description, and execution sequence
    etResponsibility_direction erdDirection; // position of label
    boolean highlighted; // flag set when figure is to be high-
        lighted
    DynamicArrow *dynamic_arrow; // pointer to possible dynamic
        pointer

};
```

Class Dynamic Arrow*

```
typedef enum {MOVE, MOVE_STAY, CREATE, DESTROY, COPY}
    etDynamicArrow_type;
typedef enum {ARROW_UP, ARROW_DOWN, ARROW_RIGHT, ARROW_LEFT}
    etDynamicArrow_direction;
typedef enum {INTO, OUTOF} etDynamicArrow_position;

class DynamicArrow {

public:

    DynamicArrow( etDynamicArrow_type edatNew_type,
        etDynamicArrow_direction edadNew_direction,
        etDynamicArrow_position edapNew_position,
        ResponsibilityFigure *figure,
        const char *pool = NULL );

    ~DynamicArrow();

    virtual void Draw( Presentation *ppr ); // implements Draw
        method

    void SetupArrow(); // sets up the arrows corrdinates according
        to user selctions of characteristics
    float GetLength(); // access methods for length
    void SetLength( float fNew_length );

    // access methods for pointer attributes
    void SetAttributes( etDynamicArrow_type edatNew_type,
```

Application of Use Case Maps to System Design with Tool Support

```
        etDynamicArrow_direction edadNew_direction,
        etDynamicArrow_position edapNew_position );

void GetAttributes( etDynamicArrow_type &edatNew_type,
        etDynamicArrow_direction &edadNew_direction,
        etDynamicArrow_position &edapNew_position );

const char * SourcePool() { return( source_pool ); } // access
        methods for source pool name
void SourcePool( const char *new_pool ) { strncpy( source_pool,
        new_pool, 19 ); }

void Generate( ofstream &linear ); // implements linear form
        generation method
void Save( FILE *fp ); // implements Save method

private:

char * ActionDescription( etDynamicArrow_type type ); //
        returns textual description of arrow action

ResponsibilityFigure *figure; // the parent responsibility
        figure object for this dynamic pointer
float sX, sY, eX, eY; // the starting and ending coordinates of
        the arrow
etDynamicArrow_type edatType; // type of arrow
etDynamicArrow_direction edadDirection; // direction of arrow
etDynamicArrow_position edapPosition; // position of arrow
char source_pool[20]; // the name of the pool from which the
        component came
float fLength; // length of arrow

};
```

Class StubFigure

The **StubFigure** class performs a similar function to the **OrFigure** and **SynchronizationFigure** objects in displaying a graphical figure for stubs which may have many inputs and outputs. It however does not need placeholder figures. It contains lists of dependent paths which may start or end at the stub.

```
class StubFigure : public HyperedgeFigure {

public:
```

Application of Use Case Maps to System Design with Tool Support

```
StubFigure( Hyperedge *edge, boolean collections = TRUE );
~StubFigure();

Cltn<Path *> *GetCurves() { return( stub_paths ); } // returns
    list of paths which start or terminate at the stub
void ReplaceDependentPath( Path *new_path, Path *old_path ); /
    / replaces a dependent path with a new one
virtual void SetPosition( float f_nx, float f_ny, boolean limit
    = TRUE, boolean enclose = TRUE, // sets position, notifies
    dependent paths
    boolean dependent_update = TRUE, boolean interpolate = TRUE
    );
virtual void SetPath( Path *new_path, boolean collections =
    TRUE ); // installs the path in the stub paths list
virtual boolean PointInside( float XCoord, float YCoord ); //
    implements oplymorphic hit detection method
virtual void PathChanged(); // notifies dependent paths of
    position change

void Draw( Presentation *ppr ); // implements Draw method
void Save( FILE *fp ); // implements Save method

private:

    Cltn<Path *> *stub_paths; // list of paths which start or ter-
        minate at the stub

};
```

Class SynchronizationFigure - Concrete

The **SynchronizationFigure** class performs the same functions as **OrFigure** but for multipath synchronization hyperedges. Both are responsible for notifying their component placeholder figures when they are moved.

```
class SynchronizationFigure : public HyperedgeFigure {

public:

    SynchronizationFigure( Hyperedge *edge );
    ~SynchronizationFigure();

    virtual void SetPosition( float f_nx, float f_ny, boolean limit
        = TRUE, boolean enclose = TRUE,
```

Application of Use Case Maps to System Design with Tool Support

```
    boolean dependent_update = TRUE, boolean interpolate = TRUE
    ); // sets position, notifies dependent paths
virtual void PathChanged(); // notifies dependent paths of
    position change
void ChangeOrientation(); // changes the orientation from for-
    ward to backward and vice versa
virtual void Draw( Presentation *ppr ); // implements Draw
    method
virtual void Save( FILE *fp ); // implements Save method
virtual boolean PointInside( float XCoord, float YCoord ); //
    implements polymorphic hit detection method for synchroniza-
    tions

int AddDependent( SynchNullFigure *new_null, path_direction
    dir, boolean collections = TRUE ); // add/remove dependent
    placeholder figures as dependents
void RemoveDependent( SynchNullFigure *null, path_direction dir
    );
void GetNullPosition( path_direction dir, int ordinal, float&
    f_rx, float& f_ry );
// returns the position at which a null figure with path order-
    ing ordinal and input/output flag should be drawn
void RotatePaths( path_direction dir ); // rotates the position
    of the input/output paths based on flag dir

private:

    Cltn<SynchNullFigure *> *input_paths; // collection of place-
        holder figures for input paths
    Cltn<SynchNullFigure *> *output_paths; // collection of place-
        holder figures for output paths
    figure_orientation orientation; // the orientation of the fig-
        ure, forward/backward
    float top, bottom; // coordinates for the top and bottom of
        the bar
    boolean deleted; // flag set when element has been deleted

};
```

Class SynchNullFigure - Concrete

Objects of class SynchNullFigure acts as placeholders for the starts and ends of path segments entering and leaving multipath synchronization. They are similar to OrNullFigure objects but have an additional direction variable.

Application of Use Case Maps to System Design with Tool Support

```
class SynchNullFigure : public Figure {

public:

    SynchNullFigure( SynchronizationFigure *mp_figure, Path
        *new_path, path_direction dir, boolean collections = TRUE );
    // constructor, includes pointer to parent figure and flag for
    // direction
    ~SynchNullFigure();

    virtual void GetPosition( float& f_rx, float& f_ry ); //
        returns position at which spline should begin/end, queries
        parent figure
    virtual void Save( FILE *fp ); // implements Save method

    int Ordinal() { return( ordinal ); } // access methods for path
        ordinal value
    void Ordinal( int new_ordinal ) { ordinal = new_ordinal; }

private:

    SynchronizationFigure *parent_figure; // the parent Synchroni-
        zationFigure to which this null figure is registered
    path_direction path_dir; // flag referring to whether path ter-
        minated by this figure enters or leaves a synchronization
    int ordinal; // path ordinal value, i.e. index of path in fork/
        join

};
```

Class TimestampFigure - Concrete

The graphical figure for Timestamps is given its own class **TimestampFigure** and not included in **PointFigure** as timestamp points contains additional state.

```
class TimestampFigure : public HyperedgeFigure {

public:

    TimestampFigure( Hyperedge *edge, boolean collections = TRUE );
    ~TimestampFigure();

    virtual void Draw( Presentation *ppr ); // implements Draw
        method
```


Application of Use Case Maps to System Design with Tool Support

```
virtual boolean PointInside( float XCoord, float YCoord ); //
    implements polymorphic hit detection method
virtual void Save( FILE *fp ); // implements Save method

void Direction( timestamp_direction new_direction ) { direction
    = new_direction; }
timestamp_direction Direction() { return( direction ); } //
    access methods for direction
void Highlight() { highlighted = TRUE; } // set, reset methods
    for highlighted flag
void Unhighlight() { highlighted = FALSE; }

private:

    timestamp_direction direction; // orientation of timestamp
    point
    boolean highlighted; // flag set if figure is to be highlighted
};
```

Map Component classes

Map	Hypergraph	Node
Label	Component*	Path

Class Map

The above classes are all of the components of the **Map** class which forms a container for all of the objects in a use case map. The UCM Navigator is designed to be able to navigate multi-level maps which consist of root maps and possibly many levels of plug-in maps. The Map object provides a container for all objects contained in a UCM. It is responsible for saving its information to file and for generating the linear form from its structure. Both of these responsibilities are implemented by invoking appropriate methods on the maps's component objects.

```
class Map {
```

Application of Use Case Maps to System Design with Tool Support

public:

```
Map( const char *label, map_type type = ROOT_MAP ); // constructor, contains label and map type
~Map();

Hypergraph * MapHypergraph() { return( graph ); } // returns hypergraph object for map
Cltn<HyperedgeFigure *> * Figures() { return( figures ); } // returns list of hyperedge figures
Cltn<Path *> * Paths() { return( paths ); } // returns list of paths
Cltn<Component *> * Components() { return( components ); } // returns list of components
Cltn<ResponsibilityFigure *> * Responsibilities() { return( responsibilities ); } // returns list of responsibilities
Cltn<Label *> * Labels() { return( labels ); } // returns list of path labels
char * MapLabel() { return( map_label ); } // access methods for map label, title and description
void MapLabel( const char *new_label );
char * MapTitle() { return( map_title ); }
void MapTitle( const char *new_title );
char * MapDescription() { return( map_description ); }
void MapDescription( const char *new_description );
Map *Copy( const char *new_label ); // returns a map object which is a copy of itself

boolean IsRootMap() { return( mtype == ROOT_MAP ); } // boolean functions which identify type of map
boolean IsPlugIn() { return( mtype == PLUGIN ); }

void IncrementCount() { reference_count++; } // increments reference count
void DecrementCount(); // decrements count and deletes itself if count is zero

void Save( FILE *fp ); // saves all components of the map, calls Save methods of map elements
void Generate( ofstream &linear ); // generates linear form for all elements of the map, calls Generate methods of map elements
int GetNumber() { return( map_number ); } // returns integer identifier of map
```

private:

```
map_type mtype; // flag for type of map ( root/plugin )
Hypergraph *graph; // the hypergraph of the map
Cltn<HyperedgeFigure *> *figures; // list of hyperedge figures
Cltn<Path *> *paths; // list of paths
Cltn<Component *> *components; // list of components in map
Cltn<ResponsibilityFigure *> *responsibilities;
Cltn<Label *> *labels; // list of path labels in map
char *map_title, *map_description; // user given map title and
    description
char map_label[20]; // user given map name

int map_number, load_number, reference_count; // integer identifiers of map
static int number_maps; // global counter for number of maps

};
```

Class Hypergraph

The **Hypergraph** class forms a container for all of the Hyperedges (described previously) and Nodes which form the basic data structures (hypergraphs) upon which the editor's underlying data structures are built. Hypergraphs are responsible for managing their component elements, that is adding hyperedges and nodes to the hypergraph and removing segments of the hypergraph in a stable manner. The **Node** class forms the connections between the hyperedges in the hypergraph model.

```
class Hypergraph {

public:

    Hypergraph();
    ~Hypergraph();

    Cltn<Hyperedge *> * Hyperedges() { return( edge_pool ); } //
        returns list of hyperedges
    Cltn<Node *> * Nodes() { return( node_pool ); } // returns list
        of nodes
    void PurgeEdge( Hyperedge *edge ); // removes hyperedge edge
        from hypergraph
    void PurgeNode( Node *node ); // removes node from hypergraph
    void PurgeSegmentFrom( Node *start_node, Node *end_node ); //
        removes section of graph between start_node and end_node
    void RegisterEdge( Hyperedge *edge ); // registers an edge as
```

Application of Use Case Maps to System Design with Tool Support

```
    part of the hypergraph
void RegisterNode( Node *node ); // registers a node as part of
    the hypergraph

void TrashEdge( Hyperedge *edge ); // marks hyperedge as
    deleted by placing in deleted list
void TrashNode( Node *node ); // marks node as deleted by plac-
    ing in deleted list
Cltn<Hyperedge *> * EdgeTrash() { return( edge_trash ); } //
    returns list of "deleted" hyperedges
Cltn<Node *> * NodeTrash() { return( node_trash ); } // returns
    list of "deleted" nodes
void EmptyAllTrash(); // permanently deletes all "deleted"
    hyperedge, nodes

void Save( FILE *fp ); // saves the contents of the hypergraph
    to disk, calls Save functions of elements
void Generate( ofstream &linear ); // generates the linear form
    of the hypergraph
void GenerateEdge( Hyperedge *edge, ofstream &linear ); //
    generates the linear form for a hyperedge
void GenerateAtomIdentifiers(); // generates textual identifi-
    ers for hyperedges which need them
void Output(); // debugging method

protected:

void PurgeSubSegment( Node *start_node, Node *end_node ); //
    removes a section of hypergraph

Cltn<Node *> *node_pool, *node_trash; // collections of current
    nodes and deleted nodes
Cltn<Hyperedge *> *edge_pool, *edge_trash; // collections of
    current hyperedges and deleted hyperedges

};
```

Class Node

```
class Node {

public:

    Node( nodeColour newColour ); // constructor, with node colour
```

Application of Use Case Maps to System Design with Tool Support

```
    as a parameter
~Node() {}

void AttachNext( Hyperedge *edge ) { next = edge; } // manipu-
    lation methods for next, previous pointers
void AttachPrevious( Hyperedge *edge ) { previous = edge; }
void DetachNext() { next = NULL; }
void DetachPrevious() { previous = NULL; }

nodeColour GetColour() { return ( colour ); } // returns
    colour of node
boolean NextAttached() { return( next ? TRUE : FALSE ); } //
    boolean functions specifying whether node is attached
boolean PreviousAttached() { return( previous ? TRUE : FALSE );
    }
Hyperedge *NextEdge() { return ( next ); } // returns pointer
    to the next hyperedge
Hyperedge *PreviousEdge() { return ( previous ); } // returns
    pointer to the previous hyperedge
void Save( FILE *fp ); // implements Save method
int GetNumber() { return( node_number ); } // returns integer
    identifier of node

void AtomIdentifier( int identifier ) { atom_identifier = iden-
    tifier; } // access methods for node identifier used in
int AtomIdentifier() { return( atom_identifier ); } // linear
    form generation

static void ResetNodeCount() { number_nodes = 0; } // resets
    global count of nodes

private:

    Hyperedge *next, *previous; // pointers to next and previous
        hyperedges in hypergraph
    nodeColour colour; // node colour attribute

    int node_number, load_number; // integer identifiers for node
    int atom_identifier; // identifier used in linear form genera-
        tion
    static int number_nodes; // global count of number of nodes

};
```

Class Label

Application of Use Case Maps to System Design with Tool Support

The **Label** class encapsulates the text string given to a scenario label. They use reference counting and are responsible for deleting themselves when no longer needed.

```
class Label {

public:

    Label( const char *new_label, boolean install = TRUE ); // constructor, includes initial label
    ~Label();
    const char * TextLabel() const { return( text_label ); } // access methods for path label
    void TextLabel( const char *new_label );

    void IncrementCount() { reference_count++; } // increments reference count
    void DecrementCount(); // decrements reference count, deletes object if zero
    void Save( FILE *fp ); // implements Save method
    int GetNumber() { return( label_number ); } // returns integer identifier

    void SemiPath( int new_number ) { semi_path_number = new_number; } // semi path identifier manipulation methods used in linear form generation
    int SemiPath() { return( semi_path_number ); }
    void SetSemiPathConnected() { semi_path_number = abs(semi_path_number); }
    boolean SemiPathConnected() { return( semi_path_number > 0 ); }

    static void ResetLabelCount() { number_labels = 0; } // resets global count of labels

private:

    char text_label[20]; // user given path label
    int label_number, load_number, // current and past integer identifiers for label object
        reference_count, // reference count of object
        semi_path_number; // identifier of semi-path for linear form generation
    static int number_labels; // global count of number of labels

};
```

Class Component

Application of Use Case Maps to System Design with Tool Support

The **Component** class represents the software objects that exist in use case maps. They may contain child components and map elements which are bound to them. They are responsible for displaying themselves, managing their child components and figures, saving their information to file, and generating the linear form description of their structures and contained map elements. This class existed in the prototype.

```
typedef enum {TEAM, OBJECT, PROCESS, ISR, POOL, AGENT }
    etComponent_type;

class Component {

public:

    Component( float fNew_x, float fNew_y, float fNew_width, float
        fNew_height,
        Component *pbNew_parent = NULL );
    ~Component();

    void SetPosition( float fNew_x, float fNew_y ); // access meth-
        ods for component position
    void GetPosition( float &fReturn_x, float &fReturn_y );

    void SetType( etComponent_type ectNew_type ) { component_type
        = ectNew_type; } // access methods for component type
    etComponent_type GetType() { return( component_type ); }

    void SetAttributes( int iNew_dynamic, int iNew_stack, int
        iNew_protected ); // access methods for component attributes
    void GetAttributes( int &iNew_dynamic, int &iNew_stack, int
        &iNew_protected );

    void SetColour( int iNew_colour ) { iColour = iNew_colour; } /
        / access methods for component colour
    int GetColour() { return( iColour ); }

    float GetHeight(); // access methods for component width,
        height
    float GetWidth();
    void SetHeight( float fNew_height, int iResizing_decoupled = 0
        );
    void SetWidth( float fNew_width, int iResizing_decoupled = 0 );
    boolean PointinBox( float fCheck_x, float fCheck_y ); // hit
        detection method

    void ResetGeometry(); // methods which set and check dimen-
```

Application of Use Case Maps to System Design with Tool Support

```
sions with respect to
void CheckGeometry(); // parent and child components

Component *GetParent() { return( pbParent ); } // returns the
    parent component
int SetParent( Component *pbNew_parent ); // sets a new compo-
    net as parent
void AddChild( Component *pbNew_child ); // adds a component as
    a child
void RemoveChild( Component *pbChild ); // removes a component
    as a child
int GetNumChildren() { return( children.Size()); } // returns
    number of children
Component *GetChild( int iChild_num ); // returns a child com-
    ponent

int IsPassive(); // checks if component is passive and does not
    contain active components
int CheckTypeContainment( etComponent_type ect ); // checks
    type containment validity of type
int CheckTypesDownward( etComponent_type ect );

int IsComponentDescendent( Component *pbComponent ); // deter-
    mines if component is a descendent of this one
Component *FindRoot(); // finds top level parent component
void Draw( Presentation *ppr ); // draws the component on
    screen or in postscript file

char * GetLabel() { return( component_label ); } // returns
    component label
char * ProcessorName() { return( processor_name ); } // access
    methods for processor name
void ProcessorName( const char *new_name ) { strncpy(
    processor_name, new_name, 19 ); }
int GetComponentNumber() { return( component_number ); } //
    returns integer identifier

Component *Copy( Component *pcCopy_parent ); // copies this
    component

void GetAllChildren( Cltn<Component *> &components ); //
    returns list of child components

void Save( FILE *fp ); // saves component's data to file
void Generate( ofstream &linear, boolean components, int level
    = 0 ); // generates linear form for component
```


Application of Use Case Maps to System Design with Tool Support

```
void NotifyDependents(); // notifies dependent objects of size,
    position change

void RegisterDependentFigure( HyperedgeFigure *new_figure ); /
    / adds a figure as a dependent of this component
void PurgeDependentFigure( HyperedgeFigure *dependent ); //
    removes a figure as a dependent of this component
void BindEnclosedFigures();

boolean Actual() { return( actual ); } // access methods for
    actual, anchored flags
void Actual( int act ) { actual = act; }
boolean Anchored() { return( anchored ); }
void Anchored( int anc ) { anchored = anc; }

static void ResetComponentCount() { number_components = 0; } /
    / resets global count of components

private:

char * ComponentType(); // returns string describing componet
    type
char * Indent( char *message, int level ); // prints indented
    line in linear form

const float STACK_OFFSET; // class variables for drawing off-
    sets
const float PROTECT_OFFSET;

float fX, fY, fWidth, fHeight; // current dimensions of compo-
    nent
float fOld_x, fOld_y, fOld_width, fOld_height; // previous
    dimensions of component
Component *pbOld_parent; // the previous parent component

int iColour; // the component's colour
Component *pbParent; // the parent component of this component

Cltn<Component *> children; // the child components of this
    component
Cltn<HyperedgeFigure *> dependent_figures; // the figures that
    are inside the component boundaries

int iDynamic, iStack, iProtected; // flags specifying component
    characteristics
etComponent_type component_type; // enumerated type for compo-
```

```
nent type

int component_number, iLoad_number; // current and past integer
    identifiers
static int number_components; // global count of number of com-
    ponents
static char line[BUFFER_LENGTH]; // global buffer used in lin-
    ear form generation
char component_label[64]; // user given component label
char processor_name[20]; // user given processor name for
    active components
boolean actual, anchored; // flags specifying if components
    are actual or anchored

};
```

Class Path

The **Path** class represent the path segments that make up use case paths. They reference a list of graphical figures which may be hyperedge figures or null figures which are simply place holders and use interpolated spline curves to display themselves. Paths are reinterpolated whenever one or more of their elements changes position.

```
class Path : public BSpline {

public:

    Path( boolean create = TRUE ); // constructor, flag creates
        specifies whether spline is to be created
    ~Path();

    void SetPathEnd( Hyperedge *end_edge ) { path_end = end_edge; }
        // access methods for path start, end
    void SetPathStart( Hyperedge *start_edge ) { path_start =
        start_edge; }
    Hyperedge *GetPathEnd() { return( path_end ); }
    Hyperedge *GetPathStart() { return( path_start ); }

    void AddFigure( Figure *new_figure ); // adds a new element to
        the path
    void PurgeFigure( Figure *figure ); // removes an element from
        the path
    void AddBeforeFigure( Figure *new_figure, Figure *ref_figure );
        // adds new element before reference element
    void AddAfterFigure( Figure *new_figure, Figure *ref_figure );
```

Application of Use Case Maps to System Design with Tool Support

```
// adds new element after reference element
void DeleteEndFigure(); // removes last element of path
void DeleteStartFigure(); // removes first element of path
void GetCurveEnd( float &fx, float &fy ); // returns position
    of spline end
void SetFigures( Cltn<Figure *> *new_figures ) { figures =
    new_figures; } // sets a new list of path elements
Cltn<Figure *> *GetFigures() { return( figures ); } // returns
    current list of path elements
boolean Changed() { return( changed ); } // returns state of
    changed flag
void PathChanged() { changed = TRUE; } // sets changed flag
void AddStartFigure( Figure *new_figure ); // adds new element
    as start of path
void AddEndFigure( Figure *new_figure ); // adds new element as
    end of path
Figure *GetStartFigure() { return( figures->GetFirst() ); } //
    returns first element of path
Figure *GetEndFigure() { return( figures->GetLast() ); } //
    returns last element of path
void Destroy() { destroyed = TRUE; } // manipulation methods
    for destroyed flag
boolean Destroyed() { return( destroyed ); }

void Save( FILE *fp ); // saves list of figures to file
int GetNumber() { return( path_number ); } // returns integer
    identifier

static void ResetCounter() { number_paths = 0; } // resets glo-
    bal counter of paths
```

protected:

```
Hyperedge *path_end; // hyperedge which starts path
Hyperedge *path_start; // hyperedge which ends path

int path_number, load_number; // current, past identifiers for
    path
static int number_paths; // global count of number of paths
boolean destroyed; // flag set if path is to be deleted

};
```

Manager Classes

ConnectionManager, TransformationManager, FileManager,
ComponentManager, LabelManager, LinearGenerator

The above classes are singleton objects that exist in the UCM Navigator to provide necessary functionality. Most are utility classes such as the TransformationManager, FileManager, ComponentManager, and LabelManager.

Class ConnectionManager

The **ConnectionManager** is the largest and most complex object in the design. Its major responsibilities include managing the graphical display of the editor (drawing the display, handling mouse events), managing the display hierarchy of maps when multilevel maps are displayed and performing all of the manipulations of paths that are necessary whenever use case map transformations are performed by the user. It accomplishes this last responsibility by being a large utility class that may be used by any of the hyperedges to perform the graphical part of any transformation. It essentially provides the means for separating the two concerns of a transformation, the hypergraph manipulations and the graphical manipulations that involve adding and removing figures from paths and splitting and joining paths.

```
class ConnectionManager {  
  
public:  
  
    ConnectionManager( TransformationManager *trans_manager );  
    ~ConnectionManager();  
  
    void SetMap( Map *new_map ); // map placement functions  
    void InstallMap( int map_id );  
    void InstallSubmap( Map *submap, Stub *stub );  
    void InstallMapElements( Map *map );  
    void PlaceMap( Map *new_map );  
    Map * CurrentMap() { return( current_map ); }  
    void InstallParent(); // installs parent map  
    void Submaps( Map **plugins, int &num_plugins ); // submap  
        query functions  
    boolean HasSubmaps();  
};
```

Application of Use Case Maps to System Design with Tool Support

```
void Add( Hyperedge *new_edge, Hyperedge *ref_edge ); // adds
    new element to path at position of reference element
void AddBL( Hyperedge *new_edge, Hyperedge *ref_edge ); // adds
    new element to path between reference element and previous
    element
void AddBR( Hyperedge *new_edge, Hyperedge *ref_edge ); // adds
    new element to path between reference element and next ele-
    ment
void AddAfter( Hyperedge *new_edge, Hyperedge *ref_edge ); //
    various positioning functions
void AddBetween( Hyperedge *new_edge, Hyperedge *left_edge,
    Hyperedge *right_edge );
void AddAtPosition( Hyperedge *new_edge, Hyperedge *ref_edge,
    Hyperedge *next_edge );
void AddRight( Hyperedge *new_edge, Hyperedge *ref_edge );
void AddRight( Figure *new_figure, Figure *ref_figure );
void AddLeft( Hyperedge *new_edge, Hyperedge *ref_edge );
void AddFirst( Hyperedge *edge );
void AddFirstToRight( Hyperedge *new_edge, Hyperedge *ref_edge
    );

void Connect( Hyperedge *edge1, Hyperedge *edge2 ); // connect,
    disconnect two elements
void Disconnect( Hyperedge *edge1, Hyperedge *edge2 );

void CreateStub( Stub *stub ); // stub manipulation functions
void DeleteStub( Stub *stub );
void CreateStubJoin( Hyperedge *edge, Hyperedge *stub );
void DisconnectStubJoin( Hyperedge *empty, Hyperedge *stub,
    Hyperedge *result );
void DisconnectStubFork( Hyperedge *empty, Hyperedge *stub,
    Hyperedge *start );

void CreateNewPath( float XCoord, float YCoord ); // path cre-
    ation functions
void ExtendPath( float XCoord, float YCoord );
void CreateNewSegment();

void SplitPaths( Hyperedge *end_edge, Hyperedge *start_edge );
    // splits path into two
void JoinPaths( Hyperedge *end_edge, Hyperedge *start_edge ); /
    / joins a pair of paths

void CreateAndFork( Synchronization *synch, Empty *empty ); //
    creates graphical elements of and fork
```

Application of Use Case Maps to System Design with Tool Support

```
void AddAndForkBranch( Synchronization *synch, Empty *empty,
    Result *result ); // performs graphical manipulations for
    adding or fork branch
void CreateAndJoin( Synchronization *synch, Empty *empty ); //
    creates graphical elements of and join
void AddAndJoinBranch( Synchronization *synch, Empty *empty );
    // performs graphical manipulations for adding and join
    branch

void CreateOrFork( OrFork *fork, Empty *empty ); // creates
    graphical elements of or fork
void AddOrForkBranch( OrFork *fork, Empty *empty ); // performs
    graphical manipulations for adding or fork branch
void CreateOrJoin( OrJoin *join, Empty *empty ); // creates
    graphical elements of or join
void AddOrJoinBranch( OrJoin *join, Empty *empty ); // performs
    graphical manipulations for adding or join branch
void DisconnectJoin( Hyperedge *empty, Hyperedge *result ); //
    performs graphical manipulations for disconnecting joins

void DeleteEndNull( Hyperedge *edge ); // deletes the end ele-
    ment of path
void DeleteStartNull( Hyperedge *edge ); // deletes the start
    element of path

void TransApplicableTo( Hyperedge *edge ); // determines appli-
    cable transformations for hyperedge
void TransApplicableToPair( Hyperedge *edge1, Hyperedge *edge2
    ); // determines applicable transformations for pair of
    hyperedge

HyperedgeFigure *FindFigure( float XCoord, float YCoord ); //
    performs hit detection for hyperedge figures
void HandleDoubleClick( float XCoord, float YCoord ); // per-
    forms handling of double clicks in the editing area

void RegisterMap( Map *new_map ); // registration and deregis-
    tration functions for maps, plug-ins, figures, paths, and
    responsibilities
void PurgeMap( Map *map );
void RegisterStubExpansion( Map *new_plugin );
void PurgeStubExpansion( Map *plugin );
void RegisterFigure( HyperedgeFigure *new_figure );
void PurgeFigure( HyperedgeFigure *figure );
void RegisterPath( Path *new_path );
void PurgePath( Path *path );
```

Application of Use Case Maps to System Design with Tool Support

```
void RegisterResponsibility( ResponsibilityFigure
    *new_responsibility );
void PurgeResponsibility( ResponsibilityFigure *responsibility
    );

Cltn<ResponsibilityFigure *> *ResponsibilityList() { return(
    responsibility_pool ); }
Cltn<Map *> * Maps() { return( map_pool ); }
void AddMap( Map *new_map ) { map_pool->Add( new_map ); }
Cltn<HyperedgeFigure *> * Figures() { return( figure_pool ); }
TransformationManager *GetTransformationManager() { return(
    trans_manager ); }
void Draw( Presentation *ppr ); // draws screen
void ClearDisplay();
Path *CurrentPath() { return( current_path ); }
void SetPath( Path *path );
void Update();
void SetActive( HyperedgeFigure *figure );
void ResetCurrentPath() { current_path = NULL; }

private:

    void PropagateForward( Hyperedge *split_edge, Path *new_path,
        Path *old_path, Label *new_label = NULL );
    void PropagateBackward( Hyperedge *split_edge, Path *new_path,
        Path *old_path, Label *new_label = NULL );
    TransformationManager *trans_manager;
    Cltn<Map *> *map_pool; // lists of maps, figures, pools, and
        responsibilities
    Cltn<HyperedgeFigure *> *figure_pool;
    Cltn<Path *> *path_pool;
    Cltn<ResponsibilityFigure *> *responsibility_pool;
    const float XSpacing, YSpacing;
    Map *current_map;
    Map *parent_map;
    HyperedgeFigure *active_figure; // current figure pointer
    Path *current_path; // current path pointer
    Cltn<Hyperedge *> *edges;
    hierarchy_element hierarchy[MAXIMUM_HIERARCHY_LEVELS];
    int decomposition_levels;

};
```

Class TransformationManager

Application of Use Case Maps to System Design with Tool Support

The **TransformationManager** is a utility object used by the editor to store the transformations that individual hyperedges can perform and to validate whether they can be performed at run-time in a particular situation by calling all of the appropriate validation methods.

```
class TransformationManager
{
public:
    ~TransformationManager();
    static TransformationManager * Instance(); // returns single
        instance of object

    Cltn<Hyperedge *> *CreateNewPath(); // creates a new simple
        path and returns elements
    Hypergraph *CurrentGraph() { return( current_graph ); } //
        access methods for current hypergraph
    void InstallCurrentGraph( Hypergraph *new_graph ) {
        current_graph = new_graph; }
    void TransApplicableTo( Hyperedge *edge ); // determines appli-
        cable transformations for hyperedge
    void TransApplicableToPair( Hyperedge *edge1, Hyperedge *edge2
        ); // determines applicable transformations for pair of
        hyperedges

private:
    TransformationManager(); // protected constructor enforces sin-
        gle instance pattern
    void CreateDictionaries(); // creates lists of applicable
        transformations for all hyperedges at initialization

    Hypergraph *current_graph; // pointer to the current hypergraph
        being displayed
    Dcn<transformation> transformation_dictionary[12]; // array of
        lists of single selection transformations for each hyperedge
        type
    Dcn<transformation> transformation_dictionary2[12]; // array of
        lists of double selection transformations for each hyperedge
        type

};
```

Class FileManager

Application of Use Case Maps to System Design with Tool Support

The **FileManager** is a utility object that provides file saving and loading capabilities. As file loading is a complex operation which requires thousands of lines of code and many temporary data structures to store object relationships an object is necessary to encapsulate the data structures and provide the functionality of separate steps in the process as member functions.

The **ComponentManager** object is a utility object which contains much of the functionality required to manipulate pools of components such as hit detection, resizing of components, cutting, copying, and pasting. Most of this functionality existed in the prototype. It uses the **Handle** class to draw the handles for resizing components.

The **LabelManager** is a utility object which provides functionality related to scenario labelling of paths such as determining if a user inputted label already exists in the map and relabelling paths when forks/joins are added or removed.

The **LinearGenerator** provides functionality for generating the linear form of use case maps from the maps stored in the editor's data structures. Its operation is described in detail in Chapter 5.

Performance Prediction classes

**DataStoreDirectoryDeviceDirectoryResponseTimeResponse-
TimeManager**

These classes are part of the extensions to the tool to support performance prediction. They are described in greater detail in Chapter 6. All of the main purposes of all of these classes is to capture user given information about program execution specifics and transfer that information to the simulation environment.

Class DataStoreDirectory

The **DataStoreDirectory** class manages the user given layered data stores that may be referenced by responsibilities. It allows specification of the data stores by the user as well as the referencing of these data stores by responsibilities.

```
typedef enum { DATA_STORES, ACCESS_MODES } editing_mode;  
  
class DataStoreItem { // data class for each store or access  
    mode, used by DataStoreDirectory
```

Application of Use Case Maps to System Design with Tool Support

public:

```
DataStoreItem( const char *dsi ); // constructors, accept user
    inputted string as parameters
DataStoreItem( const char *dsi, int id );
~DataStoreItem() {}

int Identifier() { return( identifier ); } // returns integer
    identifier
char *Item() { return( item ); } // access modes for string
    representing item
void Item( const char *new_item ) { strncpy( item, new_item, 19
    ); }

static void ResetCount() { item_count = 0; max_count = 0; } //
    resets counters for elements
```

private:

```
char item[20]; // string entered by user representing data
    store or access mode
int identifier; // internal integer identifier used to match
    elements with references

static int item_count, max_count; // counter for items, current
    maximum reference
};
```

class DataStoreDirectory {

public:

```
~DataStoreDirectory() {}
static DataStoreDirectory * Instance(); // returns pointer to
    sole instance, creates instance on first call

void List(); // list stores or modes in list box
void DisplayDataStores( Responsibility *responsibility ); //
    displays available stores in responsibility data usage list
    box
void DisplayAccessModes(); // displays available access modes in
    responsibility data usage list box
void ViewDataStores(); // invokes data store dialog box
void SetMode( editing_mode mode ) { emode = mode; } // set
    method for editing mode
```

Application of Use Case Maps to System Design with Tool Support

```
char * Item( int index ) { return(
    data_stores[emode].Get(index) ->Item() ); } // access methods
    for elements
char * Item( editing_mode mode, int identifier );
DataStoreItem *DSItem( int index ) { return(
    data_stores[emode].Get(index) ); }
DataStoreItem *DSItem( editing_mode mode, int index ) { return(
    data_stores[mode].Get(index) ); }
void AddItem( const char *item ) { data_stores[emode].Add( new
    DataStoreItem( item ) ); }
void AddItem( const char *item, int identifier ) {
    data_stores[emode].Add( new DataStoreItem( item, identifier
    ) ); }
void EditItem( const char *item, int index ) { this->DSItem(
    index )->Item( item ); }
void DeleteItem( int index ); // deletes the item at index
void ClearDirectory(); // removes all elements from directory
boolean IsReferenceValid( DataReference *dr ); // determines if
    an element still exists
void Save( FILE *fp ); // saves the directory contents
```

protected:

```
Cltn<DataStoreItem *> data_stores[2]; // lists of data stores
    and access modes
```

private:

```
DataStoreDirectory() { emode = DATA_STORES; } // constructor,
    sets initial editing mode
editing_mode emode; // the elements being currently edited (
    data stores / access modes )
```

```
};
```

```
static DataStoreDirectory *SoleDataStoreDirectory; // class vari-
    able for sole instance of directory object
```

Class DeviceDirectory

The **DeviceDirectory** class manages user input that describes the hardware devices that may exist in a design.

```
typedef enum { PROCESSOR, DISK, DSP, OTHER } device_type;
```

Application of Use Case Maps to System Design with Tool Support

```
class Device { // data class used by DeviceDirectory

public:

    Device( device_type type, const char *name, const char *ct ); /
        / constructor, accept user inputted strings as parameters
    ~Device();
    char * Name() { return( device_name ); } // access methods for
        elements
    void Name( const char *new_name ) { strncpy( device_name,
        new_name, 19 ); }
    char * Characteristics() { return( characteristics ); }
    void Characteristics( const char *new_characteristics );

private:

    device_type dtype; // enumerated variable for type of device
    char device_name[20], *characteristics; // user entered strings
        for device name and characteristics

};

class DeviceDirectory {

public:

    ~DeviceDirectory() {}
    static DeviceDirectory * Instance(); // returns pointer to sole
        instance, creates instance on first call

    void ViewDevices(); // invokes device dialog
    void ListDevices( device_type type ); // displays the devices
        of a given type in dialog list box
    Device * SelectedDevice( device_type type, int index ); //
        returns the device object corresponding to index
    void AddDevice( device_type type, const char *name, const char
        *ct ) // methods to add, delete device objects from directory
        { devices.Add( new Device( type, name, ct ) ); }
    void DeleteDevice( Device *d ) { devices.Delete( d ); }
    void ClearDirectory(); // removes all elements from directory
    void Save( FILE *fp ); // saves the directory contents

    Cltn<Device *> * DeviceList() { return( &devices ); } //
        returns the list of devices
    Device * DeviceWithName( const char *device_name ); // returns
        the device object with the given name
```

private:

```
DeviceDirectory() {} // constructor, private to enforce single-  
ton pattern
```

```
Cltn<Device *> devices; // list of devices
```

```
};
```

```
static DeviceDirectory * SoleDeviceDirectory; // class variable  
for sole instance of directory object
```

Class ResponseTime

The **ResponseTime** class encapsulates the response time relationship between two timestamps.

```
class ResponseTime { // class which encapsulates response time  
relationships between timestamp points
```

public:

```
ResponseTime( Timestamp *t1, Timestamp *t2, const char  
*new_name, // constructor, accepts timestamp points and user  
inputted characteristics as parameters  
int time = DEFAULT_TIME, int pct = DEFAULT_PERCENTAGE );  
void Time( int time ) { response_time = time; } // access meth-  
ods for relationship elements  
int Time() { return( response_time ); }  
void Percentage( int pct ) { percentage = pct; }  
int Percentage() { return( percentage ); }  
char * Name() { return( name ); }  
void Name( const char *new_name ) { strncpy( name, new_name, 19  
); }  
Timestamp * Timestamps( int index ); // access method for  
timestamp points  
void Highlight(); // highlights the pair of timestamps on  
screen  
void Unhighlight(); // unhighlights the pair
```

private:

```
Timestamp *ts1, *ts2; // the two timestamp points involved in  
the relationship
```

Application of Use Case Maps to System Design with Tool Support

```
int response_time, percentage; // the desired response time and
    percentage
char name[20]; // user given name for relationship

};
```

Class ResponseTimeManager

The **ResponseTimeManager** is a singleton class that manages all of the relationships for a given design.

```
class ResponseTimeManager {

public:

    ~ResponseTimeManager() {}
    static ResponseTimeManager * Instance(); // returns pointer to
        sole instance, creates instance on first call

    void CreateResponseTime( Timestamp *t1, Timestamp *t2 ); //
        creates new response time object, invokes dialog
    void EditResponseTime( ResponseTime *rt ); // invokes dialog
        for existing requirement
    ResponseTime * ResponseTimeExists( Timestamp *t1, Timestamp *t2
        ); // determines if a requirement exists between timestamp
        points t1, t2
    boolean ResponseTimesExist( Timestamp *ts ); // determines if
        timestamp is involved in any requirement
    boolean NewResponseTimeValid( Timestamp *t1, Timestamp *t2 ); /
        / determines if a potential requirement is valid, if they can
        be reached from each other
    ResponseTime * GetResponseTime( int index ) // returns the
        response time object with given index
    { return( response_times.Get(index) ); }
    ResponseTime * GetResponseTime( int selection, int index )
    { return( timestamp_responses[selection][index-1] ); }
    void DeleteResponseTime( ResponseTime *rt );
    void AddResponseTime( ResponseTime *rt ) { response_times.Add(
        rt ); } // add methods for relationships
    void AddNewResponseTime( Timestamp *tf, Timestamp *ts, const
        char *nm, int time, int pct )
    { response_times.Add( new ResponseTime( tf, ts, nm, time, pct )
        ); }
    void DeleteTimestamp( Timestamp *ts ); // removes timestamp
```

Application of Use Case Maps to System Design with Tool Support

```
    from all relationships
void ViewResponseTimes( Timestamp *ts ); // invokes dialog for
    response time requirements for given timestamp
void ListTimestampRT( Timestamp *ts ); // lists all current
    relationships for timestamp in list box
void ViewAllResponseTimes(); // invokes dialog for all response
    time requirements
void ListResponseTimes(); // lists all current relationships in
    list box
void Save( FILE *fp ); // saves directory to file

protected:

    Cltn<ResponseTime *> response_times; // global list of response
        time relationships for the current design

private:

    ResponseTimeManager() {}
    boolean OrderCorrect( Hyperedge *first, Hyperedge *second );

    static ResponseTime *timestamp_responses[2][10];

};

static ResponseTimeManager * SoleTRM; // class variable for sole
    instance of directory object
```

Presentation classes

XFPresentation*, PSPresentation*

The XFPresentation class implements the interface to the XWindows functions by which all map elements draw themselves on the screen. Similarly the PSPresentation implements the same interface (by way of the common Presentation abstract class) to allow map elements to draw themselves in a PostScript file. These classes existed in the prototype.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved