uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Jason Kealey
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M. (Computer Science)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Enhanced Use Case Map Analysis and Transformation Tooling

TITRE DE LA THÈSE / TITLE OF THESIS

Daniel Amyot
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Stéphane Somé

Jean-Pierre Corriveau

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Enhanced Use Case Map Analysis and Transformation Tooling

Jason Kealey

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Ottawa, Ontario, Canada

September 2007

Your file Votre référence
ISBN: 978-0-494-49221-5
Our file Notre référence
ISBN: 978-0-494-49221-5

Abstract

The Use Case Map (UCM) notation enables the use of graphical scenarios to model grey-box views of a system's operational requirements and behaviour. The scenario traversal mechanism is the most popular UCM analysis technique as it allows modellers to test, explain, and transform UCM models. However, its implementation in the existing UCM-Nav tool is limited and hard to use, and its high feature coupling makes it difficult to maintain and evolve.

This thesis introduces major analysis and transformation enhancements to the recent jUCMNav Eclipse plug-in by providing an extensible scenario traversal semantics accompanied by improved model transformations to Message Sequence Charts. Furthermore, this thesis identifies a set of semantic variation points for which the behaviour is unclear in UCMs, laying the groundwork for notational clarifications and user-defined semantic profiles. Finally, the transformation from textual use cases to UCMs is presented as a demonstration of the extensibility of jUCMNav's transformation framework.

Acknowledgment

I would like to express my deepest gratitude to my supervisor, Dr Daniel Amyot, for his exceptional implication and guidance. I am extremely thankful for the liberty that I was given during the realization of this thesis and the outstanding support he has given me. Tu es un superviseur exceptionnel Daniel, je n'aurais pu demander mieux!

Many thanks go to Etienne Tremblay, Jean-Philippe Daigle, Jordan McManus, Jacques Sincennes, and especially Jean-François Roy for their contributions to jUCM-Nav. Without such an exceptional group of dedicated people, the tool would never be where it is now. In addition, Gunter Mussbacher's vision and deep understanding of the User Requirements Notation served as a guiding light during the realization of this project. Thanks for all the help!

Furthermore, sincere thanks to all the other contributors/users of jUCMNav with honorable mention to the requirements engineering course students who gave the scenario traversal algorithm a beating, discovered numerous issues, and greatly contributed to its robustness.

I would like to thank the Natural Science and Engineering Research Council of Canada (Canada Graduate Scholarship) for their financial support.

Finally, and most importantly, I would like to thank my fiancée Catherine Mougeot for her constant love and support.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
BNF	Backus-Naur Form
CASE	Computer-Aide Software Engineering
CSM	Core Scenario Model
DTD	Document Type Definition
EMF	Eclipse Modeling Framework
FSM	Finite State Machine
GEF	Graphical Editing Framework
GRL	Goal-oriented Requirement Language
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
jUCMNav	Java Use Case Maps Navigator
NFR	Non-Functional Requirement
NLP	Natural Language Processing
MSC	Message Sequence Chart
SDL	Specification Description Language
SVDPI	Subject-Verb-Direct Object-Preposition-Indirect Object
TTCN-3	Testing and Test Control Notation, Version 3
UI	User Interface
UCM	Use Case Map
UCMNav	Use Case Map Navigator
URN	User Requirements Notation
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformation

Chapter 1. Introduction

This thesis presents topics related to the requirements, design and realization of an improved scenario traversal mechanism inside jUCMNav [23], an Eclipse-based tool for integrated goal and scenario modelling based on the User Requirements Notation (URN) [2]. URN is an emerging standard that combines two notations for capturing early requirements: the Use Case Maps (UCM) notation [22] and the Goal-oriented Requirement Language (GRL) [21]. A full iteration of the software development lifecycle (requirements, design, implementation, testing) was performed to evolve jUCMNav from v2.0 to v3.0 during the course of this thesis. Even though this thesis focuses on UCMs, its relationship to GRL inside URN cannot be discarded.

Since the scenario traversal mechanism defines in a way the dynamic semantics of UCMs, this thesis also proposes semantic clarifications dearly needed in UCMs and a traversal validation technique using a fully re-written Message Sequence Chart (MSC) [19] export mechanism, with improvements in terms of maintainability. In addition, further improvements were made to the model transformation framework via the import from textual use case prototype. It should be noted that the work presented here is a focused subset of the work performed; more information can be obtained on the jUCMNav wiki site [23][26].

1.1. Motivation

Over the past decade, it has been shown that Use Case Maps (UCM) represent a useful behaviour modelling notation. UCMs can be used in a large number of contexts, not limited to software processes, because of their high level of abstraction. Furthermore, their visual simplicity allows non-technical stakeholders to take part in discussions concerning these processes. Tooling exists to create UCM models but the only tool available [31][44] that supports the analysis, execution and transformations of these models, called UCM-Nav, is hard to install, hard to use, hard to maintain and, worst of all, riddled with bugs.

Another tool, jUCMNav [23], exists which allows the creation and manipulation of UCM models but does not support analysis, execution, and transformations of UCM models. A UCM execution tool greatly enriches the power of the Use Case Maps notation. Indeed, thanks to the enriched visualization brought on by automated path highlighting, designers can easily share with all stakeholders the various scenarios contained within a set of UCMs. Manually traversing a complex set of UCMs can quickly become error-prone when a large number of nested diagrams are traversed or when complex conditions must be evaluated. Because the modeller can check behaviour and validate it with the other stakeholders, the execution engine helps improving the quality of the model.

Furthermore, an execution engine allows for regression testing during the model's evolution. The impact of refinements can be verified to be consistent with the original model thanks to various post-conditions. Moreover, when scenarios represent features, they must often be combined in a working system. The execution engine facilitates the discovery of undesirable feature interactions when separate scenarios are combined.

Finally, transformations to and from Use Case Maps are necessary for the notation to be a part of a designer's toolkit. Since UCMs can help fill the gap left by other popular early requirements notations (as opposed to imposing a methodology), tight integration with other notations improves the software engineer's efficiency. Basic UCM models can be created from more abstract representations such as textual use cases. These models are then refined to the point where the designer wishes to move on to detailed design, perhaps with Message Sequence Charts. By generating a basic MSC structure, some of the grunt work is eliminated allowing the designer to focus on the new aspects instead of repeating the same information in different models.

For these reasons, a new UCM execution engine integrated with the leading UCM editor (jUCMNav) model is very much desirable and represents an important research contribution. As jUCMNav is an editor for the entire User Requirements Notation (URN), it also paves the way to a tighter integration of the Use Case Map (UCM) notation with the Goal-oriented Requirement Language (GRL).

1.2. Thesis Goals

The goal of the work presented in this thesis is to deprecate the existing UCM analysis and transformation toolset (UCMNav [44] and UCMExporter [5]) by providing a new and improved implementation inside the new UCM modeling tool, jUCMNav [24]. Both functional and quality problems that plague the existing toolset are to be addressed. In a nutshell, the high-level goal of this thesis is to strengthen the tool support for the analysis and transformation of UCM models. A breakdown of this goal can be seen in the GRL model of Figure 1.

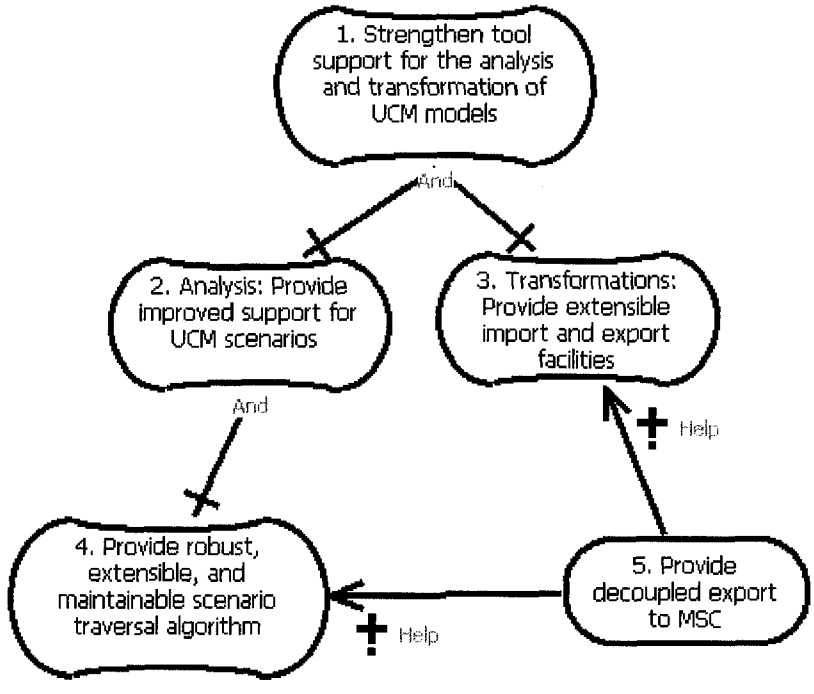


Figure 1 Thesis Goals

The two main sub-goals are improving support for UCM scenarios and providing extensible import/export facilities. More concretely, the goals of this thesis are to provide appropriate tool support for the scenario traversal algorithm and the MSC export. Although these elements are already available in UCMNav, they are plagued with multiple quality issues. Reworking UCMNav to improve the non-functional aspects of these features would be a bad investment because the first phase of its deprecation has already

been completed. jUCMNav v1.0's top level goal was to provide an easy-to-use, extensible, and maintainable editor for Use Case Map modelling. This thesis' goals are the continuation of jUCMNav's vision into the areas of analysis and model transformation.

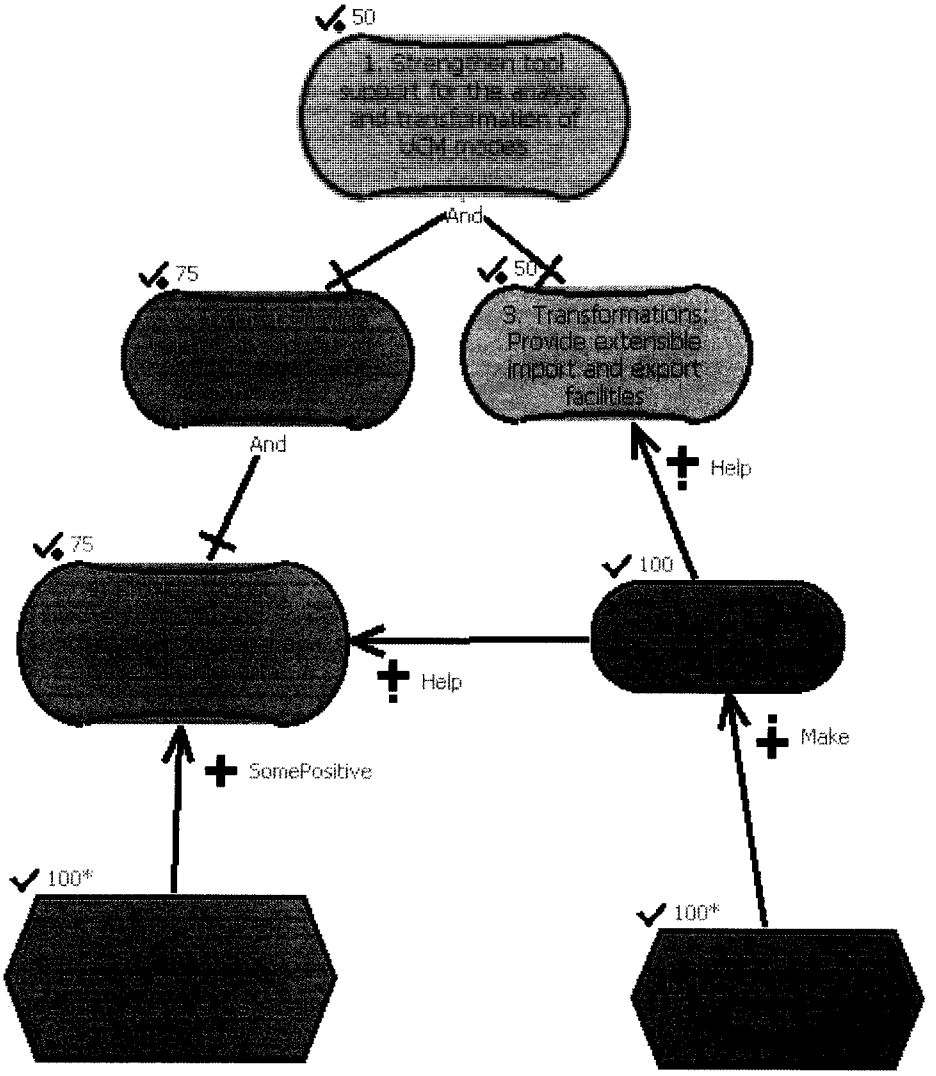


Figure 2 Possible Solution

The simplest solution, in terms of work required, to the goals presented in Figure 1 is presented in Figure 2. This solution presents the possibility to rely exclusively on external tools to fulfill the goals. Task A represents the conversion of a UCM model to another notation for which tools are readily available to perform the scenario traversal. Although this could include re-using UCMNav's scenario traversal algorithm, a more logical implementation would attempt to convert to finite state machines (FSM), use an FSM execution engine, and interpret the execution trace as a scenario execution in the original

model. Task B simply proposes to continue using UCMExporter to perform the MSC generation. Although using a well-known (and tested) external execution engine would improve the robustness of the traversal engine, this solution limits the extensibility of the tool. Furthermore, the conversion process from UCMs to any other notation would certainly have limitations and would become a problem in its own right. In addition, jUCM-Nav's maintainability and extensibility goals would not be respected, even though the functional goals could be achieved.

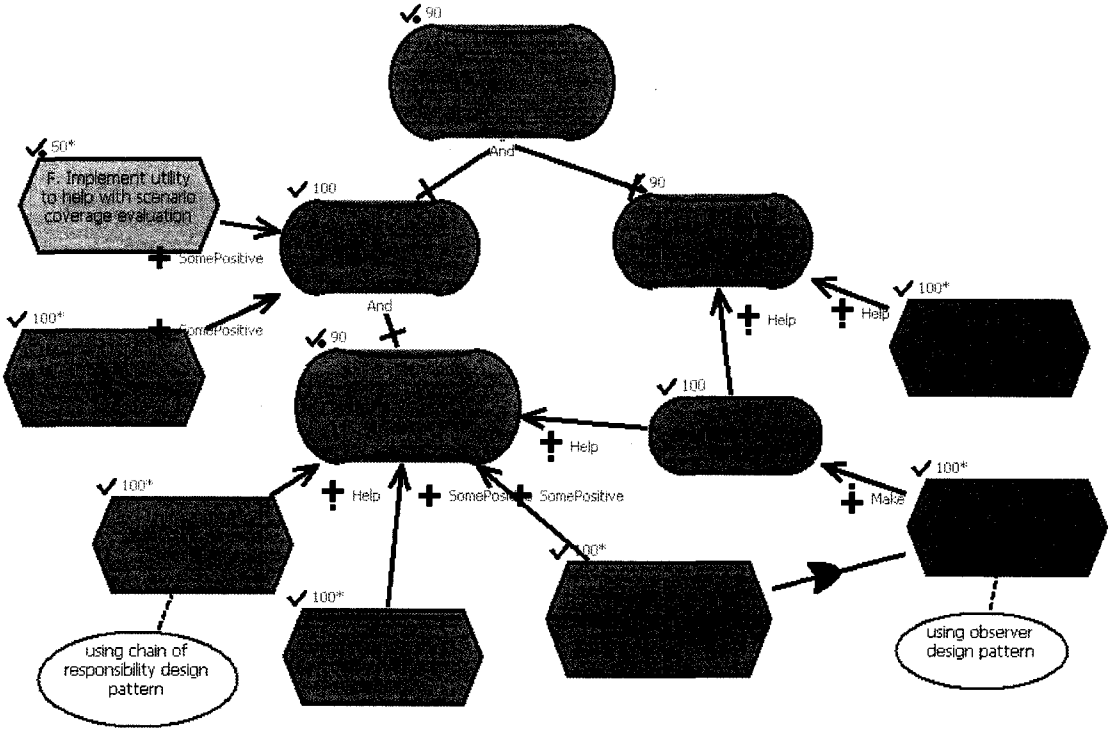


Figure 3 Chosen Solution

Figure 3 presents the alternative which drove the work performed during the course of this thesis. The scenario traversal (Task A) and MSC export (Task B) algorithms were redesigned and re-implemented from scratch and directly integrated into jUCMNav. Tasks C through G are secondary tasks which help increase the robustness, extensibility, maintainability, and usability of the main accomplishments.

1.3. Thesis Contributions

The following contributions, related to our thesis goals, are described in this thesis:

Vastly improved, completely re-engineered UCM scenario traversal (Goals 2 and 4)

With respect to the scenario traversal mechanism described in [3][32] and supported in the UCMNav tool, we provide the following enhancements:

- Task A: Richer data model (additional support for integers and enumerations).
- Task A: Usable and flexible condition and pseudo-code syntax; Java-like and SDL-like.
- Task A: New scenario definition elements: included scenarios, post-conditions, reached end points.
- Task A: Much easier-to-use and reliable user interface.
- Task A: Completely re-engineered architecture with focus on extensibility and maintainability.
- Task F: Possibility to run all scenarios and evaluate path coverage.
- Task D: Validated scenario traversal algorithm using MSC Export.
- Task A: Prototyped integration with GRL Strategies.

Extensible import and export plug-in framework (Goals 3 and 5)

With respect to the MSC export facilities described in [3][32] and supported in the UCMNav / UCMExporter tools, we provide the following enhancements:

- Task B: Completely re-engineered the export to Message Sequence Charts (MSC).
- Task C: New UCM model debugging capabilities via scenario export scenario to flat UCM models.
- Task G: Prototype of import from textual use cases.
- Goal 3: Export to Telelogic DOORS, a requirement management tool. (Not discussed in thesis, see publications).

Identification of UCM traversal semantic variation points (Task E)

- Task E: Identification of relationships to UCM workflow pattern limitations.
- Task E: Identification of possible future enhancements to the notation.

Goal 1: Illustration and validation via a case study

Publications

Many of the above contributions have already led to publications:

- Goal 1: Kealey, J., and Amyot, D., *Enhanced Use Case Map Traversal Semantics.* In: 13th System Design Language Forum (SDL'07), Paris, France, September 2007. LNCS 4745, Springer, 133-149.
- Goal 3: Roy, J.-F., Kealey, J., and Amyot, D., *Towards Integrated Tool Support for the User Requirements Notation.* In: SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modelling, Kaiserslautern, Germany, May. LNCS 4320, Springer, 198-215.
- Goal 3: Kealey, J., and Amyot, D., *Towards the Automated Conversion of Natural-Language Use Cases to Graphical Use Case Maps.* In: 2006 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE06), Ottawa, Canada, May 2006. IEEE, 2377-2380.
- Goal 3: Kealey, J., Kim., Y., Amyot, D., and Mussbacher, G.: *Integrating An Eclipse-Based Scenario Modeling Environment With A Requirements Management System.* In: 2006 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE06), Ottawa, Canada, May 2006. IEEE, 2432-2435.

1.4. Thesis Outline

This thesis is structured as follows. Chapter 2 presents high-level background information relevant throughout the thesis. Each chapter thereafter will introduce additional background information as required. Chapter 3 presents this thesis' main accomplishment, the complete re-engineering of the Use Case Map scenario traversal algorithm. Chapter 4 introduces another important achievement: the generation of Message Sequence Charts from UCMs. A case study is expanded in Chapter 5, unifying most of the new elements introduced in the previous chapters. In its subsections, Chapter 6 discusses various subjects related to the main goal of strengthening the tool support for the analysis and transformation of Use Case Map models. Finally, Chapter 7 concludes the thesis by revisiting the main contributions and by presenting future work.

Chapter 2. Background

2.1. Requirements

2.1.1 Requirements Engineering

Requirements engineering is a practice that helps transform abstract and vague ideas into concrete specifications addressing the needs and goals of the stakeholders while respecting budget, time, and other types of constraints. Requirements engineering provides systematic means of eliciting, writing, validating, and managing simple-to-understand, uniform, traceable, testable, and prioritized requirements which are required to design, build, and test a system. With the help of well-defined requirements, an organization can properly manage its projects because progress and quality become measurable metrics.

Regardless of how requirements are expressed, they are the unifying thread of the software development process even if they are usually created during the project's inception (or at the start of each iteration in an iterative development process). They drive the design of the system, where the architecture of the system is defined, and serve as reference material during the implementation. When software has been produced, it can be validated against the initial requirements. Because of their importance, large systems often have multiple sets of requirements at varying abstraction levels, e.g. from the simple business goals expressed in the vision and scope document, to the more concrete user requirements expressing high-level scenarios, to the explicit software requirements specifications containing fine-grained system details.

This thesis deals mainly with early functional requirements engineering: the high-level behaviour of the system. The description and analysis of non-functional requirements, which express quality concerns (such as speed, reliability, cost, and usability for example), are also important in early requirements engineering but are somewhat outside the scope of our work.

2.1.2 Requirement Notations

Traditionally, people have written free-form textual discussions to explain the behaviour of a system. This is good for initial discussions but does not scale because of the nature of unstructured text: ideas are not located easily and details are often sparse and scattered. It is better practice to list different functionality in a clear, structured pattern such as the following extract taken from the jUCMNav's 1.0 requirements [24]:

jUCMNav SHALL support the export of a selection of maps to individual bitmap files.

How to write good textual requirements is a topic that has been discussed by many authors (e.g., [1][42]). However, while the mileage that a team can obtain with a collection of textual requirements varies from project to project, more often than not these are insufficient. Structured textual requirements are good to describe features but not to describe scenarios or relationships between features. To better represent scenarios, a common tool of the requirement engineer's arsenal is the textual use case [11]. Textual use cases describe scenarios in which actors and a system under description interact. Such sequence of events increases the comprehension of the context in which the various features come into play. Use cases illustrate the system's functionality and do not usually include technical jargon. By utilizing language that the domain expert and end users understand, a broader group of stakeholders can discuss the system's behaviour (unlike more formal approaches).

The UML standard [35] provides numerous graphical notations to express requirements such as the Use Case Diagram, the Activity Diagram, and the Sequence Diagram. These notations are very important requirements engineering notations but this thesis does not focus on a UML-based approach. Still, the text will mention these notations on occasion and the uninitiated reader should refer to the standard for more information. UML Activity Diagrams are of particular interest as they share many similarities with the Use Case Map notation presented in 2.2.2.

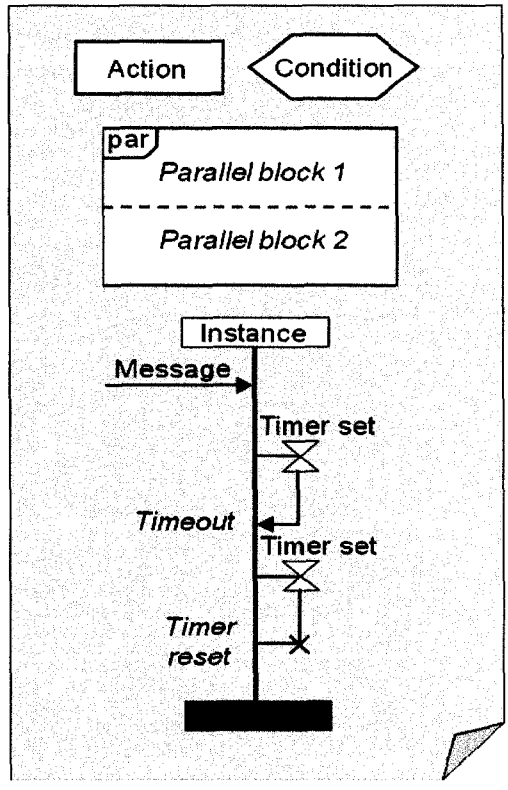


Figure 4 MSC Constructs

We shall end this discussion by mentioning a very popular visual notation for modelling sequence of events. Message Sequence Charts [19], whose main notation elements are introduced in Figure 4, have been used for decades to model protocols, interactions and behaviour at different levels of abstraction. The common usage of this notation is to visualize the interactions between various entities while focusing on a single scenario. Basic MSCs are especially close to the "execution" of the previously mentioned UML notations because they filter out extra information such as alternatives (although it is still possible to illustrate alternatives with MSCs). Because more often than not they are used at a low level of abstraction, these diagrams are more useful in the design, testing, and implementation phases as they illustrate the specifics of instance interactions. Conceptually and visually, the Sequence Diagrams found in UML 2.x are now very similar to MSCs. An export mechanism to generate MSC models from UCM models will be presented in Chapter 4. More information on scenario notations in general can be found in [4][29].

2.2. Introduction to the User Requirements Notation (URN)

2.2.1 The User Requirements Notation: Visual Requirements

The User Requirements Notation (URN) [20] is undergoing standardization by the International Telecommunication Union, Telecommunication Standardization Sector (ITU-T Z.150 series of recommendations). URN is a graphical notation that includes two complementary notations to help describe high-level, abstract requirements during the early analysis phase. The first notation is the Use Case Map (UCM) notation which represents the functional and operational aspects of a software system. The abstraction level of UCMs makes them an ideal notation to express user requirements as high-level scenarios (see 2.1.1). The second notation is the Goal-Oriented Requirement Language (GRL) which is discussed in section 2.2.4. GRL describes the non-functional aspects of a software system and is well adapted to model high-level business goals.

2.2.2 Use Case Maps

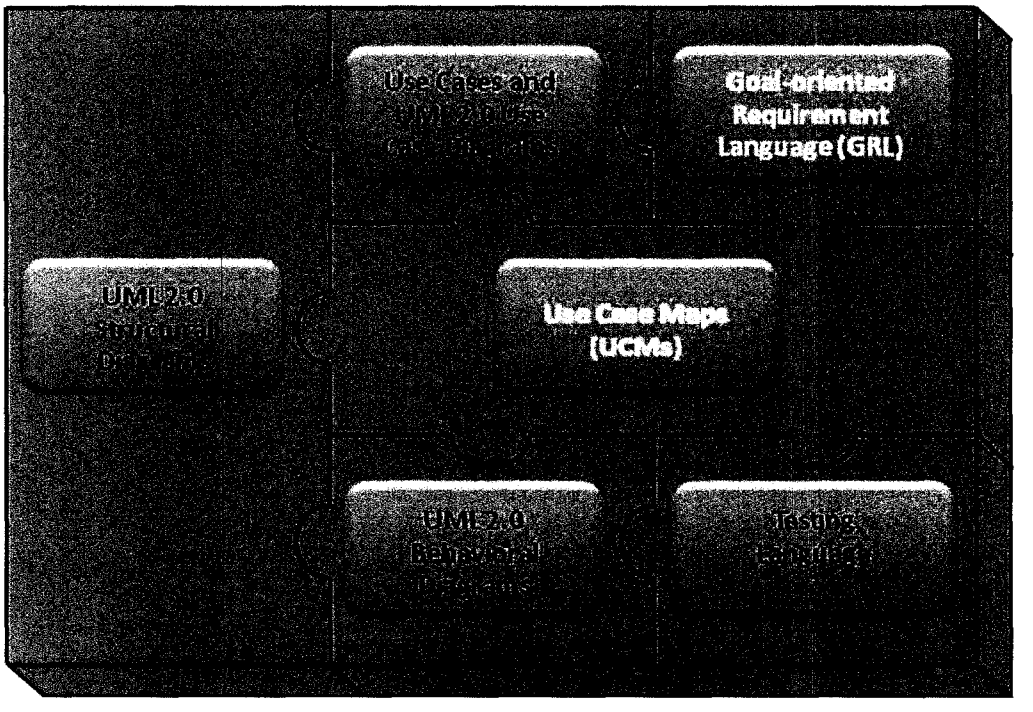


Figure 5 UCMs Bridge the Gap between Various UML Notations

This thesis is directly related to the tool support for the creation, analysis and transformation of Use Case Maps. Use Case Maps can model various types of reactive and distributed systems using graphical scenarios. Functional requirements can be illustrated as paths that cross architectural components, including actors. Thanks to its encapsulation capabilities and especially because of the abstraction level it retains (above that of interaction messages and component states), UCMs are very versatile. Because of these characteristics, UCMs find themselves as being an ideal complement to the various UML notations (see Figure 5). They can help model and validate requirements, capture business processes, describe patterns, do performance analysis, generate test cases, and visualize software in a reverse-engineering context, all in just about any application domain[1]. An example can be seen in Figure 7.

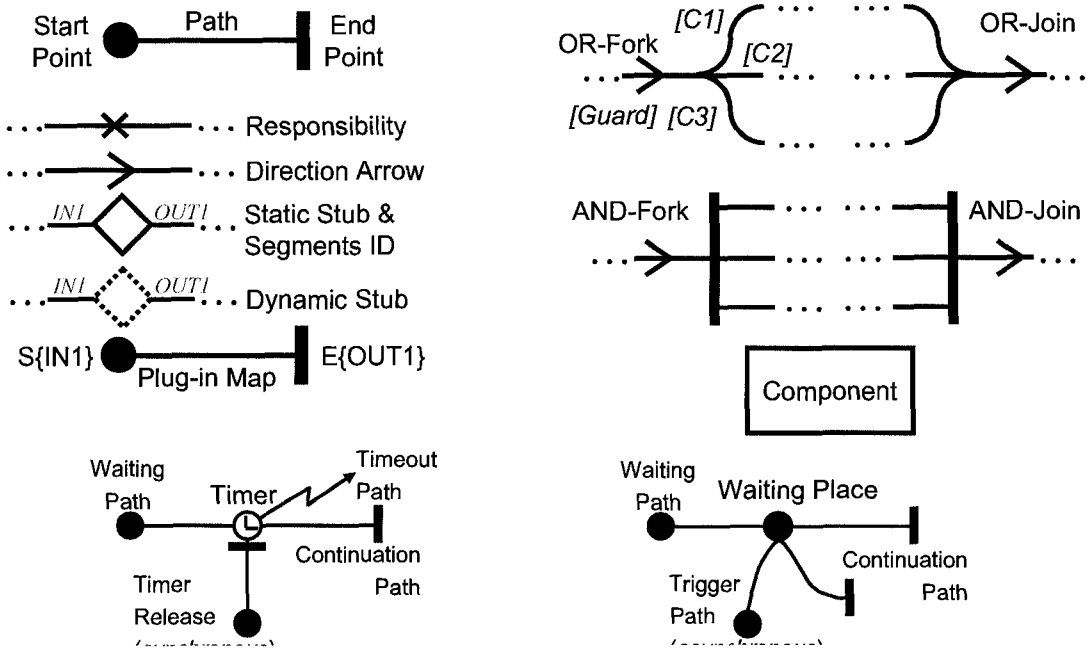


Figure 6 Use Case Map Notation Summary

The main constructs of the UCM notation are shown in Figure 6: a UCM model can contain multiple paths that begin at a one or many *start points* and end with one or more *end points*. Along the way, *responsibilities* illustrate tasks that are to be performed, optionally by the underlying components (boxes). Guarded alternatives are illustrated with *or-forks* that can later be merged back together with *or-joins*. Concurrency is intro-

[1] The UCM Virtual Library, http://www.UseCaseMaps.org/pub/, contains a collection of over 140 papers and theses illustrating these topics.

duced with the concept of *and-forks* and *and-joins*. The primary encapsulation and decomposition mechanism used in UCMs is the *stub*, which contains other maps that are called *plug-in maps*. A stub can either be *static* which means it can only contain a single plug-in, or *dynamic*, which allows it to be linked to multiple plug-ins with the appropriate plug-in chosen at runtime given a selection policy (see 2.2.3). Details about the semantics of *timers*, *waiting places* and different types of *connects* will be given in 3.4. UCMs separate component and responsibility references from their definitions allowing the same concept to be re-used multiple times in a model. More details on the notation can be found in [12][22].

The Use Case Map notation is very similar to the UML 2.0 Activity Diagram [7], but because of its additional constructs (e.g., dynamic stubs, timers, component structures) and analysis capabilities (scenarios definitions and the data model, presented in section 2.2.3), UCMs represent a richer visualization notation for many types of systems.

2.2.3 UCM Scenarios

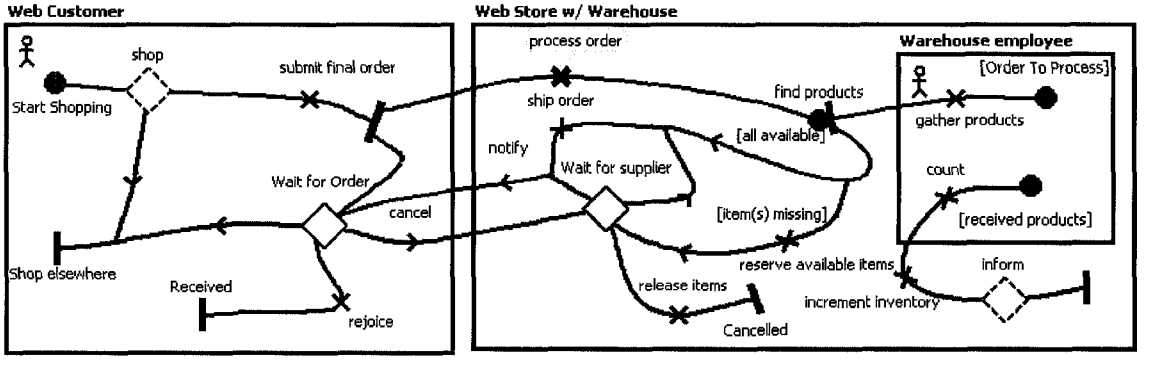


Figure 7 Sample UCM with an Active Scenario (highlighted)

UCM scenarios represent the "execution" of a particular behaviour through a pre-defined model. In their simplest form, scenarios are defined as a set of start points that are launched. The paths are traversed until they reach their end points and they visualize *which* alternative was taken at or-forks or dynamic stubs. We will go into greater detail in this thesis, but it is simple enough for now to present UCM scenarios as highlighted paths on a UCM model.

2.2.4 Goal-Oriented Requirement Language

The Goal-Oriented Requirement Language [21] is the URN component that focuses on non-functional requirements and quality aspects. It supports visual modelling of goals, alternatives, and rationales. Goal models depict the context in which decisions are taken: top level goals are broken down into sub-goals and tasks and relationships between the goals are visually illustrated (contribution, dependency, decomposition, etc.). Figure 1 in Section 1.2 describes this thesis' goals using a GRL model.

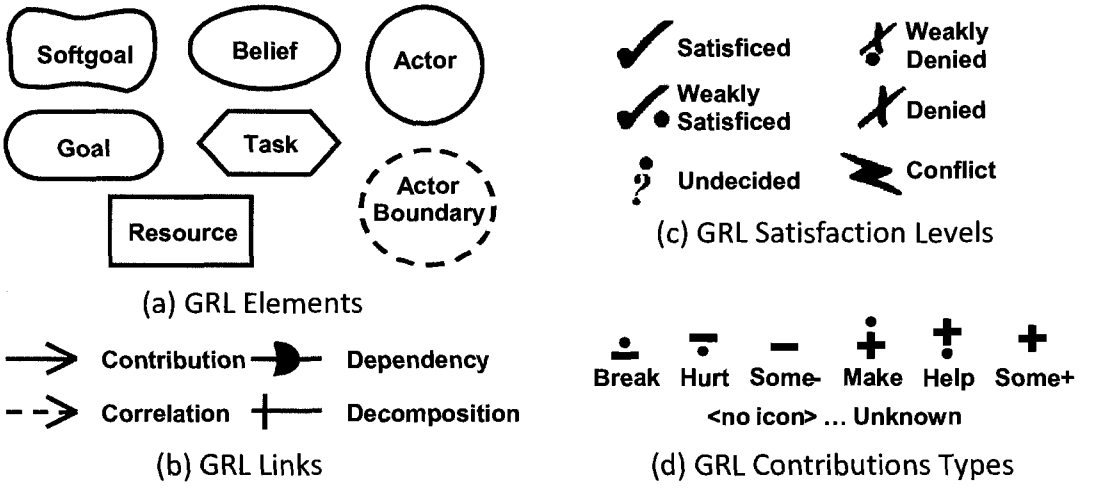


Figure 8 Goal-oriented Requirement Language Summary

Actors are stakeholders in the decision making process, they can contain a set of goals that impact their satisfaction levels. GRL was inspired from other goal modelling languages such as the NFR Framework and i*, but has its particularities, especially in the analysis domain [36]. For more detailed explanations on the different constructs, please refer to [21]. Non-functional aspects are not the focus of the work presented in this thesis, but future work is tightly coupled.

2.2.5 GRL Strategies

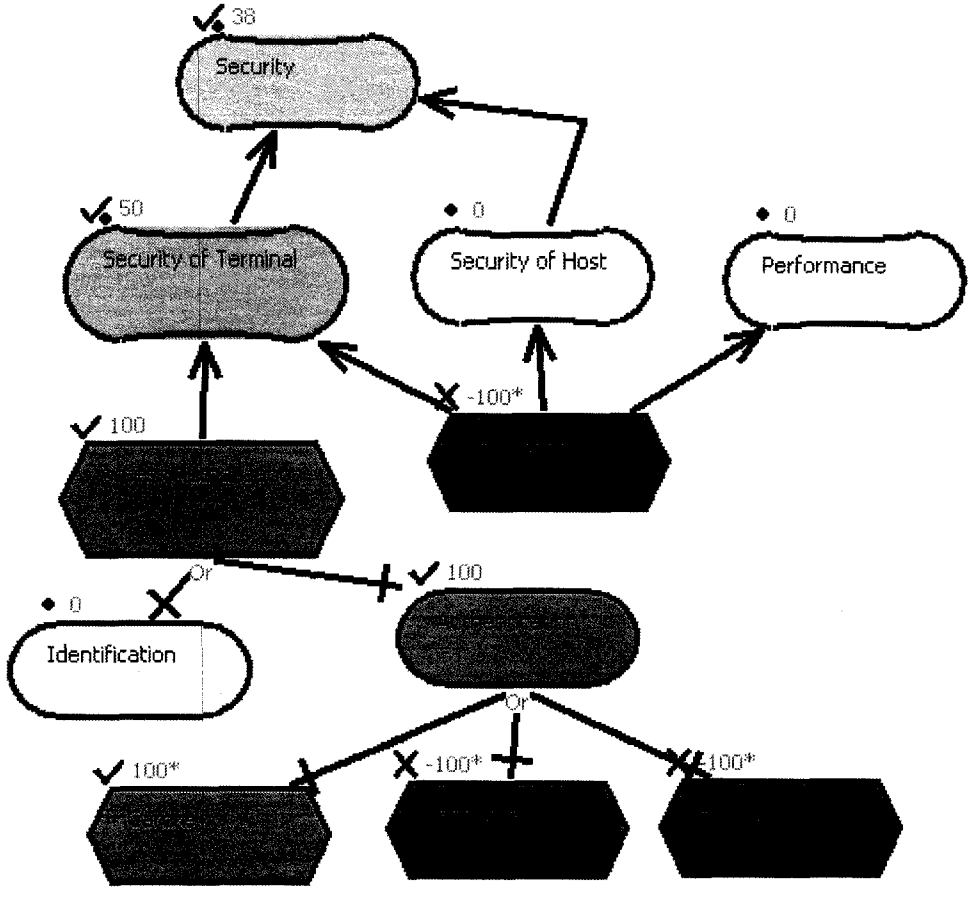


Figure 9 Sample GRL with an Active Strategy (Cardkey)

Similarly to UCM scenarios, GRL strategies illustrate the evaluation of alternative solutions using a different colour scheme. Certain elements in the model are given initial valuations and these valuations propagate up in a way determined by the relationships defined between the different elements. Therefore, the visualization of which goals are satisfied and which ones are not is instantaneous. Because goals can be allocated to actors, with varying factors of priority and criticality, the valuation of the goals impact the actor's overall satisfaction level. The propagation algorithm hence helps visualize the impact of a certain decision on stakeholder satisfaction.

2.3. Tool Support for URN Modelling

2.3.1 Historical Note: UCMNav

Around 1996, before the days of URN, UCMNav [44] was created to allow the manipulation of Use Case Map models. The technological choices made at the time now make it difficult to use and deploy and next to impossible to maintain as the notation and its applications evolve. UCMNav contains more than 100 classes representing 70 000 lines of undocumented C++ code, with only 10% of the code being automatically generated by a user interface editor. The initial architecture has vanished over the years with the changes made by about twenty people, mostly students. Strong coupling between the user interface and the model has caused its evolution to grind to a halt.

The user interface is based on the XForms [47] library and the tool requires the presence of an X Window server. While this choice made the tool available on many operating systems (GNU/Linux, Solaris™, HP/UX™, MS Windows™, and Mac OS X™), the X Window server dependency is a major obstacle to deployment on the most common platform, MS Windows™. The user interface itself is non-standard, counter-intuitive, incomplete, and imposes too many constraints. Furthermore, as the tool does not have a regression test suite, it is very brittle. Documentation on architecture and implementation decisions is also very limited. Finally, UCMNav has a dependency on external tools for MSC generation (e.g., UCMExporter [5]) and visualization (e.g., Telelogic Tau), hence hindering usability.

The disease that has affected UCMNav is not out of the ordinary in the industry, especially considering it was created as a simple proof-of-concept prototype. Albeit plagued with various issues, UCMNav has served as the supporting platform for close to a hundred publications related to UCMs [45].

2.3.2 jUCMNav

In January 2005, a team of software engineering students (lead by me) started working on their capstone project: jUCMNav [24][28], a Java-based UCM editor. The tool is a plug-in for Eclipse [13] and uses two main Eclipse projects: the Eclipse Modelling Framework

(EMF) [14] and the Graphical Editing Framework (GEF) [15]. Figure 10 gives an overview of jUCMNav's architecture.

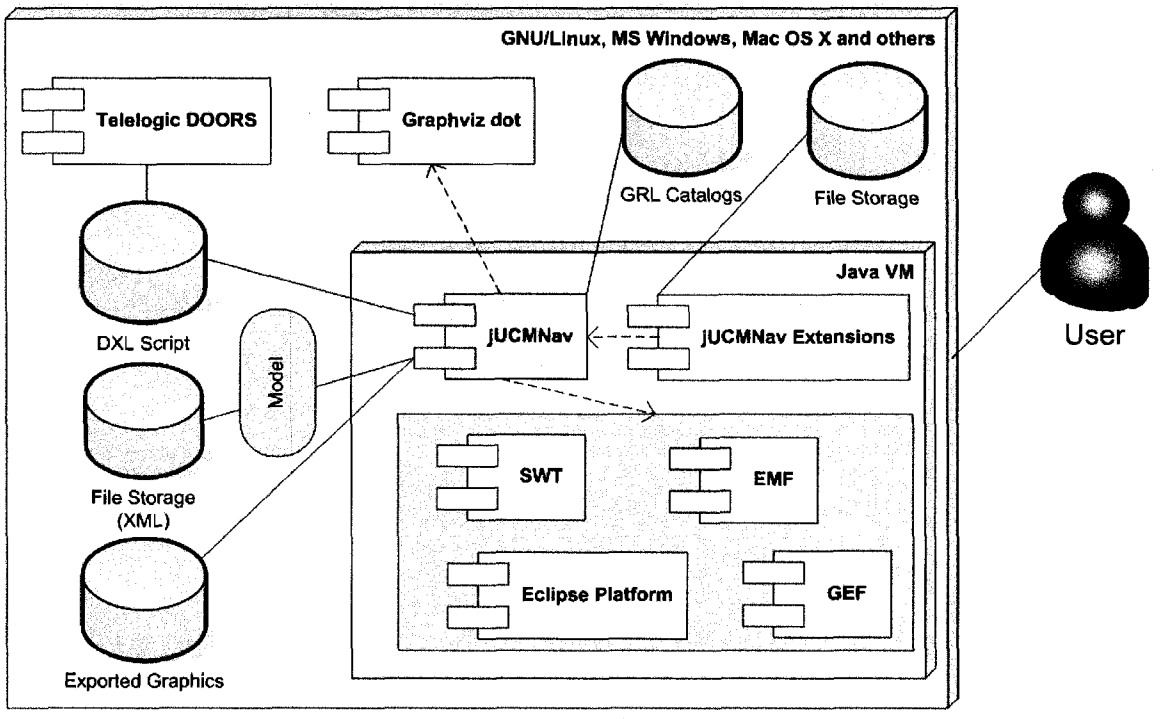


Figure 10 jUCMNav Architecture Diagram

The tool was built from the ground up using new frameworks to replicate a subset of the functionality found in UCMNav but with two additional goals: usability and maintainability. GEF allows the rapid development of a well-architected graphical editor. As for EMF, it generates code automatically from a UML class diagram describing the URN metamodel (see an extract in Figure 11). For jUCMNav, 50 000 lines of Java code were generated.

The combination of these technologies enabled the development of a well-documented, tested, architected, and usable UCM editor. UCM scenarios were not available in this iteration; this is the subject of Chapter 3 of this thesis. As for the GRL portion of the User Requirements Notation, the tool was updated in 2006 by Jean-François as part of his thesis [36]. Given the significant enhancements produced between jUCMNav v1.0 and v2.0 to support the creation and analysis of GRL models, and given the ease with which the concepts were generalized to support both URN sub-notations, it can be concluded that the initial maintainability goals were met if not exceeded.

All the features discussed in this thesis are enhancements to this tool.

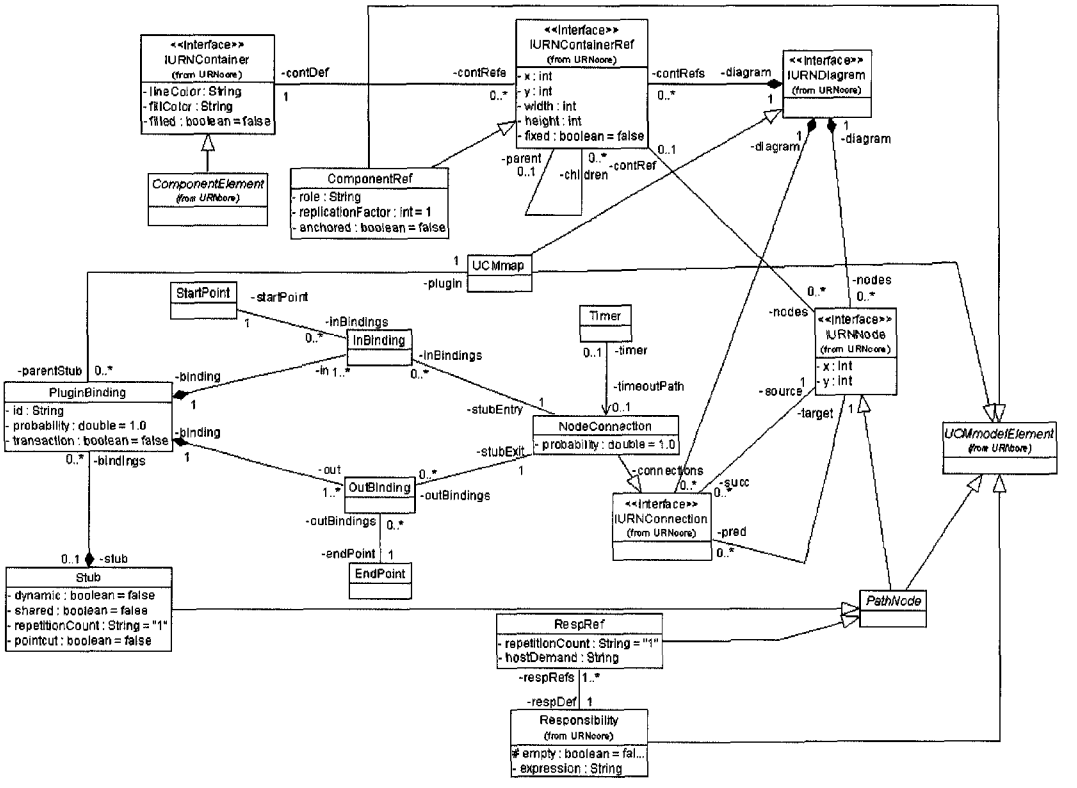


Figure 11 Partial URN Metamodel

2.4. Chapter Summary

This chapter presented background information used throughout the thesis. The field of requirements engineering is both wide and deep, and the topics presented here are only the tip of the iceberg. Each chapter shall present additional background information when necessary. This thesis focuses on the Use Case Map notation and extensions to jUCM-Nav, an Eclipse-based URN editor.

Chapter 3. Analysis – UCM Scenarios

This thesis' main accomplishment is the design and implementation of Use Case Map scenarios inside jUCMNav, identified as Task A in section 1.2. The goal of this chapter is to present the main challenges faced, the default traversal algorithm, and the architecture that enables maintainability and extensibility. Task F is discussed in 3.3.3.

3.1. Problem Statement

Scenarios are the single most important analysis tool available to Use Case Map designers. As summarized in 2.2.3, scenarios result from the traversal of a UCM graph starting at a list of start points. Scenarios are interesting for many reasons:

- Scenarios enable designers to visualize the execution of the UCM model, in context.
- Scenarios enable designers to verify and validate their UCM models.
- Stakeholders can easily read individual scenarios as a ground up approach to understanding a model.
- Scenarios are the starting point for any type of test-case generation from UCM models.

In the background, however, the implementation should achieve a few non-functional goals. First of all, the traversal engine should be easy to maintain as the notation evolves. Second, the implementation should be extensible. As will be presented in this chapter, there are many decisions concerning semantic variations that must be taken by the scenario traversal algorithm. Different contexts may require different traversal algorithms.

3.2. Background, Context, and Related Work

3.2.1 Background

As mentioned in section 2.3.2, jUCMNav was designed as a replacement for UCMNav (Figure 12). Apart from the creation of various reports and the generation of CSM (Core Scenario Models) from UCMs, the only remaining barrier to UCMNav's deprecation was support for UCM scenarios, which is now implemented thanks to the work presented in this chapter. (Reporting and CSM export were done in parallel to this work, by other contributors.) One of UCMNav's weak points was its maintainability, as identified in the original jUCMNav capstone project. The UCM scenario implementation in UCMNav was one of the main factors that drove this non-functional requirement into jUCMNav. UCMNav's implementation did not allow for evolution or growth; it severely limited the analysis capabilities mentioned in section 3.1.

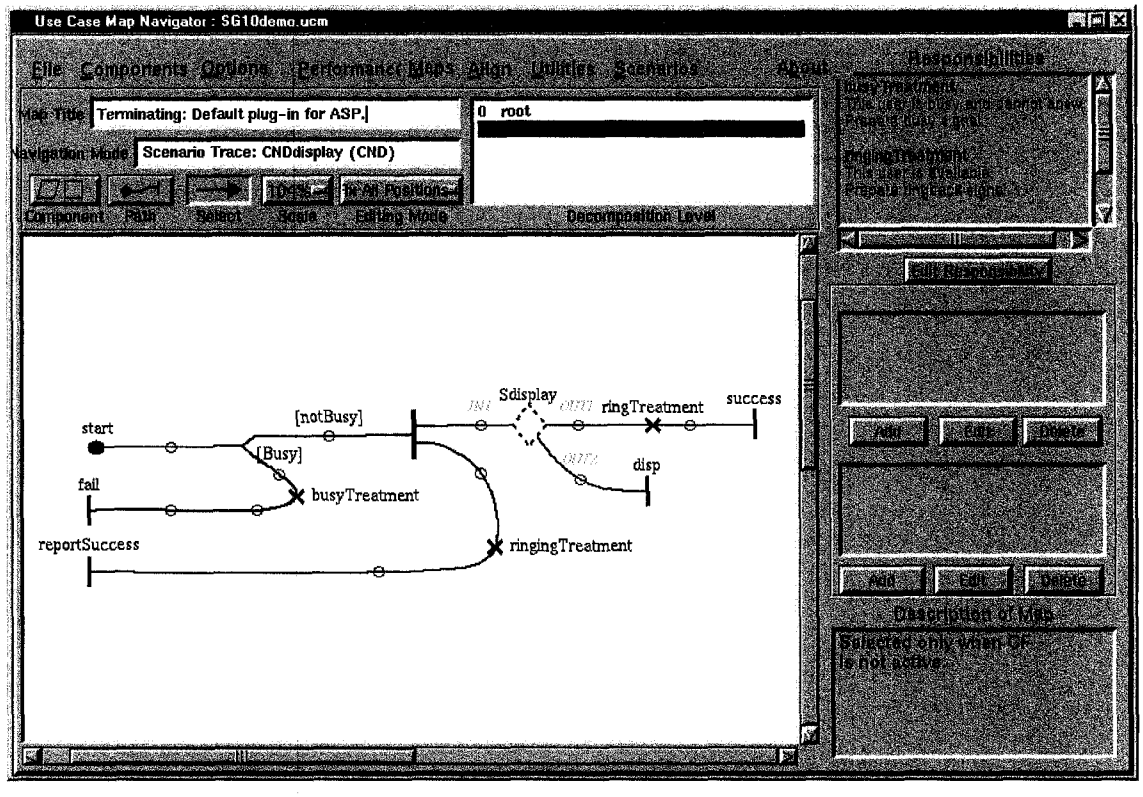


Figure 12 UCMNav with Active Scenario

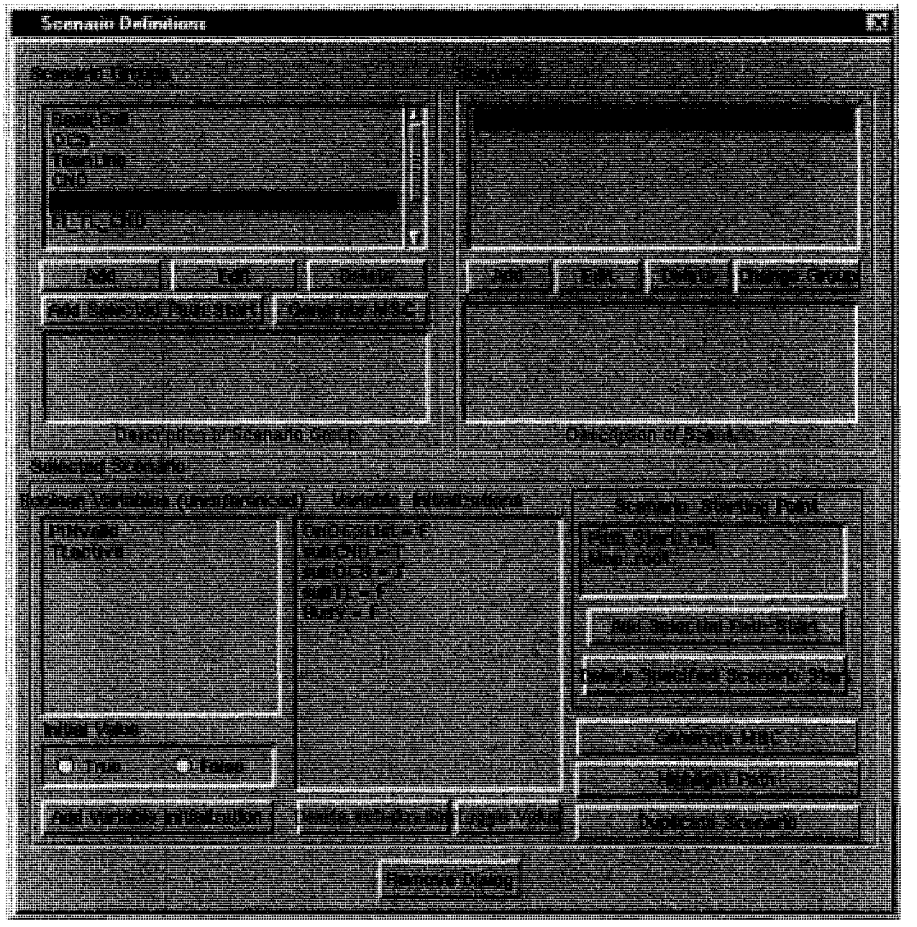


Figure 13 Original UCMNav Scenario Definition Dialog

The UCMNav implementation of the scenario traversal mechanism (Figure 13) suffers from many limitations. The only data type allowed is Boolean. Conditions and assignments are described using a very simple action language with a non-standard syntax. The same start points cannot be triggered multiple times. Scenario definitions are not reusable, leading to scalability and management issues. The traversal is combined with a linearization algorithm (for MSC generation) that is prone to errors; hence it is difficult to debug and maintain. The traversal is rigid, without semantic variation points and without tolerance for errors (it often blocks if something unexpected happens or is not initialized properly). Only one scenario can be run at a time. Still, the general theme and ideas behind the traversal engine are the same and these will be presented in detail in 3.3.

UCMNav implemented scenarios as a series of modal dialogs that were not very helpful. The implementation presented in this chapter is more in line with the user inter-

face for GRL strategies introduced in [36]: Scenarios are created, launched, and modified from the scenario and strategy view (see 3.3.1).

3.2.2 Context

Figure 14 presents the ucm.scenario package from the URN meta-model used in jUCM-Nav. The additions required to support scenarios (all located in the ucm.scenario package) made between version v2.0 and v3.0 are indicated. Reading this class diagram, one discovers that the root UCMspec can contain multiple groups which in turn can contain multiple scenarios. A model can also contain a series of variables which can be Booleans, integers or enumerations. Scenarios contain start points that define where to start the traversal and variable initializations that indicate the initial context. There are additional features that are closely related to scenario verification such as scenario pre/post conditions and scenario end points, which define the end points that could/should have been reached at the end of the traversal. Scenarios can include other scenarios and effectively inherit their elements; this helps scenario re-use in UCM models.

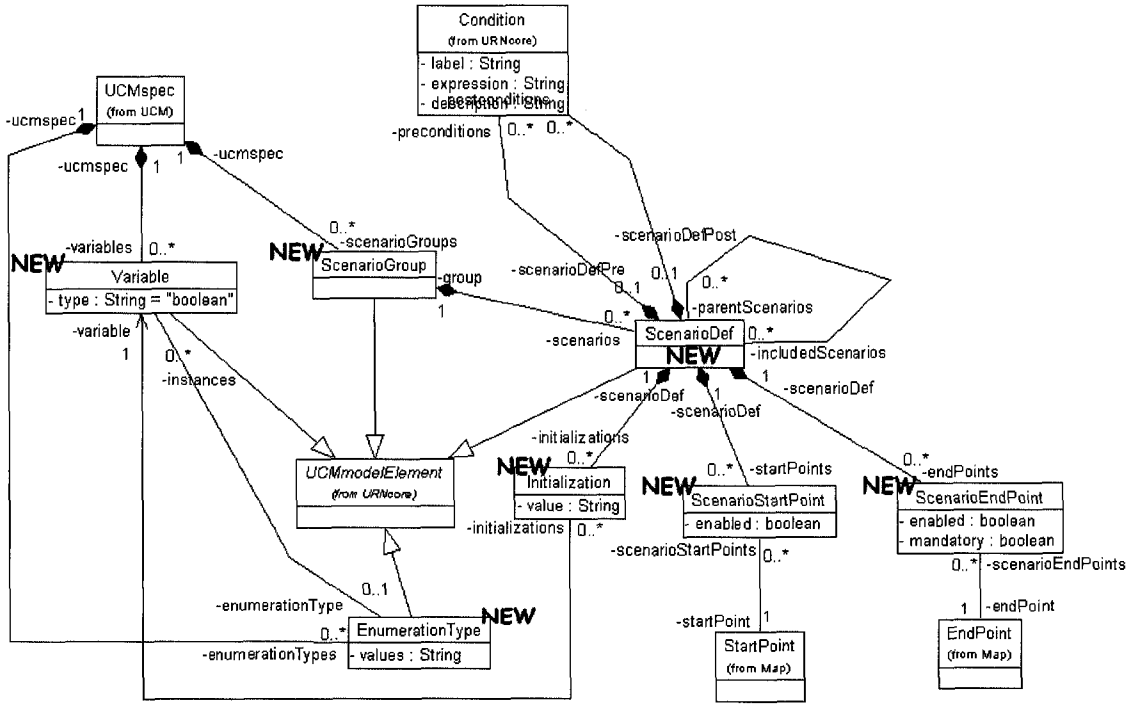


Figure 14 UCM Scenario Metamodel

The concept of inclusion requires a few clarifications. When a scenario includes another, variable initializations are migrated and can be overwritten in the including scenario. Scenario start points and preconditions of the included scenarios are executed and verified first in the execution; their order is imposed by their definition in the included scenario. As for scenario end points and postconditions, they are also verified before the including scenario's addition, although they could be evaluated after; given their nature, the order of verification is not important because all elements will be verified against the same final context.

Not presented in Figure 14 but as equally important are conditions and responsibility definitions. Conditions can indeed be used to define scenario pre/post conditions, but they are also associated with the node connections exiting an or-fork and the selection policy of a dynamic stub's possible plug-in maps. Furthermore, start points have preconditions and end points have postconditions. As for responsibility definitions, they have code that is executed when the traversal engine passes through them. The grammar for conditions and responsibilities is compatible both with Java and SDL and is defined in section 3.3.5.

3.2.3 Review of Related Literature

The concept of scenarios for UCMs was first introduced in the draft URN standard [20], mentioning them as desirable additions and using them in a tutorial. At this point, tool support (based on UCMNav) was under development; the draft standard uses these concepts without actually defining the requirements for scenario definitions or the data model. A working implementation was first introduced in [32]; the scenario definitions, the data model, and the pseudo-code and condition syntax were defined in that paper.

UML 2.x activity diagrams share many commonalities with UCMs and the type of transformation discussed here could be applicable to the generation of sequence diagrams from activity diagrams. Störrle surveyed several transformations from activity diagrams (abstracting from object flows) to different semantic domains [41]. Some are done formally using denotational semantics, others are informal by examples, and others (similar to our approach) are done by algorithm/interpreter. Other synthesis approaches for different scenario notations are surveyed by Liang et al. in [29]. In contrast with many of

these approaches, ours handles path selection based on control variables of different types, scenario models that are hierarchical and/or not well-nested, submodels with multiple input/output segments, and complex component structures.

3.3. Methodology

3.3.1 Visual Overview

jUCMNav's interface consists of various views surrounding the main editor, as is the case with most Eclipse-based editors. An overview of the model currently under design in the central pane is available in the Outline view. Although most changes are performed in the main editor, many smaller tasks (many of which with no graphical representation) are performed in the various views, such as the Properties view. The advent of jUCMNav v2.0 brought along the Strategies view, which is used to execute GRL strategies. Editing is performed using the drag & drop metaphor and via multiple contextual menus and wizards. A screenshot of jUCMNav is presented in Figure 15; it features an active scenario in a modified layout of the regular jUCMNav perspective (focus is given to the problems view whereas one would normally show the properties while in this perspective).

3.3.2 Enhancements Made

The impact on jUCMNav's user interface related to the introduction of the scenario traversal mechanism was fairly limited. Most changes are located in the Scenarios and Strategies view (previously named the Strategies view), which preserves the same activation/deactivation user interface metaphor introduced for strategies in jUCMNav 2.0. This view now contains the variable definitions, the enumeration types, scenario groups, and the scenarios themselves. All this information is structured in a tree view and modifications are made using wizards accessible from the contextual menu and using the properties pane. In addition, a standard Eclipse view was imported into jUCMNav: the Problems view. When traversing the model, we log our information messages, warnings, and errors to this view. Some of these messages have quick fixes available while some others simply offer the possibility to navigate to the source of the problem using a simple double click.

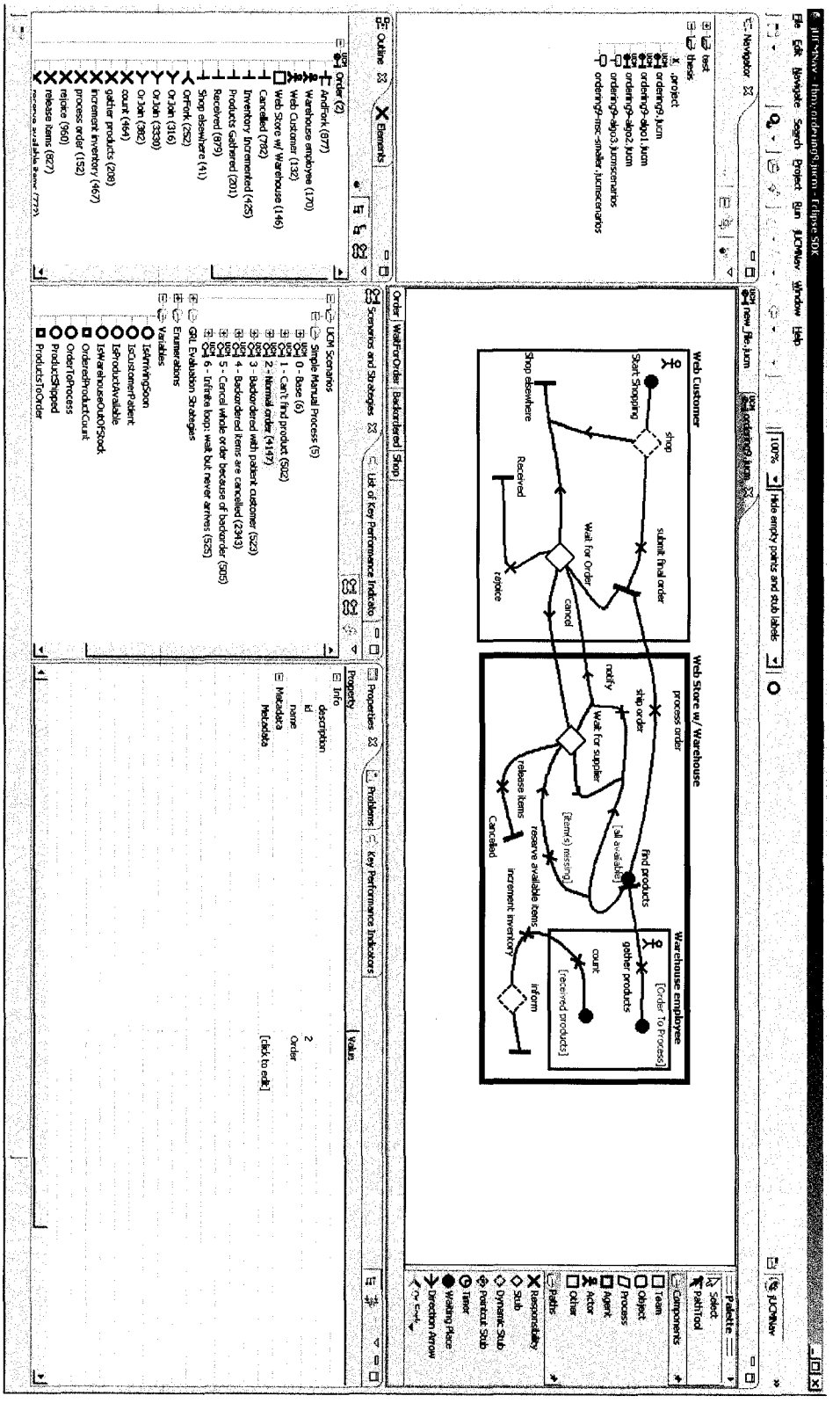


Figure 15 jUCMNav with an Active Scenario

Because designing and analysing URN models do not make use of the same set of views, we added a new execution layout (called a *perspective* in Eclipse). When they enable the execution mode in the Scenarios and Strategies view, a simple click changes the active scenario. The scenario traversal visualization is shown as a red highlighting in the editor, but this color is customizable in jUCMNav's preferences.

In addition to simple contextual menu actions that create default Boolean and integer variables, a variable creation wizard was created (see Figure 16). This wizard reuses elements from the enumeration creator wizard and the variable initialization wizard (see Figure 17). Together, this set of wizards provides a simple yet convenient way to create and manage a set of variables and enumerations in a model and to update scenario definitions as new variables are created.

Finally, we also created a code and condition editor (see Figure 18). Albeit quite simple, this code and condition editor provides a few features that are useful additions to the original UCMNav editor. Immediate feedback is given as the author types in the code or condition, flagging any syntax or typing errors. Authors can insert existing variables with a simple double click or create new ones using the variable creation wizard without having to leave the code editor. Navigation between related alternatives (e.g., in an or-fork) has also been simplified, as well.

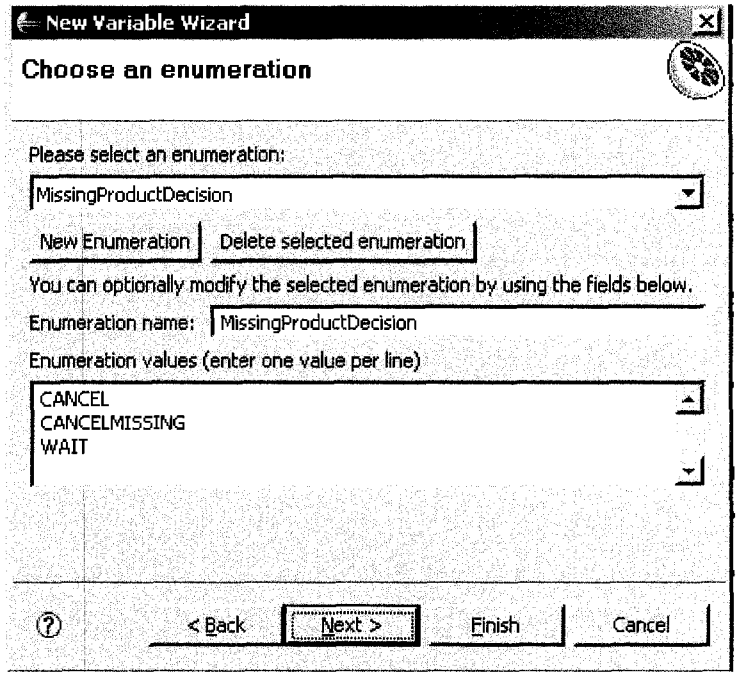


Figure 16 New Variable Wizard

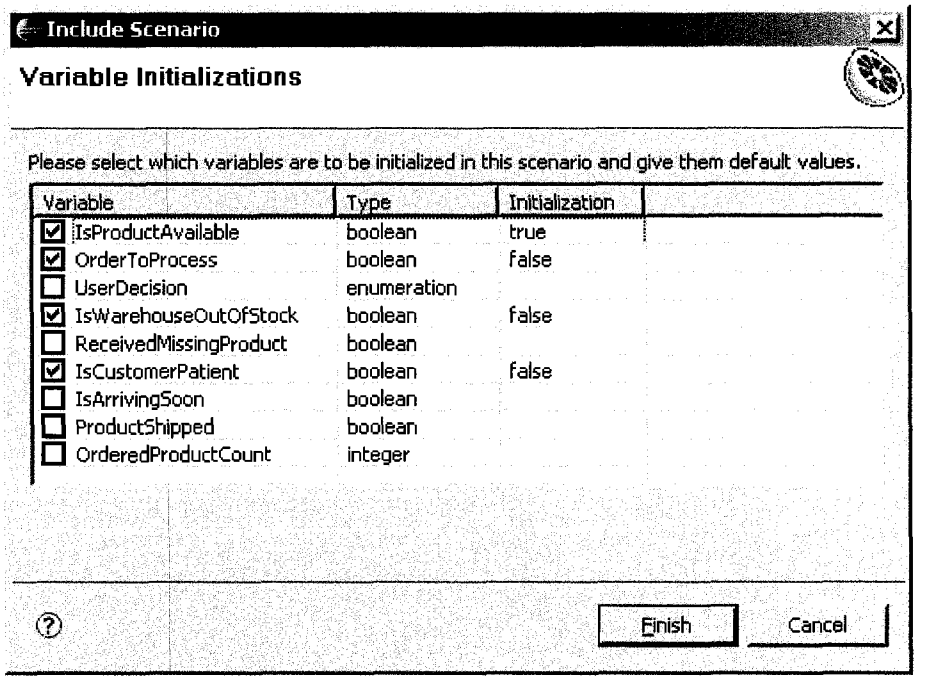


Figure 17 Variable Initialization Wizard

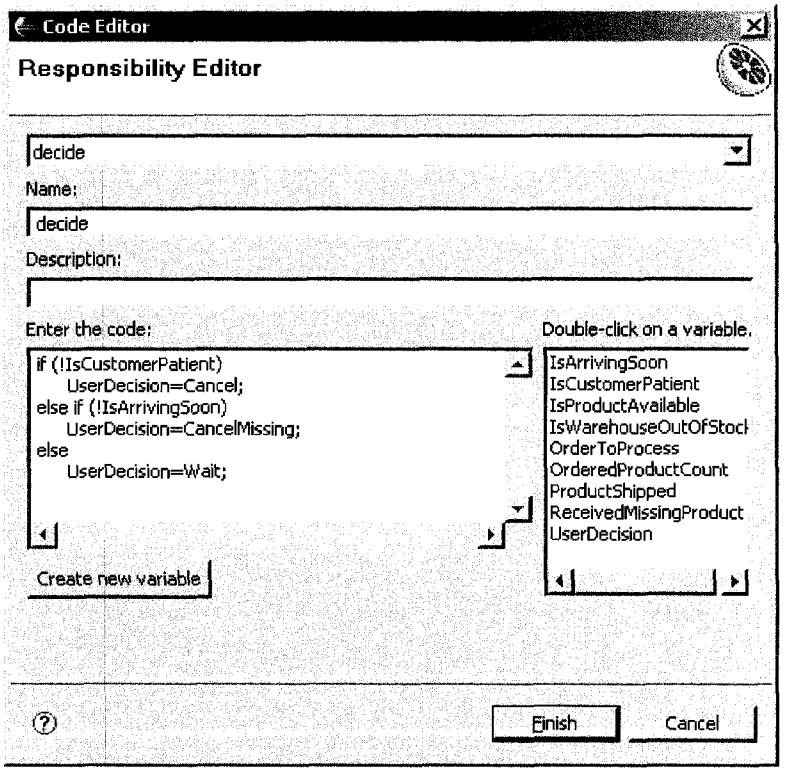


Figure 18 Code Editor

The main feature that was not migrated was a stack view of the traversal algorithm inside the various maps. When drilling down in a stub, a new map was pushed onto

the stack. We did not feel that this was particularly well suited for our initial implementation, given the fact that Eclipse already has a navigation mechanism based on the map traversal history. Admittedly, however, a view that would visualize the traversal mechanism in multiple layers without depending on the user's previous actions would be nice. On a more syntactic level, conditions can no longer express the concept of "else", when all other branches are false. Again, this might be added in the future.

3.3.3 New Scenario Coverage Feature

New in jUCMNav is the capability to run a scenario group or all scenarios with a single click. The results from the execution of each scenario is accumulated and presented to the user as if only one scenario had been run. The modeller can now run all scenarios and look for path segments that are not highlighted and properly augment the scenario suite, which acts as a test suite for the UCM model. Additionally, a new hit count feature was added in jUCMNav, allowing the modeller to know the precise number of times an element or path was traversed (see Figure 19), in addition to the traditional path highlighting. Combined, these features open the door to finer-grained UCM model analysis.

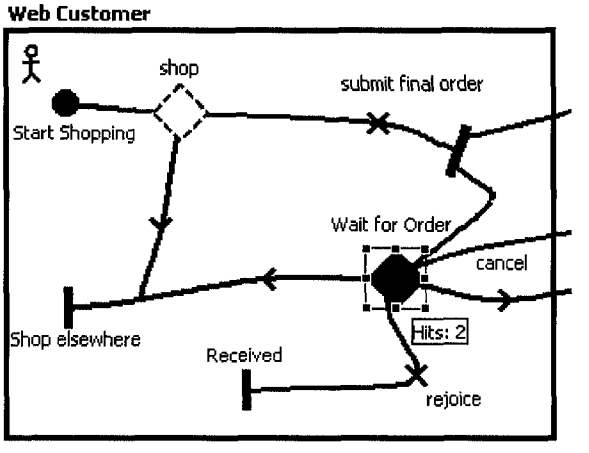


Figure 19 Scenario Traversal Hit Count Tooltip

3.3.4 Architecture and Algorithm Overview

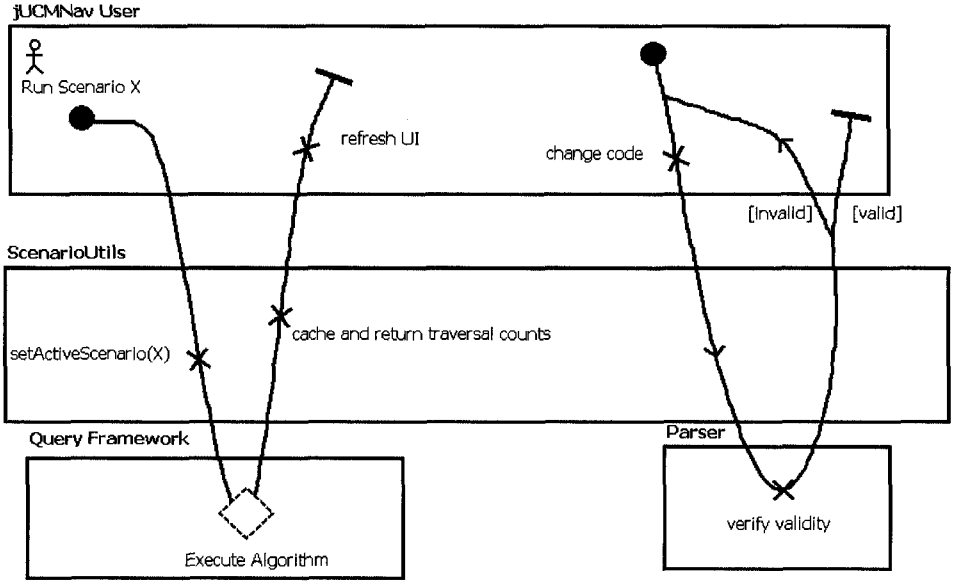


Figure 20 Traversal Architecture Overview

As seen in Figure 20, the scenario traversal algorithm is decoupled from jUCMNav's user interface by using the façade design pattern. Other than in the scenarios and strategies view, jUCMNav's user interface goes through the *ScenarioUtils* class to run scenarios, access traversal results, and parse code/conditions. The underlying implementation touches a fair number of classes in the scenarios and model.utils.modelexplore.queries packages but this complexity is isolated from the rest of the application. Furthermore, this extra layer is responsible for storing the results of the most recent scenario execution, for quick access by the UI. jUCMNav's parser is automatically generated from a BNF grammar (presented in 3.3.5) using JavaCC and JJTree [23].

Figure 20 is fairly abstract, but the class diagram presented in Figure 21 gives a more detailed view of jUCMNav's query framework infrastructure. The actual scenario traversal algorithm is implemented as an *AbstractQueryProcessor* in jUCMNav's query framework, which uses the chain of responsibility design pattern. An *AbstractQueryProcessor* receives a *QueryRequest* and return a *QueryResponse*; all *AbstractQueryProcessors* are initialized on start-up and chained together (*IQueryProcessChain*). The head of the linked list is located in *GraphExplorer* singleton. When an *AbstractQueryProcessor* receives a *QueryRequest*, it can either process it or send it to the next processor down the chain.

The *ScenarioUtils* façade receives a request to activate a scenario and creates and initializes a new instance of *ScenarioTraversalAlgorithm* using a *UcmEnvironment*. This new instance is responsible for caching the traversal results for this scenario; a *ScenarioTraversalQuery* is built which represents the scenario to be executed and gets passed through the *GraphExplorer* singleton until an appropriate query processor (in this case the *DefaultScenarioTraversal*) is found. The *DefaultScenarioTraversal* processes the *ScenarioTraversalQuery* and, when the scenario traversal is complete, it returns an instance of *ScenarioTraversalResponse* which makes its way back to the *ScenarioTraversalAlgorithm* where it is interpreted. Information is returned back to *ScenarioUtils* which updates the user interface (scenario highlighting and problems view). The façade design pattern simplifies the interactions with the traversal mechanism.

Because the receiver of the query is decoupled from the sender thanks to the chain of responsibility pattern, developers can create their own scenario traversal engines as plug-ins of jUCMNav. Furthermore, new query processors could conditionally choose to ignore certain requests (possibly require certain metadata to be in place) and jUCMNav would fall back to the default traversal algorithm. jUCMNav's responsibility is to load all query processors and order them as defined in the user preference page; although user preferences have not yet been implemented, they should be an easy enhancement.

The *Execute Algorithm* dynamic stub in Figure 20 currently only offers Figure 22 as a plug-in map, but as discussed previously, other algorithms could be implemented. The default scenario traversal algorithm is architecturally separated in two: *DefaultScenarioTraversal* and *DefaultScenarioTraversalDataStructure* (see Figure 26). The former defines the flow of control in the traversal algorithm and how each path node should be processed, according to the default semantics of each path node. The latter encapsulates data structures such as the stack of nodes that have to be processed and the waiting list (a queue). By using a stack and a blocked node list, we defined a depth-first traversal algorithm. A breadth-first implementation could be trivially added by simply changing the stack to a queue inside the *DefaultScenarioTraversalDataStructure*. Figure 23 and Figure 24 clarify the data structure used by the virtual machine.

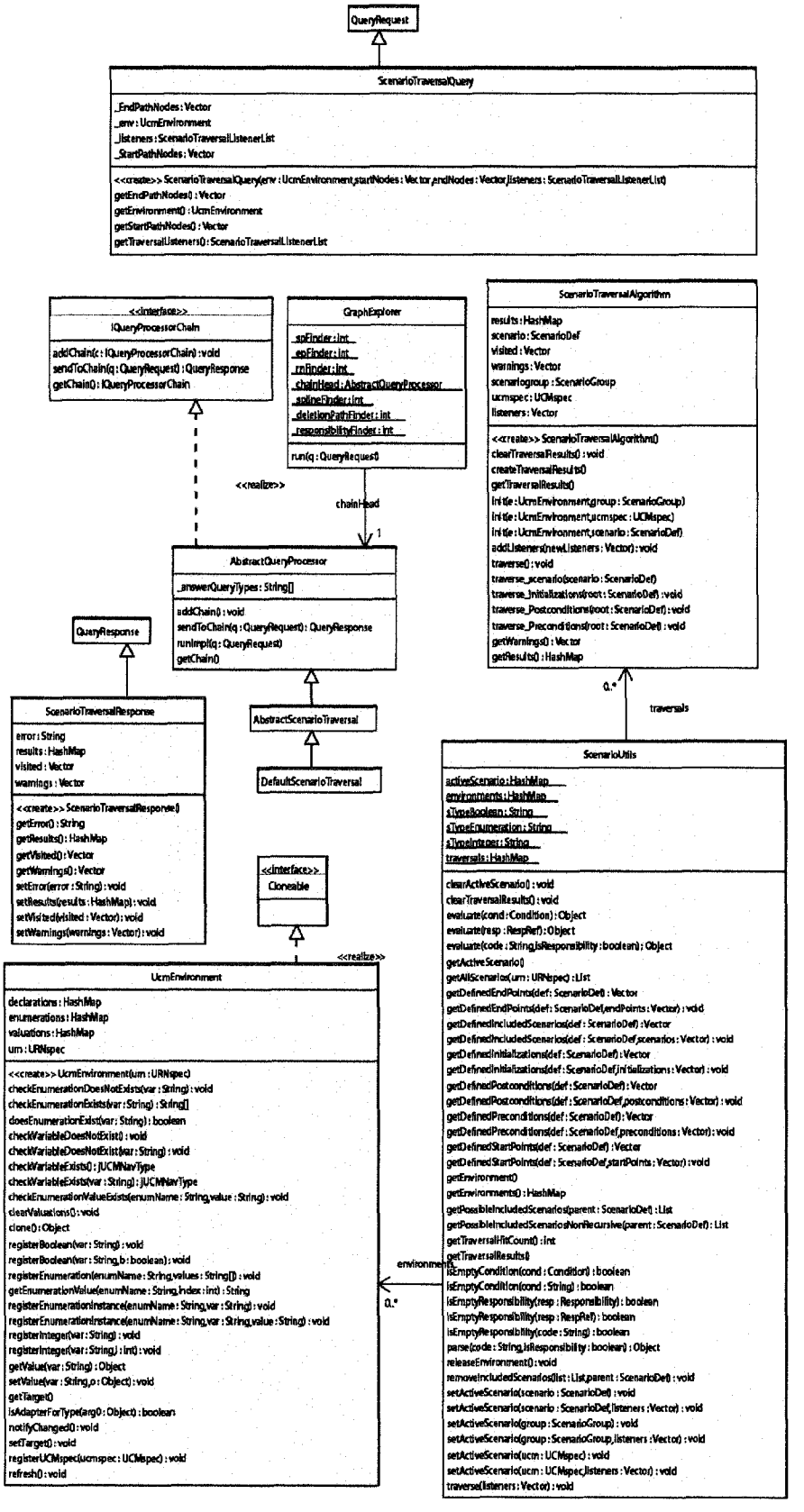


Figure 21 jUCMNav Query Framework Class Diagram

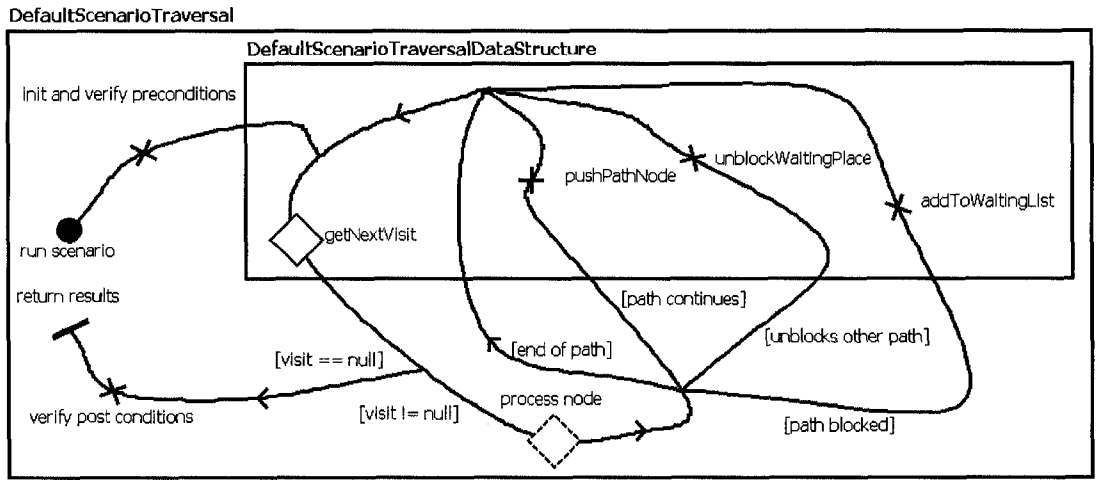


Figure 22 Default Traversal Algorithm

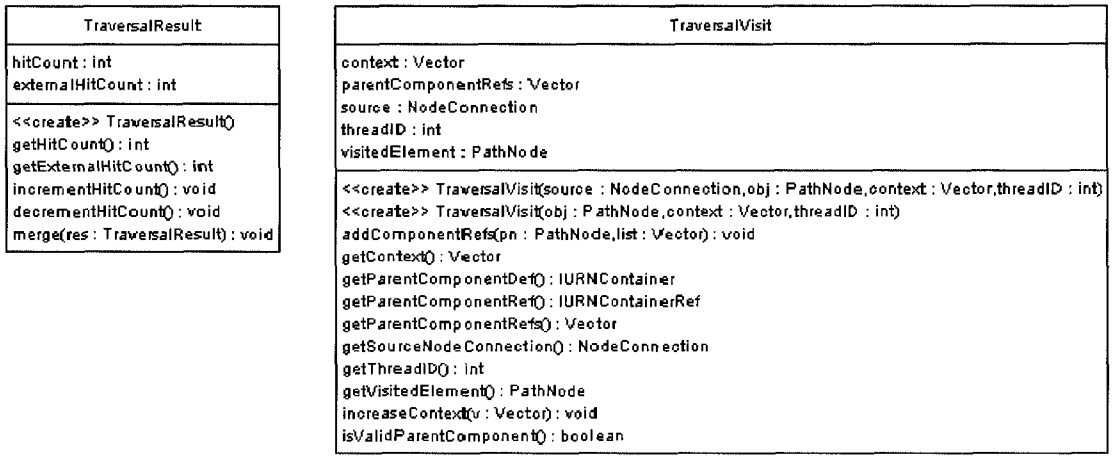


Figure 23 TraversalVisit and TraversalResult Classes

Extensibility being one of the goals here, a new scenario traversal algorithm could be easily created in a number of different ways:

- Create a new *AbstractQueryProcessor* from scratch
- Implement new control flow handler
    - Modify *DefaultScenarioTraversal* and add new user-defined preferences
    - Extend *DefaultScenarioTraversal*
    - Extend *AbstractScenarioTraversal*
- Implement a new data structure
    - Modify *DefaultScenarioTraversalDataStructure* and add new preferences
    - Extend *DefaultScenarioTraversalDataStructure*
    - Extend *AbstractScenarioTraversalDataStructure*

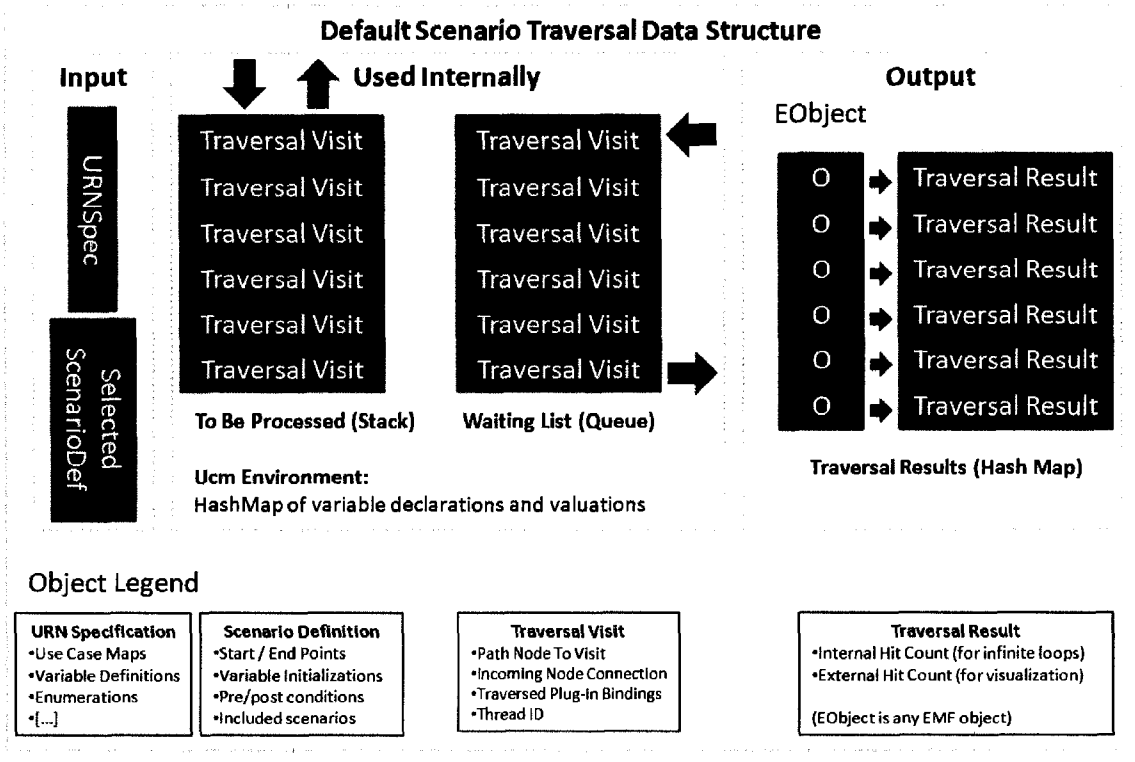


Figure 24 Virtual Machine Data

Hence, this implementation facilitates the most probable extension scenarios. Creating a robust scenario traversal algorithm requires a great understanding of UCMs and their implementation in jUCMNav. The infrastructure that was put in place opens the door for the creation of new algorithms by any modeller.

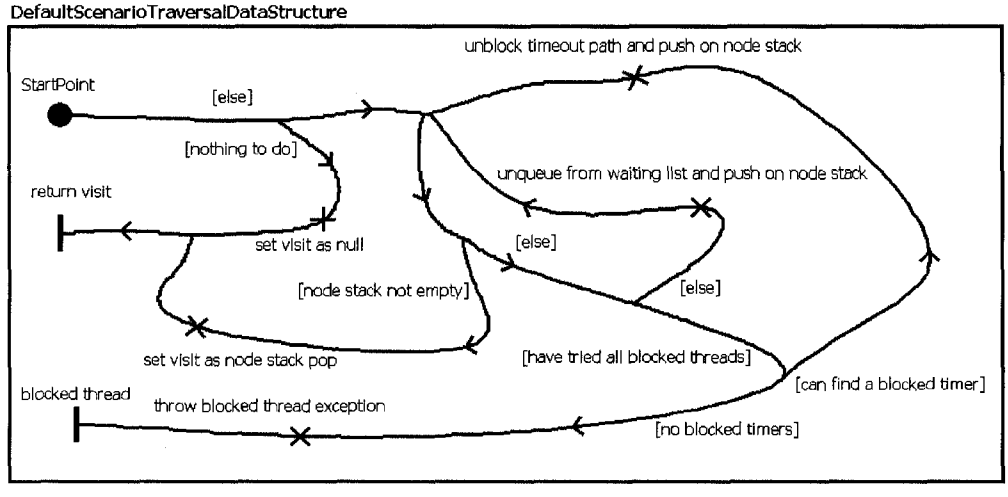


Figure 25 Finding the Next Node to Be Processed

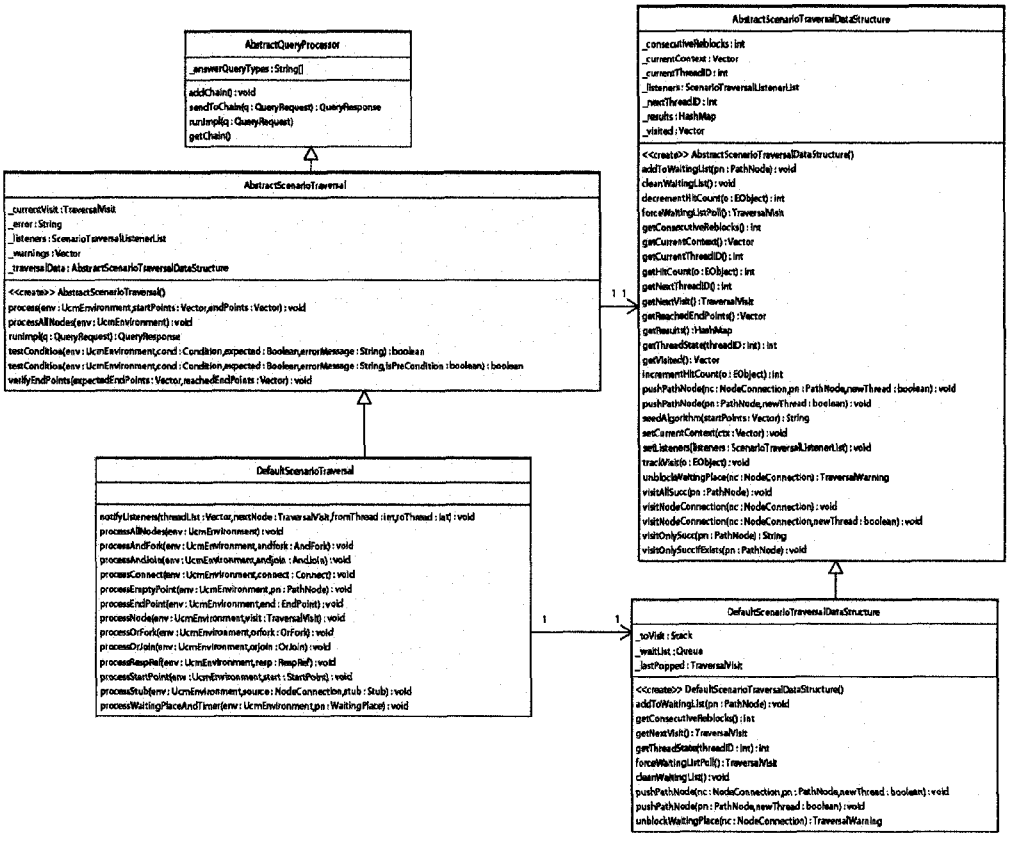


Figure 26 Traversal Algorithm Class Diagram

Figure 25 illustrates the behavior of the *DefaultScenarioTraversalDataStructure*. Detailed explanations about the processing of each UCM path node will be presented in 3.4. As mentioned previously, the scenario traversal algorithm performs a depth first search by utilizing a stack. When a node is processed, subsequent nodes are pushed onto the same stack. (In reality, the algorithm tracks more information that simply the next node to be processed.) As was the case in UCMNav, a waiting list is also used, to hold the path nodes that are blocked during the traversal. An example of a blocking node is the and-join, which must wait for all its incoming branches to arrive before proceeding on the outgoing branch. The algorithm churns along until there are no unblocked path nodes left to process, at which point it verifies whether it can process any of the blocked path nodes. If this is the case, the normal process resumes; otherwise, intervention is required to change the system's state. Because UCMs do not have any concept of time, these blocked states are used as the launch condition for a timer's timeout path. If all timeout paths have been fired and there are still blocked nodes, the remaining nodes are removed from the waiting list and errors are reported.

Although not shown in Figure 25, there are checks for infinite loops where a path will be aborted if it is attempted too many times. The maximum number of visits can be set by users in jUCMNav's preferences.

In summary, the traversal algorithm is similar to that implemented in UCMNav but offers reduced coupling and better extensibility thanks to a few design patterns: strategy, chain of command, and façade [17]. Another pattern that is utilized but not yet mentioned is the observer design pattern. During the execution, the scenario traversal algorithm informs listeners of its progress. The message sequence chart (MSC) export plug-in discussed in Chapter 4 is implemented as a traversal listener. Once again, focus was given on reducing the coupling between these components; the original UCMNav implementation was short-sighted in this respect.

3.3.5 Grammar, Parser and Type-Checker

This section describes the data model used in jUCMNav. The first iteration of the work presented in this section was performed as a course project in CSI5110: Principles of Formal Software Development. It was later augmented to support additional data types. As mentioned in 3.3.3, the parser was automatically generated from a grammar in Backus-Naur Form (BNF). jUCMNav supports Boolean variables, integer variables, and variables of user-defined enumeration types whereas UCMNav only supported Boolean variables. All variables are global in scope. There are a variety of supported operators in our syntax:

Booleans

- Compare (equals, not equals)
- Binary operations (and, or, xor, implies)
- Unary operations (not)
- Assignment

Integers

- Compare (equals, not equals, greater than, less than, greater or equal to, less or equal to)
- Binary operations (addition, subtraction, multiplication)
    - Division is not supported.

- Unary operations (additive complement (-VarName))
    - Pre/post-increment/decrement are not supported.
- Assignment

User-defined Enumerations

- Compare (equals, not equals)
- Assignment

As for the concrete notation itself for each of these operators, diverging goals led to an interesting compromise. As jUCMNav is developed in Java and because most of the students using jUCMNav know Java, there was a driving force pushing for a Java-like syntax. However, because URN is undergoing standardization by the ITU-T, an SDL-compatible notation was also desirable. The implemented compromise allows the use of operators from both notations interchangeably. (Currently, one can even mix and match, although that is not recommended for readability reasons). The only conflicting operator is the lone equals sign ("=") which represents assignment in Java and comparison in SDL. Luckily, the context in which the syntax is used (responsibility or condition) only allows for one interpretation.

```
Expression              := Implication
Implication             := Disjunction (Implies Disjunction)*
Disjunction             := Conjunction ((Or | Xor ) Conjunction)*
Conjunction             := Comparison (And Comparison)*
Comparison              := BooleanUnit ((Equals| NotEquals) BooleanUnit)*
Negation                := Not BooleanUnit
BooleanUnit             := Negation | RelationalExpression | BooleanConstant
RelationalExpression    := AdditiveExpression ((GreaterThan | GreaterOrEqualTo |
                             LowerThan | LowerOrEqualTo) AdditiveExpression)?
AdditiveExpression      := MultiplicativeExpression ((Addition | Substraction)
                             MultiplicativeExpression)*
MultiplicativeExpression:= UnaryExpression (Multiplication UnaryExpression)*
UnaryExpression         := (Addition | Substraction)? ("(" Expression ")" |
                             IntegerConstant | Identifier )

And                     := "and" | "&&"
Or                      := "or" | "||"
Xor                     := "xor" | "^"
Implies                 := "=>"
Equals                  := "=" | "=="
NotEquals               := "not" | "!"
GreaterThan             := ">"
GreaterOrEqualTo        := ">="
LowerThan               := "<"
LowerOrEqualTo          := "<="
Addition                := "+"
Substraction            := "-"
Multiplication          := "*"
IntegerConstant         := ["0"-"9"]+
BooleanConstant         := "true" | "false"
Identifier              := ["_","a"-"z","A"-"Z"](["_","a"-"z","A"-"Z", "0"-"9"])*
```

Figure 27 BNF Grammar for Conditions

Figure 27 represents grammar used for the conditions in jUCMNav. The grammar supports Boolean logic and basic arithmetic, as long as the evaluation of a condition produces a Boolean value. Figure 28 presents the augmented grammar that represents the pseudo-code used in responsibility definitions.

```
ResponsibilityAction    := Statement+
Statement               := Assignment | CompoundStatement  | IfStatement
Assignment              := Identifier AssignmentOperator Expression StatementTerminator
CompoundStatement       := "{" Statement* "}"
IfStatement             := If Expression Statement (Else Statement)?

AssignmentOperator      := "=" | ":="
If                      := "if"
Else                    := "else"
StatementTerminator     := ";"
```

Figure 28 BNF Grammar for Responsibilities

The grammars should speak for themselves, but additional comments with regards to types are required. A simple type-checking engine was built to verify that the types are properly used (Figure 29); types cannot be mixed in jUCMNav. There is no convention that a non-zero integer is equivalent to "true" and there is no numerical equivalent for an enumeration value.

```
(Boolean) Implies (Boolean) : (Boolean)
(Boolean) Or (Boolean) : (Boolean)
(Boolean) Xor (Boolean) : (Boolean)
(Boolean) And (Boolean) : (Boolean)
(Boolean) Equals (Boolean) : (Boolean)
(Boolean) NotEquals (Boolean) : (Boolean)
(Integer) Equals (Integer) : (Boolean)
(Integer) NotEquals (Integer) : (Boolean)
(EnumerationName) Equals (EnumerationName) : (Boolean)
(EnumerationName) NotEquals (EnumerationName) : (Boolean)
Not (Boolean) : (Boolean)
(Integer) GreaterThan (Integer) : (Boolean)
(Integer) GreaterOrEqualTo (Integer) : (Boolean)
(Integer) LowerThan (Integer) : (Boolean)
(Integer) LowerOrEqualTo (Integer) : (Boolean)
(Integer) Addition (Integer) : (Integer)
(Integer) Substraction (Integer) : (Integer)
(Integer) Multiplication (Integer) : (Integer)
( (Boolean) ) : (Boolean)
( (Integer) ) : (Integer)
( (EnumerationName) ) : (EnumerationName)
Substraction (Integer) : (Integer)
Addition (Integer) : (Integer)
If (Boolean) (void) : (void)
If (Boolean) (void) Else (void) : (void)
(Boolean) AssignmentOperator (Boolean) : (void)
(Integer) AssignmentOperator (Integer) : (void)
(EnumerationName) AssignmentOperator (EnumerationName) : (void)
{ (void) } : (void)
```

Figure 29 Type-Checker Rules

3.3.6 Data Structures and Interfaces

A few implementation level classes are used by the scenario traversal algorithm. These classes are not mentioned in the metamodel and are not persisted once the file is closed. An overview of the different classes is given in Figure 24. *UcmEnvironment* represents the environment in which scenarios are included. Albeit quite lengthy, the class is not very complicated. Its main responsibility is to know about the current valuation of all variables at a certain point in the execution of the scenario. jUCMNav being a simple environment-passing interpreter [16], these valuations start at default values and are changed during the execution, and could even be cloned (implements *Cloneable*) during the traversal to support local variables in addition to global ones.

As seen in 3.3.4, an instance of *TraversalResult* determines how many times an element was passed through during the scenario traversal. This information is used by the user interface to determine path highlighting. For the time being, it only contains a simple integer but more information might be added in the future. Each model element (path node, node connection, map) in the source model is associated to at most one *TraversalResult*. Internally, we also use the *TraversalResult* to interrupt infinite loops.

Also mentioned in 3.3.4 is the concept of a stack of elements that should be process next. It is indicated that we push more than the next path node on the stack: we push an instance of the *TraversalVisit* class. This class contains:

- The next path node that must be visited
- The incoming node connection that was traversed to get to this path node.
- A list of *PluginBinding*s that were traversed to get to this point
- A thread identifier
- [inferred when needed] A list of parent component references that depends on the *PluginBinding* stack.

The first three elements are mandatory to create a valid scenario traversal algorithm. Obviously, the next path node must be remembered (or something equivalent). We track the incoming node connection so that and-joins, for example, know when all their incoming branches have been followed. Finally, the list of *PluginBinding*s is required to be able to go back to the parent map after drilling down. The last two elements are very useful additions that should typically be utilized, but are optional. Their use is made clear in Chapter 4, when Message Sequence Charts are generated from the traversal. Simply put, Use Case Maps support concurrency and the thread identifiers help distinguish these threads. Our implementation is linear and single-threaded, but we keep track of this information for downstream models. A multi-threaded scenario traversal algorithm could also be implemented and setup in the infrastructure described in 3.3.3. The important thing to note here is that the *DefaultScenarioTraversalDataStructure* is responsible for remembering this information when it provides a node to the *DefaultScenarioTraversal* algorithm. It remembers the *TraversalVisit* to be able to re-use the same thread and list of plugin bindings for the next node that is pushed on the stack. Obviously, the information is sometimes changed (when launching multiple parallel threads, when drilling down into a plug-in map or back up into the stub), but this behaviour is encapsulated inside the *DefaultScenarioTraversalDataStructure.*

Finally, the component reference list represents all containers, going back up the stack of maps. This is a rather controversial subject in Use Case Maps as the semantics of notation are not clear on whether a component that contains a stub should be repeated in the plug-in maps (for clarity) or not (to avoid redundancy). The implementation that is

provided will return the closest component, whether on the same map or not. The repetition of a component on a map and its plug-in will not impact the choice of the closest parent.

Our implementation also makes use of a few interfaces that are of interest here; see Figure 30 or the commented source code in Appendix A. First, *IScenarioTraversalAlgorithm* is the minimal interface that must be implemented by an algorithm that is to be installed in our framework. The interface is very minimalist as it only deals with high-level information sharing concepts.

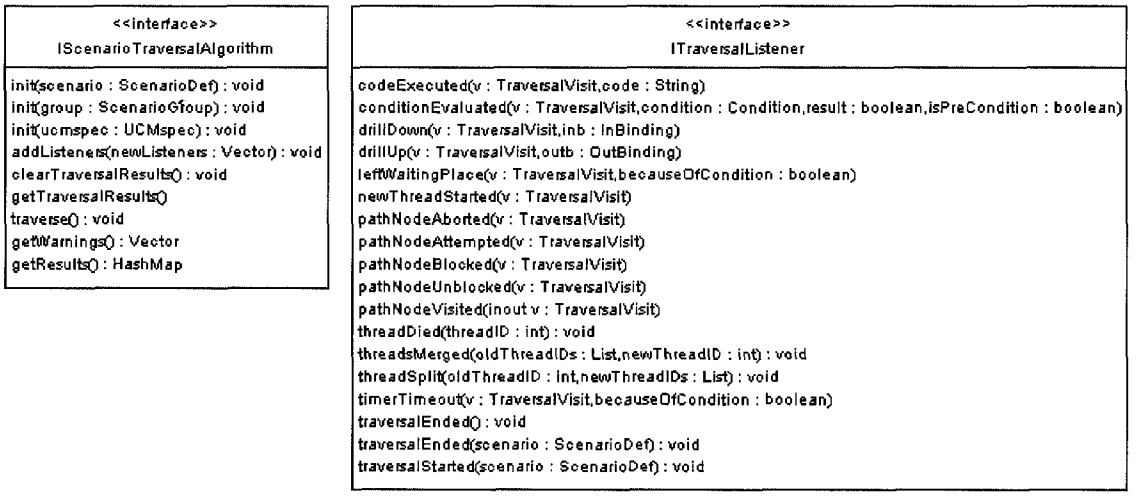


Figure 30 Scenario Traversal Interfaces

Another interesting interface is *ITraversalListener*, which represents the notifications that a traversal algorithm should output during its execution. New traversal algorithms might need to output different kinds of notifications, in which case their listeners would implement an additional interface. Most methods receive an instance of *TraversalVisit* as context information.

3.4. Traversal Preferences

jUCMNav offers a few preferences that influence the traversal engine's behaviour (Figure 31). First, jUCMNav lets the user decide if the algorithm should be deterministic or non-deterministic at certain choice points. Choosing a deterministic algorithm will ensure that the same behaviour is represented every time a scenario is executed whereas non-determinism introduces some pseudo-randomness. Second, the algorithm can be set

as patient or impatient. If the algorithm is patient, it will wait for preconditions to become true (as long as it has something else to do); otherwise it will stop abruptly and inform the user of a problem. This accommodates two different modeling styles: patient algorithms are less formal and allow the modeller to represent event sequencing without having to resort to complex constructs. A third preference is available but it is not related to the semantic interpretation of a UCM: during a scenario's traversal, an infinite loop can be encountered. jUCMNav uses a simple hit count strategy to determine stopping point. Typically, the value is low to accommodate for slower computers and simpler models, but it can be raised as necessary. The last preference allows for the automatic creation of scenario variables that correspond to the evaluation values of intentional elements found in the GRL part of the URN model. Again this one does not influence the semantic interpretation of UCM models from an algorithmic point of view.

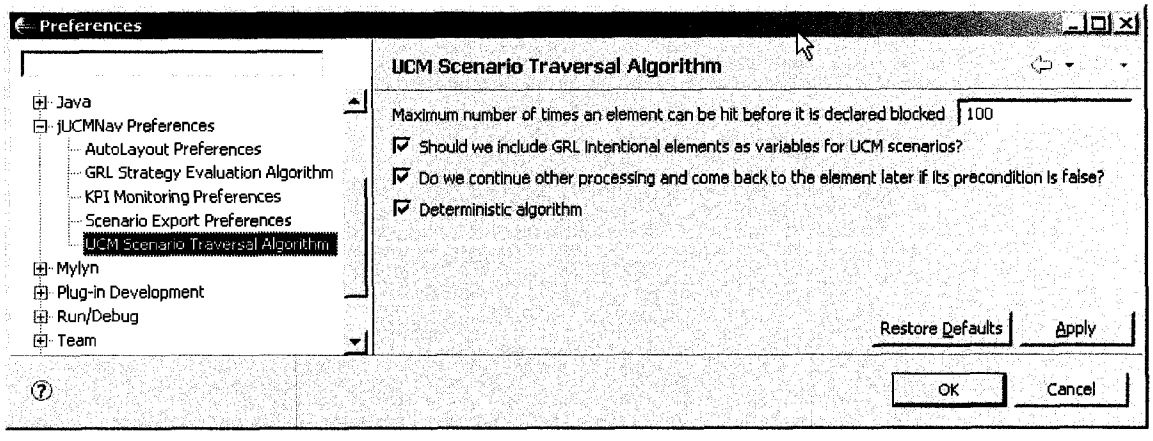


Figure 31 UCM Scenario Traversal Preferences

3.5. Individual Path Node Traversal

On a per UCM element basis, this section describes the actual implementation of the default scenario traversal algorithm, not necessarily what *must* be implemented by any traversal engine. Decisions were made during the implementation of this algorithm and the main ones are illustrated as semantic variations in section 6.2. For increased readability, the semantic variations were excluded from this section but references are made where appropriate.

3.5.1 Start Point

Because the start point is the first path node described in this section, let us discuss how start points are launched. Subsequent path nodes will only discuss what should be done when the path arrives at the path node, not how it got there. Although the initial start points are defined in the scenario definition, there are a variety of additional ways they can be reached. A start point can be launched when the traversal drills down a stub and goes into a plug-in map, via the plug-in bindings. Additionally, a start point can be connected to an end point (synchronous connect) or an empty point (asynchronous connect); when the other path arrives at the connect element, the start point is launched.

Start points have at most once successor therefore their traversal is simple. The only complication concerns what should be done when the start point's precondition is false. jUCMNav preferences (see 3.4) determine the behaviour in this case. If the algorithm is configured to be patient, it pushes the node onto the waiting list to be processed later and continues on with other work. If there is nothing left to be done, an error will be raised as would have been the case if the algorithm were impatient.

See 6.2.1: Semantic Variation #1: Start Point Preconditions

3.5.2 Simple Nodes and Asynchronous Connects

Both or-joins and direction arrows only have one outgoing path. The path is followed and the next path node is launched. They are the simplest elements to process in the traversal. As for empty points, they have one added complexity: asynchronous connects. An empty point can be connected to a start point. This means that when the traversal reaches the empty point it launches, in parallel, the connected start point, and continues on its own path. A small side note: jUCMNav implements the connect element as a regular path node (although invisible). Launching the connected start point is actually a two-step process. First, when we arrive at an empty point or direction arrow, we simply launch all outgoing branches in parallel. (When there are two outgoing branches, we have encountered an asynchronous connect.) This pushes the connect element on the node stack, along with the next path node. In the second step, the connect path node is processed, leading it to the start point.

3.5.3 Responsibility References

A critical path node is the responsibility reference, linked to a responsibility definition, which is the only element that can contain pseudo-code that modifies the *UcmEnvironment*. To process a responsibility reference, we simply evaluate the pseudo-code associated with the responsibility definition in the current *UcmEnvironment*. Responsibility references have an integer repetition count, which indicates how many times the pseudo-code should be executed.

3.5.4 Or-Forks

Or-forks represent the simplest form of alternatives in Use Case Maps. Each of the outgoing node connections is associated with a condition. In conventional UCM wisdom, only one of the branches should be true as or-forks are actually exclusive. In jUCMNav's default implementation this constraint was loosened and the behaviour depends on the algorithm's preferences (see 3.4). As a side note, all incoming and outgoing branches are ordered in jUCMNav. Unless modifications are made to the path, the branch order will remain fixed even when the model is reloaded from the .jucm file. This fact does not impact the visual appearance of a model but does play an important role in the scenario traversal.

If multiple branches are true, a warning will be launched but the algorithm will not abort as was the case in UCMNav. It will always pick the first true branch if the algorithm is deterministic or pick one randomly if it is not. If no branches are true, the traversal engine will stop and error will be reported unless the algorithm is configured to be patient.

See 6.2.2 Semantic Variation #2: Multiple Or-Fork Branches Are True
See 6.2.3 Semantic Variation #3: No Or-Fork Branch Is True

3.5.5 And-Forks and And-Joins

And-forks introduce the concept of concurrency into the model. A very simple algorithm could simply push all of the subsequent path nodes onto the *To Be Processed Stack* and ignore concurrency all-together. However, as stated in section 3.3.6, instances of *Traver-*

salVisit keep track of the current pseudo-thread ID. When an and-fork is processed, all of its outgoing branches are launched using new thread IDs.

As for and-joins, they merge multiple incoming paths into a single outgoing thread. All incoming branches must be traversed for the path to continue past the and-join; if this does not occur an error is raised.

See 6.2.4 Semantic Variation #4: Not All Paths Arrive at And-Join

The default traversal algorithm uses a very simple technique to determine if an and-join can progress: are the hit counts on each incoming branch equal. Until that moment, the and-join remains in the waiting list. Furthermore, we do not allow the same path node to be blocked multiple times: there cannot be multiple TraversalVisit instances representing the same path node simultaneously in the waiting list. These two decisions keep the algorithms simple but open the door to situations where no error is reported yet the hit count on the and-join's incoming branches are not equal. Because of the simple technique, and-joins happen to memorize extra arrivals for future use, albeit losing arrival order that would be preserved by using a queue; they have memory as a side-effect of the simple continuation evaluation technique.

See 6.2.5 Semantic Variation #5: And-Join Memory
See 6.2.6 Semantic Variation #6: Simultaneously Blocked Path Node Instances

3.5.6 Waiting Places and Timers

In the UCM metamodel, timers inherit from waiting places. The main difference is that timers can have timeout paths (timeout paths are optional), but other than that, they are very similar. Conceptually, waiting places and timers are seen as a mix between and-joins and or-forks in the default scenario traversal algorithm. This vision is different from what was implemented in UCMNav, and requires a detailed explanation. Once arriving at a waiting place or timer, the algorithm looks at the conditions on the outgoing paths. Just like or-forks, the outgoing path(s) are associated to conditions. However, the default condition of a node connection that follows a waiting place or timer evaluates to false. If the

condition associated with one of the outgoing branches evaluates to true, the algorithm proceeds down that branch. If both are true, priority is given to the continuation path.

See 6.2.7 Semantic Variation #7: Both Timeout and Continuation Paths Active

As opposed to and-joins, waiting places do not have memory. If the incoming branch arrives when it is not expected (the waiting place or timer is not blocked), an error will be reported. Furthermore, tokens are not accumulated until they are needed. Because the same path node cannot be blocked multiple times in our waiting list, if the waiting place or timer is reached via its normal path while it is blocked, the new event will be ignored. Once the continuation path arrives, the node will no longer be in a blocked state, even if a token-based interpretation of the traversal would have expected different results.

See 6.2.8 Semantic Variation #8: Waiting Place and Timer Memory
See 6.2.9 Semantic Variation #9: Continuation of Unblocked Waiting **Place or** Timer

3.5.7 Stubs

Stubs are the core Use Case Map concept to allow for abstraction, dynamicity and re-use. The scenario traversal engine is currently the only way one can visualize the path cutting through a sequence of maps and, as such, dealing with stubs is one of the more difficult aspects of the traversal. Stubs can either be static or dynamic: static stubs can only contain one plug-in map whereas dynamic ones can contain a set of such plug-in maps. Regardless, the traversal engine treats them very similarly. The link between a stub and its plug-in(s) is a *PluginBinding* (see Figure 11 Partial URN Metamodel), which has a binding condition. The traversal algorithm looks at the set of possible binding conditions and decides which one to traverse; this is very similar to the traversal of an or-fork.

Indeed, the algorithm behaves the same way when multiple bindings are active and, as with or-forks, stubs support the patient algorithm user-defined preference when no binding conditions evaluate to true but if this preference is not set, the algorithm stops and reports that it could not proceed with one exception: if a stub has no plug-ins defined, the default scenario traversal algorithm will skip over the stub if it has only one input and

one output. Simple empty stubs can therefore be used as placeholders without blocking the traversal engine. Complex stubs cannot, because they imply some kind of synchronization or forking that must be described in its plug-ins.

When traversing a *PluginBinding*, the associated start point in the plug-in map is fired with an augmented context (it appends the current *PluginBinding* to the list of bindings that were traversed, so that we can drill back up at the end of the path).

See 6.2.10 Semantic Variation #10: Multiple Active Plug-in Bindings
See 6.2.11 Semantic Variation #11: No Active Plug-In Bindings

3.5.8 End Points

End points represent the final nodes on a path. However, they do not necessarily imply the end of the traversal. Indeed, as we have seen in the previous sections, end points might have to drill back up to the parent stub or might be synchronously connected to other elements (start point, waiting place or timer). Synchronous connects are simple; in the metamodel they are directly identified as the subsequent path node. Therefore the code is as simple as traversing a direction arrow. The difficulty when traversing an end-point is to determine which stubs to pass control onto.

The simplest scenario has only one *PluginBinding* in the *TraversalVisit*'s current context. The context is simply reduced (removing the plug-in binding) and the parent stub's outgoing branch is followed. However, the decision is not as easy when we have many bindings in the current context; the default scenario traversal algorithm fires all of them in parallel. However, if the same plug-in binding occurs multiple times in the context, the outgoing branch will only be fired once.

See 6.2.12 Semantic Variation #12: Multiple Plug-In Bindings in Current Context
See 6.2.13 Semantic Variation #13: Same Stub Exit Path Fired Multiple Times

Finally, it is worth observing that end points can have postconditions. Because postconditions are crucial to the validity of a scenario, an error is reported and the scenario traversal algorithm does not continue.

See 6.2.14 Semantic Variation #14: False Postcondition

3.5.9 Initialization and Wrap-Up

In section 3.2.2, we discussed the initialization and wrap-up in great length, mentioning scenario start points, end points, preconditions, postconditions, and included scenarios. Scenario start points are launched in the order they are defined in the scenario definition. This diverges from what is conceptually viewed as the standard UCM interpretation but is in line with what UCMNav offered.

See 6.2.15 Semantic Variation #15: How Are Start Points Launched?

3.6. Errors and Warnings

Throughout the traversal's execution, various warnings are gathered. In UCMNav, an error causes the traversal to completely stop with a modal dialog. jUCMNav is less intrusive as it continues on processing what it can while accumulating the warnings in the Eclipse Problems view.

Table 1 Various Errors and Warnings

Context	Preferences	Severity
Start point precondition is false	Impatient	Error
End point postcondition is false		Error
Scenario should have reached end point		Error
Element has multiple possible parent components		Error
Traversal is permanently blocked and cannot continue		Error
Multiple true or-fork branches	Deterministic	Error
Multiple true or-fork branches	Non-deterministic	Info
Cannot parse pseudo-code or condition		Error
Traversal blocked at or-fork	Impatient	Error
Responsibility repetition count is not an integer		Info
Repetition count is zero; ignore responsibility		Info
No in-bindings defined for a plug-in binding		Error
No plug-in bindings for simple stub		Info
No plug-in bindings for complex stub	Impatient	Error
Multiple alternatives at stub	Deterministic	Error
Multiple alternatives at stub	Non-deterministic	Info
Race condition detected at waiting place		Warning
Cannot find variable or enumeration		Error
Scenario post-condition is false		Error
Scenario pre-condition is false		Error
No scenario start points are defined		Error
Variable is not initialized		Error

3.7. Evaluation Methodology

To verify the correctness of the implementation, jUCMNav's set of unit tests was augmented. For the parser aspects only, over one hundred test cases were created. Generating a parser from a BNF grammar greatly simplifies the implementation, but one must be certain that the BNF grammar is correct. Furthermore, another fifty or so test cases were created to cover the base traversal cases explained in 3.4. These make use of the scenario preconditions, scenario postconditions, reached end points, end point postconditions, and start point preconditions to verify that the traversal behaves as intended. Of course, using the traversal to test itself reduces the test implementation time. In addition, the new commands (see the command design pattern) that were added in jUCMNav (scenario

creation, scenario deletion, variable creation, etc.) are also augmented with pre/postconditions as is jUCMNav's convention. Each command is responsible for testing that the model is in a consistent state before and after its execution (and before/after it is undone). These checks are implemented as assertions and do not impact end-user performance. All of these tests are run after each and every commit by our continuous integration framework.

The automated tests focus on the low-level aspects of the scenario generation, for the most part. As for verifying the high-level behaviour, the Message Sequence Chart (MSC) export plugin presented in Chapter 4 will be of great help. Although primarily implemented as a way to visualize scenario execution using a widespread notation, the exported files provide a manual mechanism to double-check scenario traversal results.

3.8. Chapter Summary

Most of UCMNav's limitations presented here and related to the previous work in [3][5][32] have been addressed in our extension to jUCMNav. More complex and usable data types are available, together with an action language whose concrete syntax is compatible with SDL. Scenario definitions now support post-conditions, expected end points, and start points that can be triggered multiple times. Scenarios can be included in other scenarios, hence improving management and scalability. Visual scenario highlight is supported; traversed paths and elements are shown in a different colour and offer a hit count indicating the number of times they were traversed. Multiple scenarios can be executed, enabling coverage analysis of a set of test scenarios. As we will see in the following chapter, the traversal, linearization, and MSC generation are entirely decoupled, and intermediate UCM representations (with scenarios) can be exported, enabling other types of analysis and transformations. The traversal can be guided by user preferences (e.g., for the required degree of determinism at choice points), and the entire set of algorithms can be overridden by external plug-ins. Various errors and warnings are reported in the standard Eclipse way, and double-clicking them brings the focus on the model element that caused the problem.

Chapter 4. Transformations – MSC Export

This chapter discusses the realization of Tasks B, C, and D introduced in section 1.2.

4.1. Problem Statement

Once a software system has its behaviour outlined from a user's point of view, software engineers can begin detailing the system's control flow, building a gray-box view of the system under design. Although Use Case Maps visually detail the causal scenarios across entities, Message Sequence Charts (MSCs) are still of great use to the system designer. Indeed, the UCM notation is positioned as an early requirement visualisation notation, while MSC are closer to the detailed design. A MSC-like linear view also minimizes the need to switch back and forth between multiple levels of related UCM diagrams to understand a single scenario. Furthermore, MSCs have been used by software engineers for years and are a simple way to express low-level interactions, which are abstracted out in UCMs. Because both notations complement each other nicely, exporting Message Sequence Charts from jUCMNav is a worthwhile addition. Moreover, as mentioned in Chapter 3, MSCs provide a great way to validate the scenario traversal algorithm when linear sequences are generated from the complex UCM models.

4.2. Background, Context, and Related Work

4.2.1 Background and Context

Generating MSCs from UCMs is not a new idea and that capability has been available in UCMNav for many years. A few years after its initial addition, UCMNav was restructured to output abstract scenarios in XML that allowed the creation of MSCs as well as other representations such as UML 1.4 sequence diagrams and TTCN-3 test skeletons. By the principle of low coupling, a separate tool call UCMExporter [43] was created to generate models from an XML file produced by UCMNav.

The Use Case Map notation is good for describing multiple scenarios abstractly in a condensed form and Message Sequence Charts are better for linearly presenting concrete interaction details, so a transformation from UCMs to MSCs is desirable.

The generation of MSCs in jUCMNav has been planned since its inception. In parallel to jUCMNav's creation on the Eclipse platform, a team of computer science students created an Eclipse-based MSC Viewer in the context of their capstone project [10]. The MSC Viewer can read the scenario files generated by UCMNav as well as textual MSCs described using the Z.120 format.

4.2.2 Related Work

Publications abound on the subject of Message Sequence Charts and transformations to and from this notation, but since this chapter concentrates on MSC generation from UCMs, only a few publications are worth mentioning. The pioneering publication is *Deriving Message Sequence Charts from Use Case Maps Scenario Specifications* [32] However, although it gives a nice overview of the context, it does not go into details concerning the actual transformation. The primary source of inspiration for the implementation in jUCMNav is *UCMExporter: Supporting scenario transformation from Use Case Maps* [5]. This article goes into more detail about the architecture, algorithms, and challenges. Finally, another core reference is *Generating Scenarios from Use Case Map Specifications [3]*, which provides critical insight about how to generate well-nested scenarios.

4.3. Methodology

4.3.1 Overview

In a nutshell, the MSC Export mechanism follows the same algorithms and techniques proven by fire in UCMNav. Indeed, MSCs are not exported directly in jUCMNav, an intermediate scenario notation is used (although this time no additional external tools are required). The Eclipse-based MSC Viewer can visualize this notation as MSCs, but could have visualized it as UML 2.x Sequence Diagrams as well.

4.3.2 Relationships Between UCMs and MSCs

The relationship between the UCM notation and MSC standard was explored in great detail in [32]. From the perspective of abstraction, MSCs are more concrete than UCMs because the latter may abstract out inter-component communication which can be explicitly elaborated in MSCs. The common ground where UCMs and MSCs meet in our work is at the UCM scenario execution level. During the traversal, a compact UCM structure is unravelled into a linear form which closely matches an MSC.

Components in UCMs are structural in nature: they define the possible behaviour of a certain entity via its contained responsibilities. This differs from MSCs where the entities (instances) communicate with each other – an execution level behaviour. Instances only exist for a limited amount of time (via the lifeline concept) and can thus be created/destroyed. In addition, any behaviour has to originate from an instance (message or action) in MSCs whereas components are optional in the UCM notation.

UCMs and MSCs both offer alternatives, concurrency, timers, and, in the case of hierarchical MSCs (HMSCs), nesting of scenarios. However, UCMs provide an explicit notation for scenario interactions via synchronous and asynchronous connections. This feature does not exist in MSCs.

Table 2 UCM to MSC Mapping

Use Case Maps	Message Sequence Charts
Scenario	MSC
Component	Instance
Path changes component	Abstract MSC message
Start point / End point, not bound to a stub	Instance self-message
Condition (Precondition, Postcondition, OR-Fork condition, Stub condition) which evaluates to true	Condition
Responsibility	Action
(Asynchronous connection or And-fork), possibly to (And-join or waiting place)	Parallel inline box
Timer	Timer Set
Timer triggering path arrival	Timer Reset
Timer timeout path taken	Timer Timeout

Table 2 (heavily based on [32]) summarizes the mapping between the various constructs of both notations which was chosen as the basis of the implementation pre-

sented in this chapter. It is worthwhile to mention that elements that are bound to a certain component in UCMs will be mapped as elements (action, condition, message) on the related instance. The unbound elements are mapped to a special *Environment* instance that will be created if necessary. Also, the mapping presented in Table 2 is intended to model scenario execution and therefore eliminates all alternatives in the target notation, even if the MSC alternative inline box could have been used. Therefore, the generated MSCs are useful for the understanding of end-to-end scenarios while the UCMs remain more appropriate for the global abstract view. All choice points are modeled as simple conditions where only the true condition is visualized in the target MSC; all conditions that evaluate to false are ignored. Finally, because explicit UCM loops are not currently available in jUCMNav, no MSC loops are generated (even for responsibilities and stubs with repetition counts).

4.3.3 Architecture Overview

The UCM to MSC transformation is implemented as a jUCMNav export plug-in. Similarly to image exports, the functionality is available via the File → Export menu. The export mechanism creates a *.jucmscenarios file, which can be loaded by the MSC viewer, now packaged with jUCMNav. This XML file representation is actually the serialization of an EMF metamodel (see Figure 33 and Figure 34), created during the course of the project. Therefore, MSCs are not created directly by the tool; only abstract scenarios are generated. These scenarios are imported by the MSC Viewer and visualized as Message Sequence Charts: the MSC Viewer input layer was simply augmented to load and parse the serialized model.

As mentioned in 3.3.3, the MSC generator itself is simply a listener to the scenario traversal algorithm. Simply put, the export plug-in executes a (or a set of) scenario(s) with the MSC generator listening to the various notifications (see Figure 91 ITraversalListener) iteratively building the scenarios. Generating an instance of the UCMScenarios metamodel from the scenario traversal is actually a three-step process, which simply re-uses jUCMNav's internal structure and scenario traversal mechanism on the intermediate models.

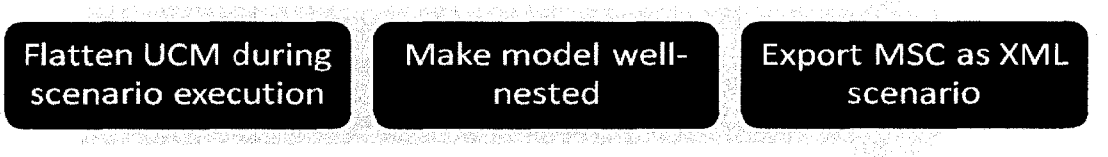


Figure 32 Three-Step Algorithm

Figure 32 represents the three-step algorithm. In the first step, the MscTraversal-Listener generates a new blank URN model. It then listens to the default traversal algorithm and incrementally builds a second URN model depending on what events are picked up. This new model represents a flat view of the scenario execution (no more forks, no more stubs). Each executed scenario is represented in its own map; more details on the actual transformation are available in 4.4. The original scenario definitions are cloned and can be re-executed verbatim on the generated URN model.

Second, the generated URN model is verified to be well-nested according to the definition provided in [3]. If it is not well-nested, the model is transformed and receives additional concurrency constraints to ensure that it can be expressed in a linear form.

During the final step of Figure 32's algorithm, the UCMScenarios model instance is built by traversing the generated well-nested URN model. This is a simple linear process because the previous operations reduced complexity and set the stage for a direct transformation. This output can then be opened and visualized in the MSC Viewer that is packaged with jUCMNav.

4.3.4 Metamodel

The UCMScenarios metamodel presented in Figure 33 and in Figure 34 is heavily based on the XML document type definition (DTD) used by UCMExporter. A number of component instances send each other messages. A scenario definition consists of a set of messages which can be in sequence or in parallel. In addition to inter-instance communication, there are a number of internal messages (called events) that can occur while traversing the UCM scenario.

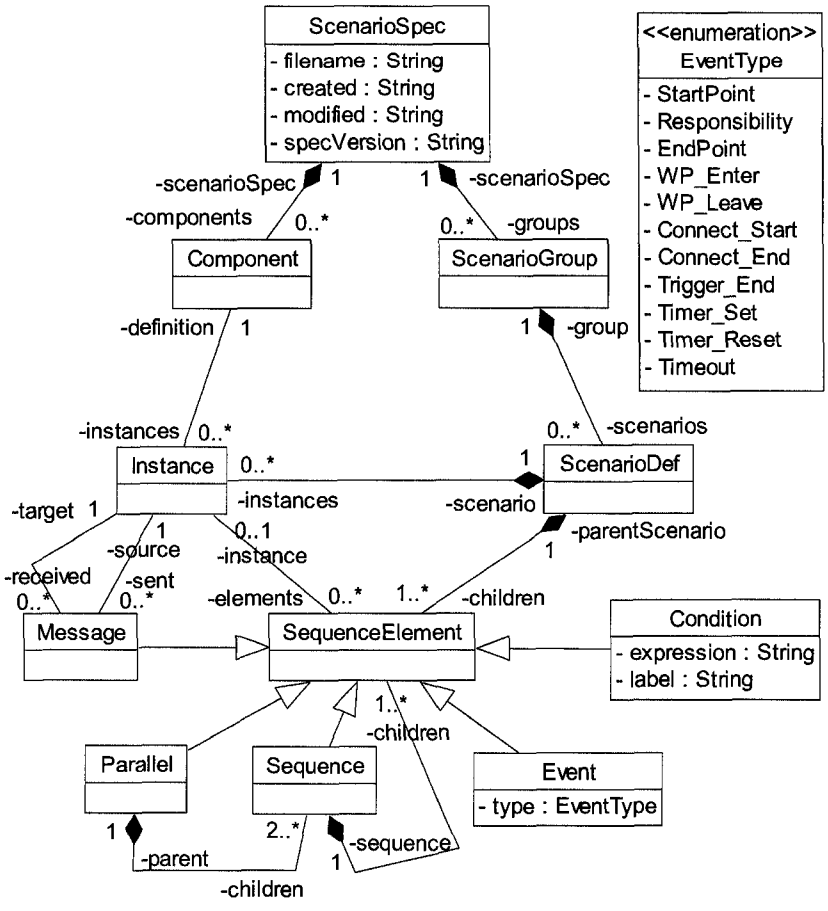


Figure 33 UCMScenarios Metamodel (1/2)

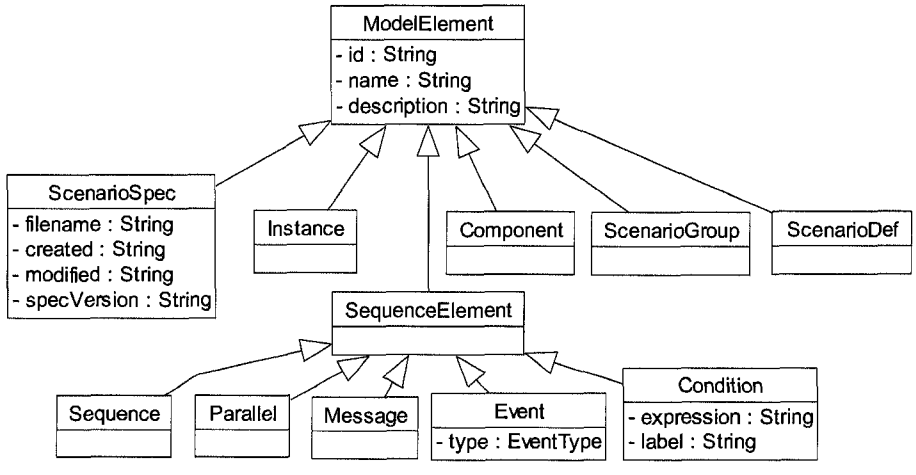


Figure 34 UCMScenarios Metamodel (2/2)

The main difference with the original DTD that was used in UCMNav is that our new metamodel removes a few attributes that were useful only because the scenario export mechanism was implemented as a set of XML transformations (XSLT). This metamodel removes a few constraints on the possible children of sequence and parallel elements, but the changes are syntactic in nature and do not impact the actual scenario expressiveness.

4.3.5 Algorithm for Step 1) UCM Flattening

Algorithm Intent

Given the execution of one or more scenarios, this algorithm generates an equivalent yet filtered URN model. All alternatives are removed and each individual scenario executes within the same map. The generated URN model contains equivalent scenarios which can be executed to replicate the same behaviour.

Algorithm Inputs

- URNSpec
    - Contains multiple related UCMs
    - Contains component and responsibility definitions.
    - Contains variable and enumeration definitions
    - Contains all scenario definitions
- List of scenarios to be executed
    - With proper variable initializations, scenario start points
    - Optionally, scenario end points, pre/post conditions

Algorithm Outputs

- URNSpec
    - Contains component and responsibility definitions.
    - Contains one UCM for each scenario given in input
    - Contains variable and enumeration definitions
    - Contains a scenario definition for each scenario given in input

Algorithm Overview

Figure 32's first algorithm (UCM flattening) is implemented as a scenario traversal listener. When elements are traversed, events are sent out to the traversal listeners and a fresh URN model is created. The data model, scenario groups and individual scenarios are migrated. All UCMs and GRL graphs are discarded, and one new UCM diagram is created for each scenario that is traversed. Table 3 summarizes the actions performed on each event (see Figure 91 for more information on the events). The most difficult part of the translation is pseudo-thread management. The traversal listener is informed when threads are split or merged, but the listener has to remember the context information for each live thread (which end point represents the current position in the traversal for which thread).

Table 3 Traversal Listener Events Used in Flattening

Method	Description
Constructor	Create empty target URN
traversalStarted	If first scenario, clone data model and definitions. Create a new empty map. Clone the scenario information.
pathNode Attempted	If waiting place, create WP_Enter event If timer, create Timer_Set event
pathNodeVisited	If responsibility reference, create respRef. If waiting place, create WP_Leave event If timer, create Timer_Reset event If connect, create Trigger_End event If end point, name the thread end.
conditionEvaluated	If not an empty condition, create a waiting place with continuation path condition. The waiting place receives the condition's label if one existed, or the visited element's name otherwise.
drillDown	Create a Connect_Start event
drillUp	Create a Connect_End event

timerTimeout	Creates a timer.
traversalEnded (env, scenario)	Setup scenario start and end points according to what was generated Get rid of unused component references
traversalEnded	Clear unused information Save the file or move on to next algorithm
threadsMerged	Create an and-join
threadSplit	Create an and-fork

Table 4 Unused Traversal Listener Events

Method	Description
pathNodeBlocked	Not used (will block on condition)
pathNodeUnblocked	Not used (will be unblocked by condition)
pathNodeAborted	Not used (represents error in source model)
leftWaitingPlace	Not used (see pathNodeVisited for waiting places)
codeExecuted	Not used (preserved in responsibility definitions)
threadDied	Not used (see pathNodeVisited for end points)

In the current implementation, the only concept that is directly mapped to a responsibility reference in the target scenario is the traversal of a responsibility reference in the source scenario. Conceptually, all of the other events could be mapped as *empty responsibilities*. However, because jUCMNav do not currently implement this concept, the decision was taken to represent these other events as direction arrows augmented with metadata. Albeit only temporary, this implementation keeps a visual distinction between the different event types.

Any path node that is created in the target model will be bound to a component reference if the source element was bound (either directly or indirectly) to one. An element is indirectly bound if it is not bound to any component reference in its current map, but the traversal engine can infer a parent component by back-tracking its way to a stub containing the current map.

The timer timeout is visualized as a timer in the target map, but does not make use of any of the timer's constructs. This is simply to make it easier to see the timeout event; keeping with the initial concept of no alternatives, the continuation path is always true and no timeout path exists. The conditions that precede a timeout indicate why it occurred; modeling them as timeout conditions would have been possible but would have left unreached end points in the target model.

4.3.6 Algorithm for Step 2) Make Well-Nested

Algorithm Intent

The concurrency found in the output of the UCM flattening is to be made well-nested.

Algorithm Inputs

- URNSpec generated in Step 1) UCM flattening.
    - (No alternatives, no stubs, no synchronous/asynchronous interactions)

Algorithm Outputs

- URNSpec where each path is well-nested.
    - Could be the input, verbatim.

Algorithm Overview

The UCM notation does not impose many constraints on the designer when creating a model. Paths can be divided and merged (via forks and joins) in any fashion, as long as a non-zero set of starting points and a non-zero set of end points remain after the modification. However, this added flexibility does come with a cost: UCMs that are not well-nested cannot be expressed directly as MSCs (expressed in a linear form). The same issue occurs with UML Activity Diagrams, as noted in [38]. The tool should warn the user when a UCM is not well-nested as it does change the meaning of the scenario.

Good examples and explanations on well-nestedness in Use Case Maps can be found in [3], but for the purpose of this section, a few simple examples (Figure 35 and Figure 36) are presented. Simply put, well-nested UCMs have matching forks and joins, although a few special cases are allowed such as the bottom two in Figure 35. The three-branch and-fork could be represented as two consecutive two-branch and forks and be well-nested. The bottom figure represents a situation where a forked path never synchro-

nizes, which is still well-nested (one could also have three start points synchronizing, for example). As defined in [3], well-nested UCMs can be expressed in a linear way with operators for sequence (;), alternatives (+), and concurrency (|), without losing any of the causal relationships found in the original UCM. Alternatively, a well-nested UCM could be defined as a UCM that can be recursively partitioned in concurrent or sequential maps.

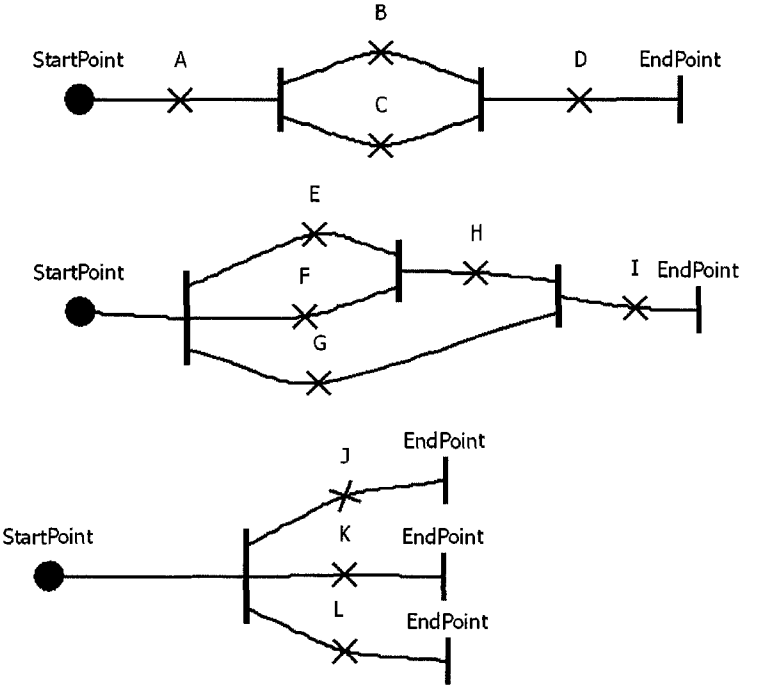


Figure 35 Well-Nested UCMs

As an example, the UCMs presented in Figure 35 could be expressed as:

- StartPoint; A ; (B|C); D; EndPoint
- StartPoint; (G| ((E|F); H)); I; EndPoint
- StartPoint; (J; EndPoint | K; EndPoint | L; EndPoint)

The three cases presented in Figure 36 are not well nested because of a cycle in the UCM or simply because the forks and joins do not match. Had they been well nested, the UCMs could have been expressed textually (linearly) using the notation presented in [3]. As a side note, only the top UCM in Figure 36 could be produced by the UCM flattening algorithm presented in 4.3.5 because the others have alternatives or cycles which are never created by the algorithm.

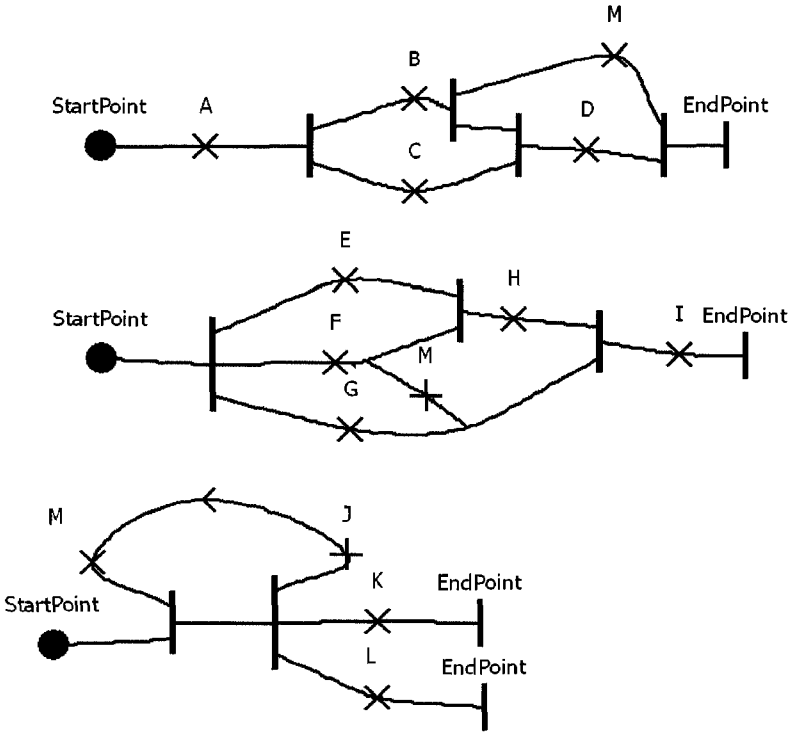


Figure 36 Non-Well-Nested UCMs

A Use Case Map defines a partial ordering of events; it defines temporal constraints on the different elements contained in that UCM. The process of making a UCM well-formed in the context of this chapter can be summarized as adding temporal constraints (forcing certain elements to be executed sequentially instead of in parallel) with the goal of being able to represent this UCM scenario linearly, so that it can be directly converted to an MSC. Obviously, all concurrency could be removed and one sequence of events could be kept. However, because MSCs do support concurrency, that would mean adding more constraints than necessary. Hence, the algorithm imposes a limited number of additional constraints to ensure well-nestedness (note that it has not been proven that this algorithm adds the minimal number of constraints).

The algorithm is implemented as a GEF command that can be run on any URN model. However, in its current state, it only verifies well-formedness with respect to concurrency, as the output of the previous algorithm does not include any alternatives (or-fork / or-join). The algorithm presented here (Figure 37) follows the general concept defined in [3] but the implementation details re-use existing jUCMNav commands to simplify the implementation.

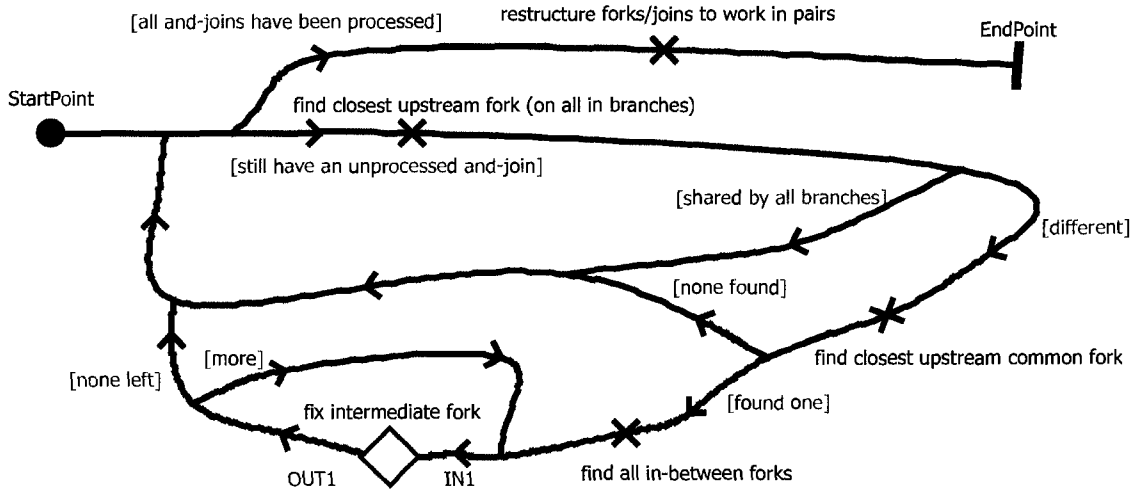


Figure 37 Make Map Well Formed

The algorithm starts by finding an unprocessed and-join and navigating back up all of its incoming branches until it finds the closest and-fork (if there is one to be found). If all branches share the closest upstream fork, the and-join is well-formed and the algorithm continues with the next and-join. If not, if finds the closest and-fork common to all incoming branches: anything between this and-fork and the and-join being processed has to be moved to ensure that the UCM is well formed. The final step consists of re-organizing all forks and joins to simplify the linearization. For example, an and-join with three incoming branches and two of these coming from the same and-fork, the algorithm will group these two into a new and-join that flows into the existing one. This last transformation does not change the meaning of the UCM, it simply restructures elements so that it would be possible to encapsulate the concurrency into a stub with one input and one output.

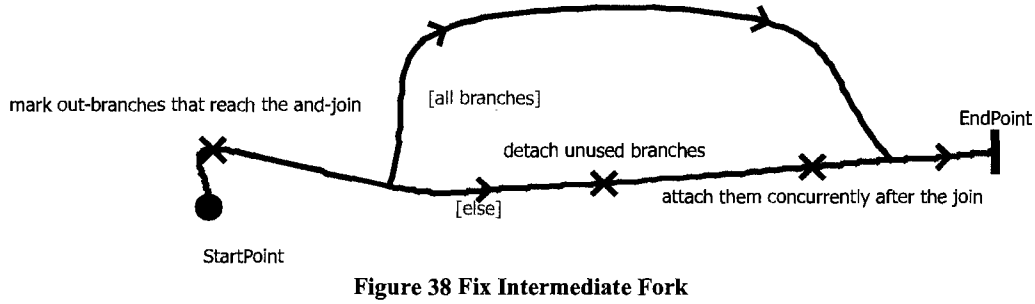


Figure 38 Fix Intermediate Fork

Fixing one of these intermediate forks is surprisingly simple, as can be seen in Figure 38. Heavily based on the restructuring functions presented in [3], the algorithm

simply moves the intermediate branches around. If the intermediate fork's branch does not lead to the and-join, the branch is moved after the and-join. When there are multiple such branches, they are run concurrently. The used branches are left intact and will be regrouped in the final step.

In conclusion, the only additional temporal constraint that is added by the algorithm is the displacement of "irrelevant" branches in the context of a fork/join pair. This is the conceptual equivalent of collapsing a group of nodes in [3]; this algorithm operates directly on a UCM instead of an intermediate generic graph as was the case in the previous work. In any case, the order in which the joins and forks are processed is non-deterministic and therefore non-optimized: the same branch may be moved multiple times. However, it should be noted that moving a branch does not undo any of the work performed by the previous iterations. Concrete examples are shown in 4.4.

4.3.7 Algorithm for Step 3) UCMScenarios Generation

Algorithm Intent

Given well-formed, flat UCMs, generate equivalent scenarios which can be visualized as Message Sequence Charts. Synthesize messages as necessary.

Algorithm Inputs

- URNSpec generated in Step 2) Make well-nested.
    - (No alternatives, no stubs, well-nested)

Algorithm Outputs

- UCMScenarios model instance
    - Each input map is transformed into an output scenario.

Algorithm Overview

The creation of the UCMScenarios model instances from the well-nested linear URN model was not as easy as it was first imagined. The two main challenges were restraining the algorithm to partial-branches while avoiding duplicates during recursion, and synthetic message generation. The first challenge is due to the fact that when we encounter a fork, we have to compute where the path will join back together again (if it does). Knowing that the path is well-formed and does not contain any or-forks or or-joins helped solve

this issue. As for the message synthesis, previous work in [5] drafted the outline of a solution. Because the new implementation is not constrained by a pipe-and-filter architecture, as was the cause with UCMExporter, the implementation was simplified but still revealed some challenges.

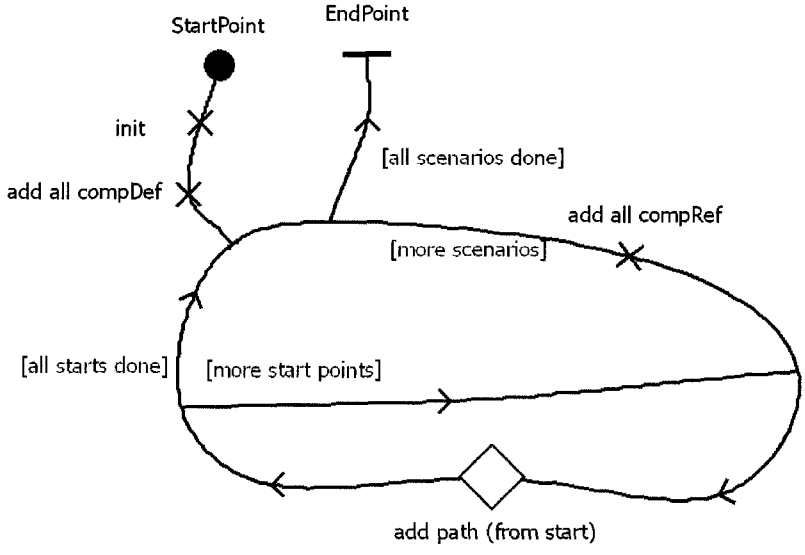


Figure 39 ScenarioGenerator Root

Simply put, the scenario generator iterates through all the scenarios and all of their scenario start points in order to create paths in the target scenario file. As seen in section 3.5.9, the current scenario traversal algorithm treats start points as if they were launched sequentially in the target model.

Figure 39 hides the complexities of the mapping within the add path stub, which is shown in Figure 40. The algorithm re-uses jUCMNav's path traversal infrastructure to read the source UCMs and generate the target scenario model. Each source path node is traversed only once and simply mapped to its corresponding target element. The only complex transformation is the and-fork which creates a parallel element in the target scenario and recursively builds its children (the add sub path stub on Figure 40 calls the Add Path algorithm recursively). As is discussed in section 4.3.6, UCMs are not typically well formed but because of the previous step, the algorithm can proceed with this assumption. When traversing an and-fork, it is known that all outgoing branches will either be reunited at a single and-join or they will be completely separate, because of the final step in

the algorithm presented in section 4.3.6. The synthesize messages stub encapsulates the plug-in map found in Figure 41.

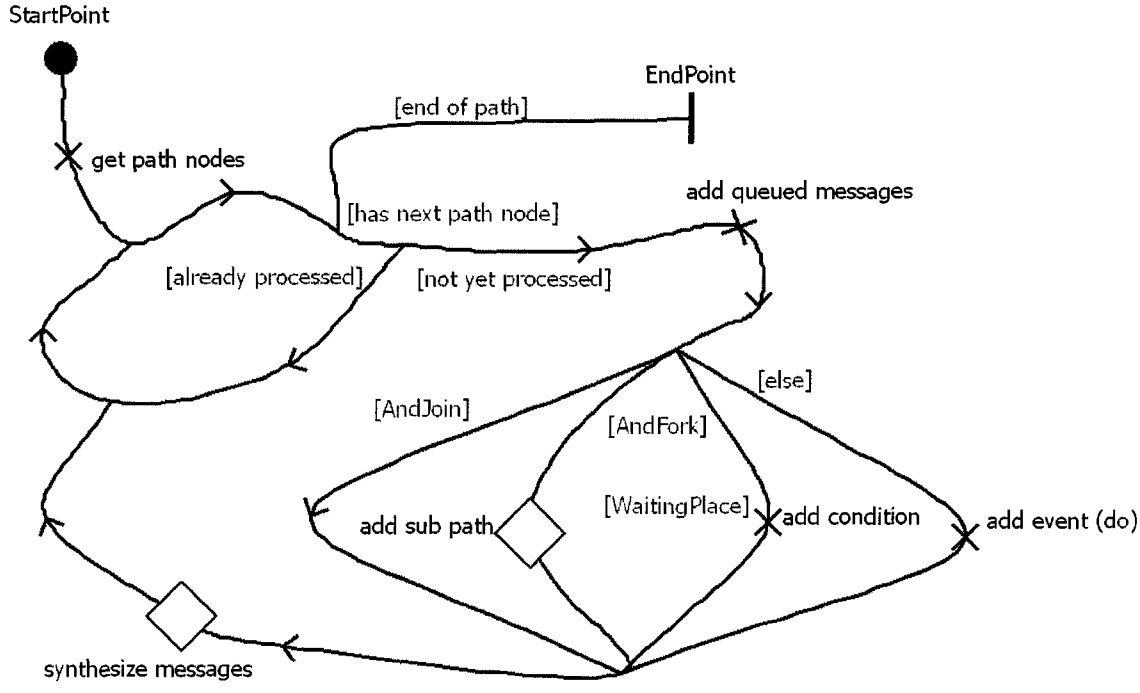


Figure 40 ScenarioGenerator Add Path

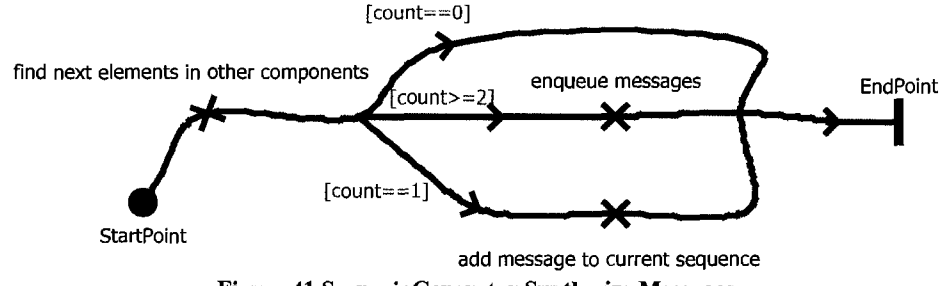


Figure 41 ScenarioGenerator Synthesize Messages

Figure 41 displays the message synthesis aspects of the scenario generator. The work presented in [5] handles four separate cases when determining how to generate messages with concurrency. Because this implementation has access to a powerful UCM query framework whereas the original implementation had to manage with a simple pipe-and-filter XML transformation, the algorithm is greatly simplified. Using this infrastructure, the algorithm can look ahead and determine where the path will lead to next, even considering forks and joins along the way. Therefore, the next component reference on

the path can be compared with the current one to determine if a message should be generated. In this implementation, the actual contents of the messages are not customizable (as they were in UCMExporter), as this is left for future work. The only consideration is the differentiation between one and multiple messages: single messages are added to the target model immediately whereas when multiple messages are required, because an and-fork was traversed, there are queued to be added once the target elements are reached. This behaviour simulates sending the messages in parallel to other instances and maximizes the proximity between the message and the target element.

4.4. Export by Example

4.4.1 Simple Case – No Components

As a first example, Figure 42 shows two UCMs with an active scenario where Boolean conditions select which branch is taken in the or-fork and which plug-in map is chosen at the stub. There are no surrounding components.

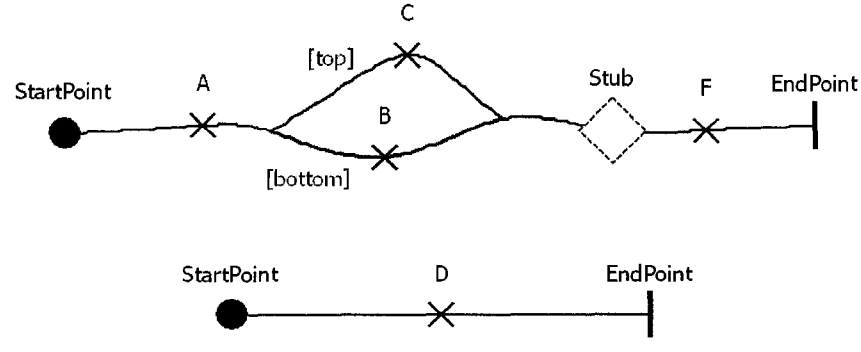


Figure 42 Simple UCM Scenario

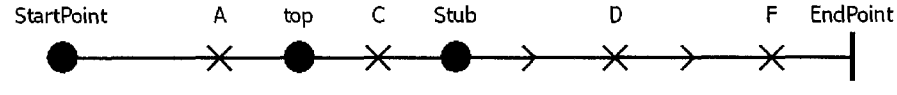


Figure 43 Flattened UCM, Already Well-Nested

Once passed through the first algorithm, the multi-map URN model shown in Figure 42 becomes the single UCM presented in Figure 43. The two waiting places represent conditions which were evaluated to true during the traversal. These conditions are set

on the waiting place outgoing path. The first direction arrow contains meta-data which can be interpreted as drilling down from the stub into the plug-in map while the second direction arrow represents returning back to the stub from the plug-in map.

The second algorithm which ensures the UCM is well nested preserves Figure 43 as it is already well nested. Finally, the scenario generation algorithm generates Figure 44. This is a trivial MSC that makes use of the Environment instance because there are no components and hence no synthesized messages.

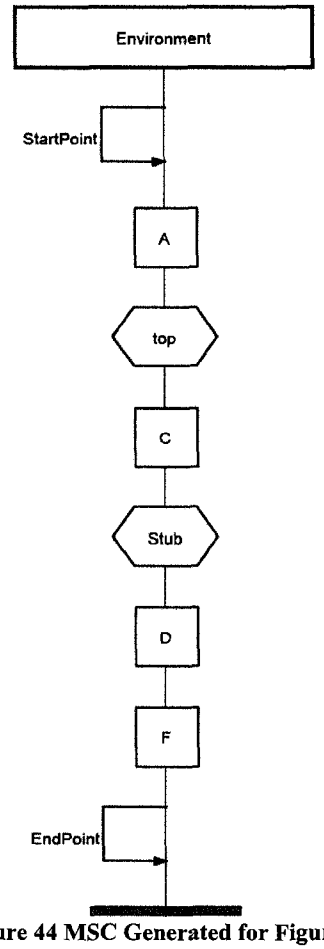


Figure 44 MSC Generated for Figure 43

4.4.2 Simple Case – With Components

This example is very similar to the previous one, except that certain UCM components are added to the source model. Figure 45 displays the same scenario in a different architecture. An interesting twist is that UserB contains the stub, but each individual plug-in map has no knowledge of its parent component.

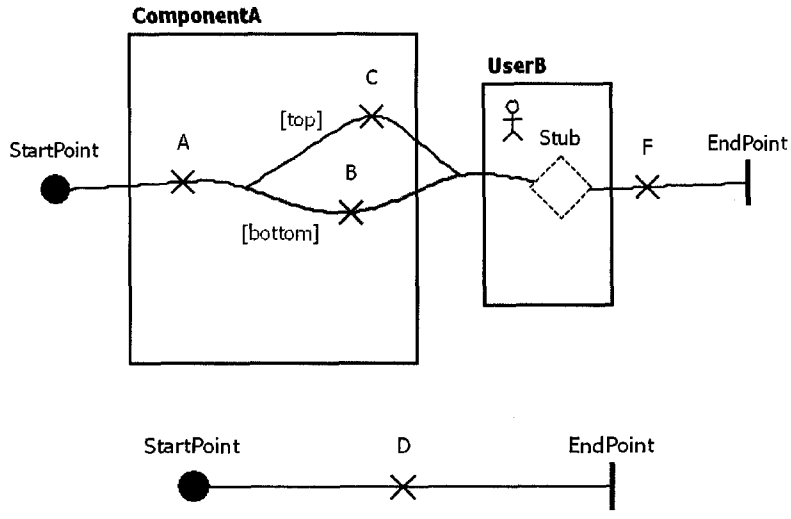


Figure 45 Simple Scenario With Components

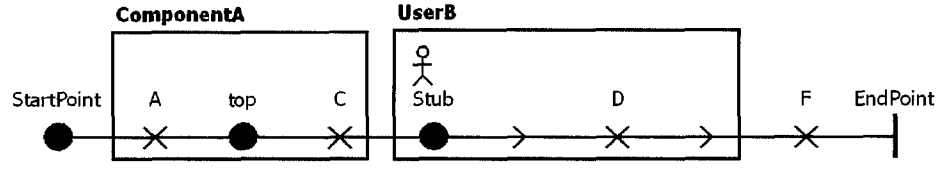


Figure 46 Flattened UCM with Components, Already Well-Nested

As we can see in Figure 46, the generated flat UCM is extremely similar to the one presented in Figure 43, except that the new components are present. It is important to note that even though responsibility D is in a plug-in map, the traversal algorithm's context information allowed the generation of a flat UCM having D inside UserB. Again, the output of the flattening algorithm is already well-formed. Therefore, the second algorithm does not change the model.

Finally, the scenario generation algorithm outputs Figure 47. Here we see message synthesis in action when going from one instance to the next. The message synthesis does not have much information on the content of these messages; therefore they are named according to the next element (condition or action) in the model. Giving appropriate labels to all conditions in the source model is very important as otherwise, as we can see with the Stub condition, a related element (in this case the stub) will be chosen to name the MSC element. Each message that is synthesised by the algorithm is an abstraction of the communication between the two instances; for the MSCs to be useful as pre-

cise specifications, these would be replaced by more elaborate (and detailed) message exchanges.

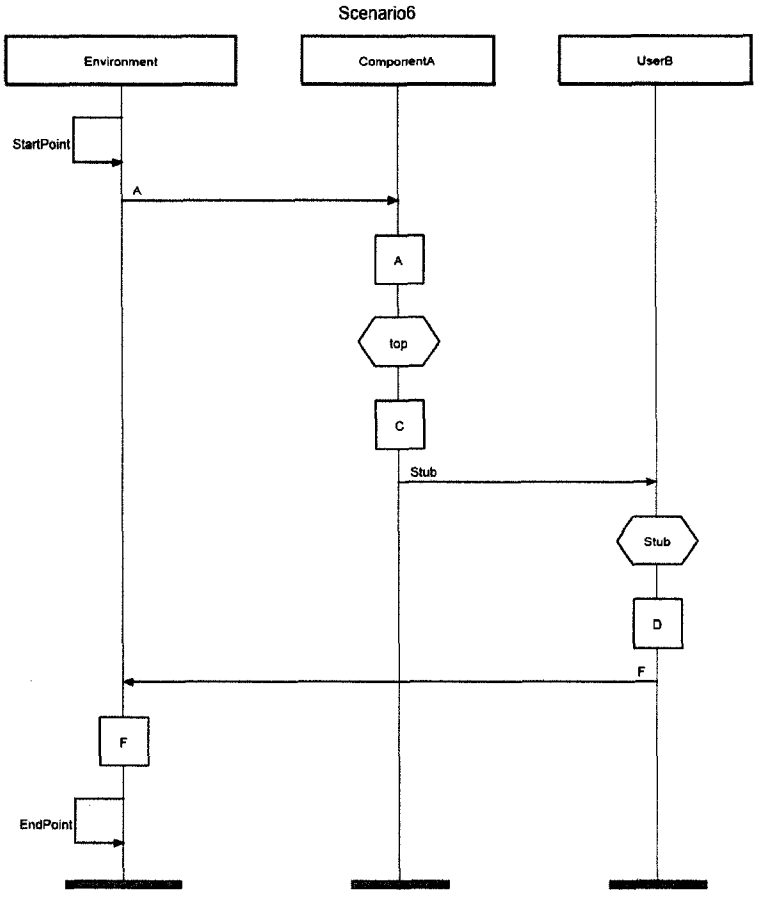


Figure 47 MSC Generated for Figure 46

4.4.3 With Concurrency

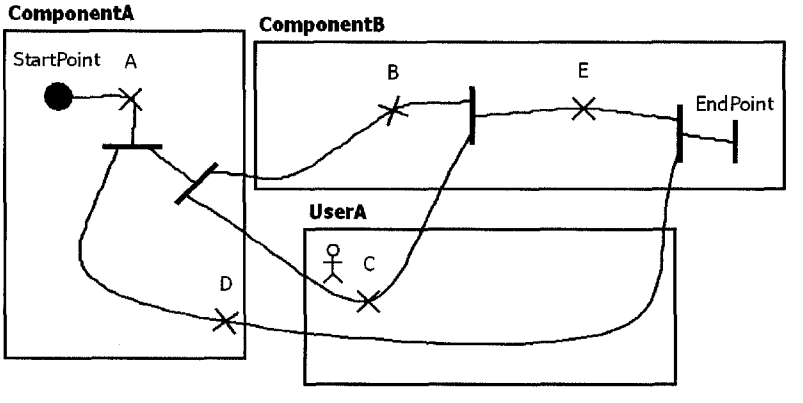


Figure 48 Scenario With Concurrency

Figure 48 introduces concurrency into a new scenario featuring two components, one actor, and a few responsibilities. An interesting thing to note is that although the path leaving responsibility D does cross UserA visually, there are no actions performed. This is interpreted as having no impact on the traversal.

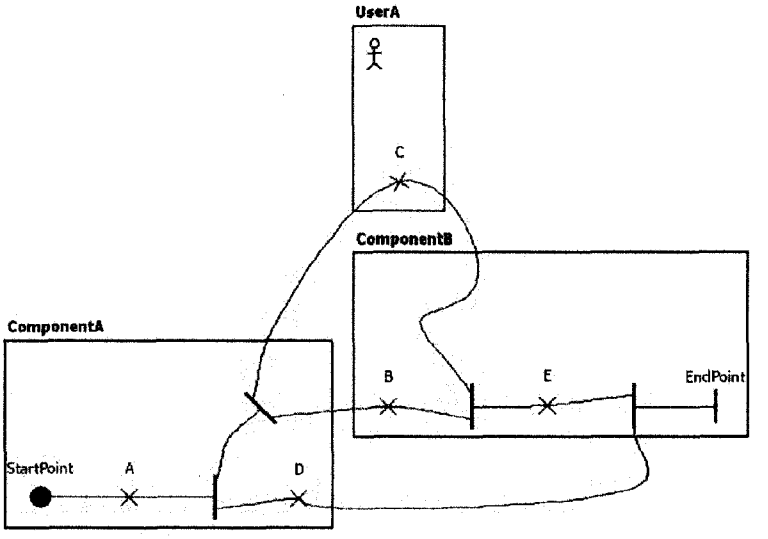


Figure 49 Flattened Scenario With Concurrency - Already Well-Nested

The UCM flattening algorithm does not have much work to do on Figure 48 as can be seen in the algorithm's output in Figure 49 as both UCMs are equivalent. Because the scenarios, data model, and definitions are migrated, the same scenario can be executed and the same result can be observed. Again, this UCM is well nested; therefore no work is performed by the second algorithm.

The generated MSC with parallelism is presented in Figure 50. The scenario starts with ComponentA performing operation A. Then, in parallel, two operations occur. First, another nested parallel block in which ComponentA informs the other instances that they must perform actions. Since this nested concurrency must synchronize in ComponentB before action E, UserA informs ComponentB that it is ready for synchronization. Second, ComponentA performs operation D and informs ComponentB that it is ready to end the scenario. Note that in this MSC, the Environment instance is not used.

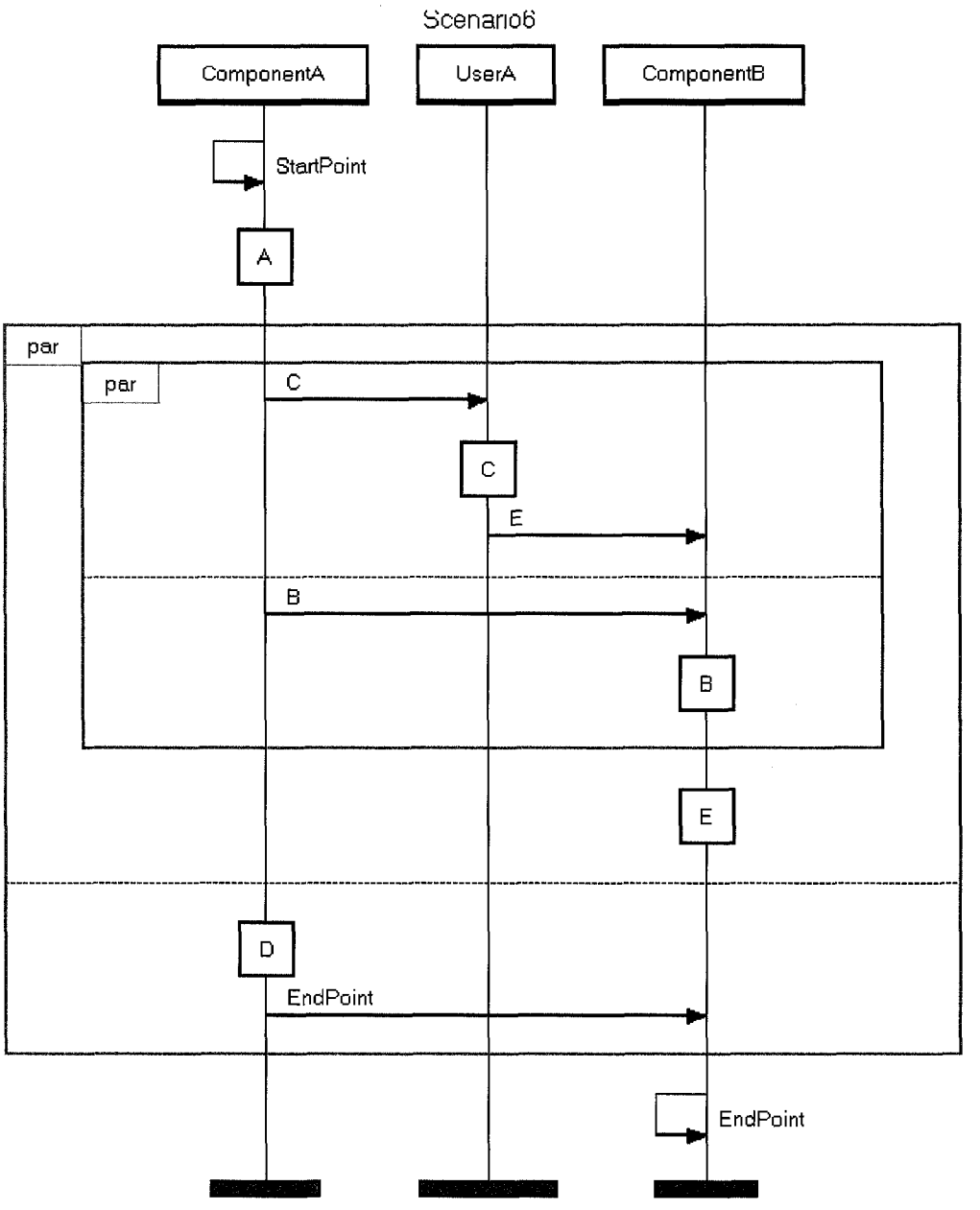


Figure 50 MSC Generated for Figure 49

4.4.4 Timers and Waiting Places

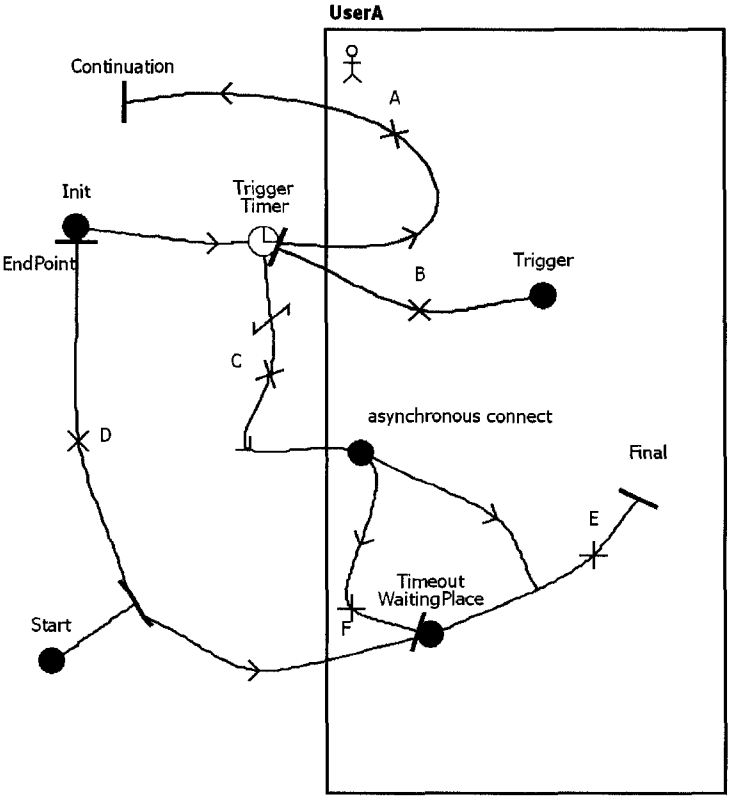


Figure 51 Contrived UCM Scenario

The scenario displayed in Figure 51 contains elements that were left unused in the previous examples: the timer, the waiting place, the synchronous connect, and the asynchronous connect. The asynchronous connect is an empty point that triggers a start point when traversed (it could also fire a waiting place or timer). This actually leads to having responsibility E being traversed twice in parallel rather than only once.

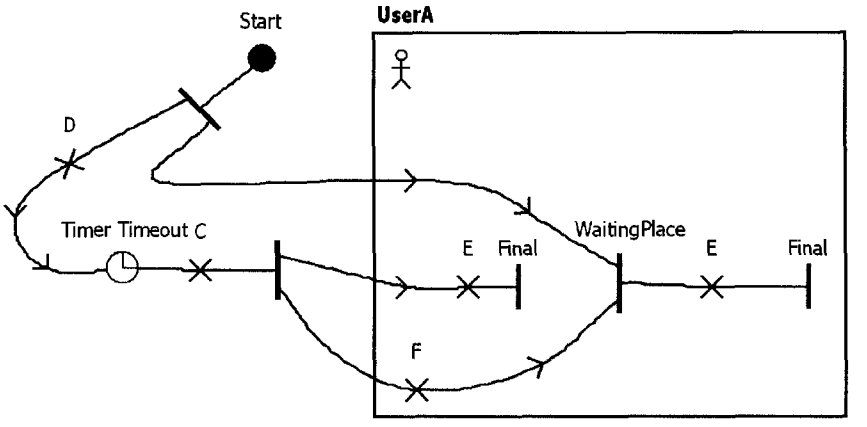


Figure 52 Flattened UCM Scenario with Timers

Figure 52 shows a simpler view of the same scenario, after the execution of the UCM flattening algorithm. The timer element is used to represent the fact that the timer timed out and, as was the case previously, the direction arrows contain various metadata. Furthermore, both the synchronous and asynchronous triggers are recorded in these direction arrows. Although one of the branches of the first and-fork appears empty, it must remain present in the model because it contains important metadata explaining events that occurred in parallel: arrival at the waiting place. Nothing important happens, but we know that parallelism is required to support this scenario (one instance waits for a timer while the other one waits for a waiting place).

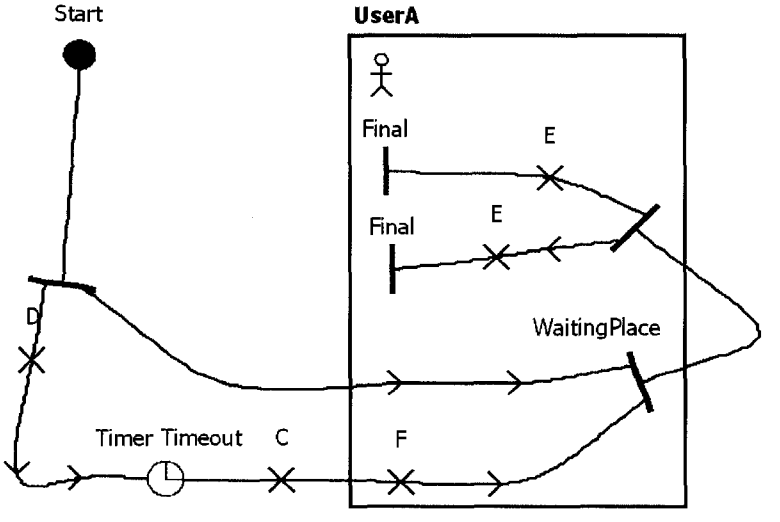


Figure 53 Well-Nested Flattened Scenario

In this case, the scenario generated by the first algorithm is not well nested. Therefore, the algorithm responsible for ensuring well-nestedness transformed the scenario shown in Figure 52 by moving the branch segment located between responsibilities C and F to after the synchronization point. This means that the generated MSC will have the additional constraint that responsibility E is never executed before responsibility F, although the source UCM did not impose this constraint. The well-nested UCM is shown in Figure 53.

Finally, the scenario generation algorithm produces the MSC shown in Figure 54. We see the timer being set and timing out. In this MSC, we see that although we know synchronization has to occur, no actions or conditions are evaluated. Therefore, the synthesized message does not have interesting names. Interestingly, the WaitingPlace message is sent before the parallel block instead of in parallel to the other branch (leaving an empty parallel branch in the MSC). The WaitingPlace message could be interpreted as the Environment telling UserA that it must begin to wait for message F, which it does in parallel to the other events.

4.5. Evaluation Methodology

The scenario export mechanism is manually tested. Being itself a way to test the scenario traversal engine, this was deemed sufficient. Although the MSC Viewer does make it easy to visualize the exported scenarios, not all of the information is used in the context of MSCs. Ideally, some automated unit tests should be created that would run the scenario export on an existing URN model and verify assertions against the generated scenario model instance.

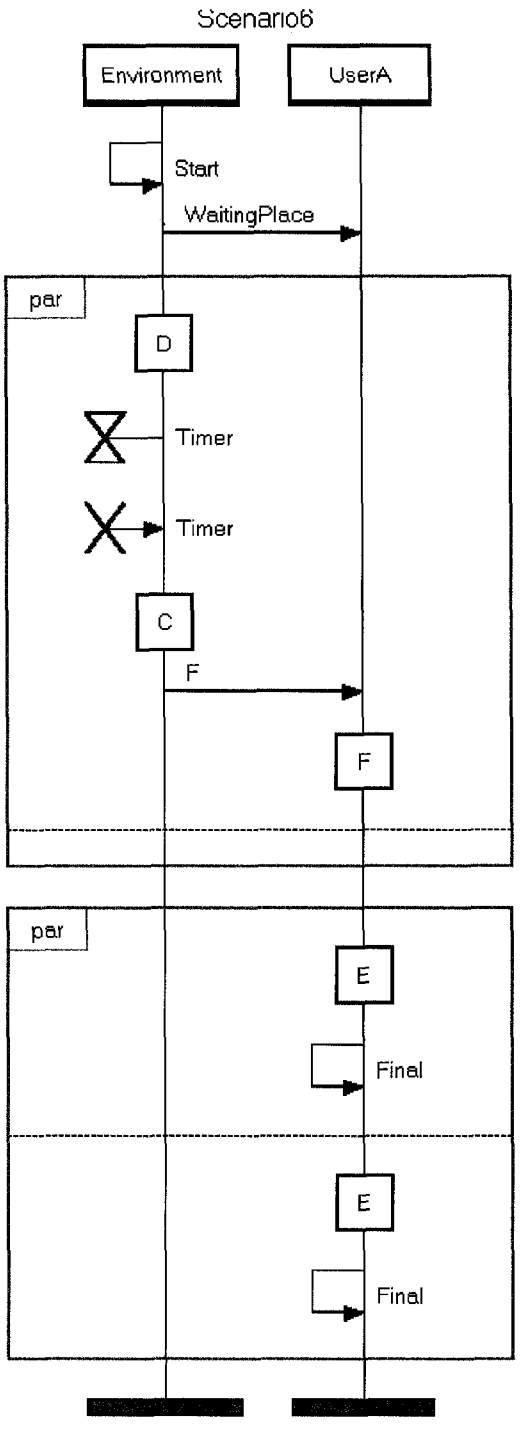


Figure 54 MSC Generated for Figure 53

Table 5 Manually Tested Features / Flattening Algorithm

Construct	Reference
Start point, End point, Responsibility, Stub (Static/Dynamic), Empty Point, Direction Arrow	4.4.1
Components (directly bound and inferred from parent stub)	4.4.2
And-fork, And-join	4.4.3
Waiting place, fired by path arrival or condition (behaves like and-join)	4.4.4
Timer, fired by path arrival or condition	Chapter 5
Timer timeout, condition or lack of arrival	4.4.4
Synchronous connection	4.4.4
Asynchronous connection	4.4.4
Pre-conditions, Post-conditions	Chapter 5

First, the flattening algorithm was manually tested on small meaningless examples. These examples used various constructs that are available in the UCM notation. Such examples are shown throughout section 4.4. The goal was to use each UCM construct or variation of a construct and manually verify the output. Table 5 shows these features and, whenever possible, a reference to a location where the feature is tested in this thesis.

Second, because the algorithm that ensures a UCM is well nested is implemented as a command in jUCMNav, a menu item was temporarily added to transform various hand crafted examples. The examples provided in [3] were tested along with other convoluted situations. With more complex UCMs with interwoven and-forks and and-joins, it quickly became challenging to do the transformation by hand on a piece of paper to compare with the result of the algorithm. A particular case that was found to be problematic was the middle case in Figure 35 where certain and-forks or and-joins need to be transformed into two consecutive elements, to make the well-nestedness more apparent.

Finally, the UCMScenarios generation algorithm was manually tested by taking simple examples like the ones presented in 4.3 and manually verifying the transforma-

tion. Once satisfied by the validity of the simple examples, the robustness of the full algorithm was improved by using various UCM examples made in the past and exporting the scenarios. The main test case was the simple telephone system re-used in various UCM presentations and even as an assignment in software engineering courses at the University of Ottawa. The main challenge was to have consistent message synthesis; typically a message should be sent every time a path crosses a component barrier. With concurrency, the difficulty was to synthesize messages at the appropriate time (whether inside the parallel construct, before it, or after it. The goal was to properly support the four situations which involved synchronization of multiple instances that were defined in [5]. Thanks to the intermediate export steps, debugging was facilitated. Once these base cases were accomplished, other large and complex instances were manually verified. A limitation in terms of scale was identified during this process: very complex UCMs that produce very long scenarios which are repeatedly forked in a loop can produce stack overflows either in the traversal engine or in the MSC viewer. Stack and heap sizes of Eclipse's virtual machine can be increased and the maximum number of loop iterations can be decreased in jUCMNav's preferences to accommodate such situations.

4.6. Summary

This chapter presented the generation of Message Sequence Charts from Use Case Map scenarios. Although some of the work presented here was already accomplished for UCMNav, the new implementation is decoupled from the traversal algorithm presented in Chapter 3 and the visualization engine is now integrated with jUCMNav. Furthermore, because the mechanism's intermediate steps re-use the metamodel and command structure, maintainability is improved. The capability to export flattened scenarios has become a tool feature in its own right, even though it was originally included for debugging purposes.

Chapter 5. Case Study

Along with the work presented in Chapter 4, the case study presented in this chapter helps realize Task D identified in section 1.2. Validating the scenario traversal results using Message Sequence Charts increases the robustness of and our confidence in the traversal algorithm.

5.1. A Simple Web-Ordering Use Case Map

What follows is a simple UCM model representing a web-ordering system that is the basis of this chapter's discussions. Figure 55 shows the top level map representing an abstract view of the system and its entities. Our sample UCM model describes an online e-commerce front-end to a warehouse selling physical products. The company's business processes do not allow it to show real-time product availability on its website; because this process is manual, an unfortunate web customer can order a product that is not available in the warehouse. Should this occur, the web store will inform the user that his order includes back-ordered products. Consequently, the user will either decide to wait for the product to become available, cancel the back-ordered items, or cancel the order completely.

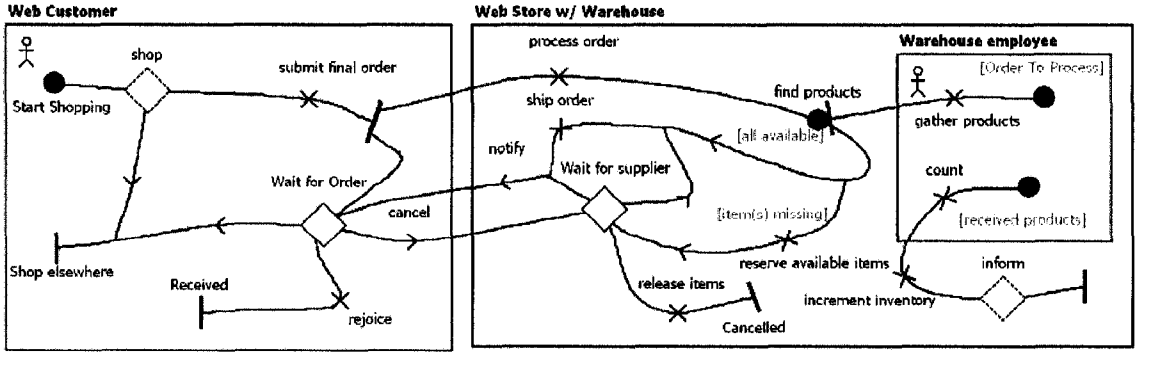


Figure 55 Web Ordering Top Level UCM

Figure 56 shows the plug-in bound to the *Wait for Order* stub in Figure 55 whereas Figure 57 is one possible plug-in for the shop dynamic stub. It is the only plug-in provided in the model but other shopping scenarios could be implemented.

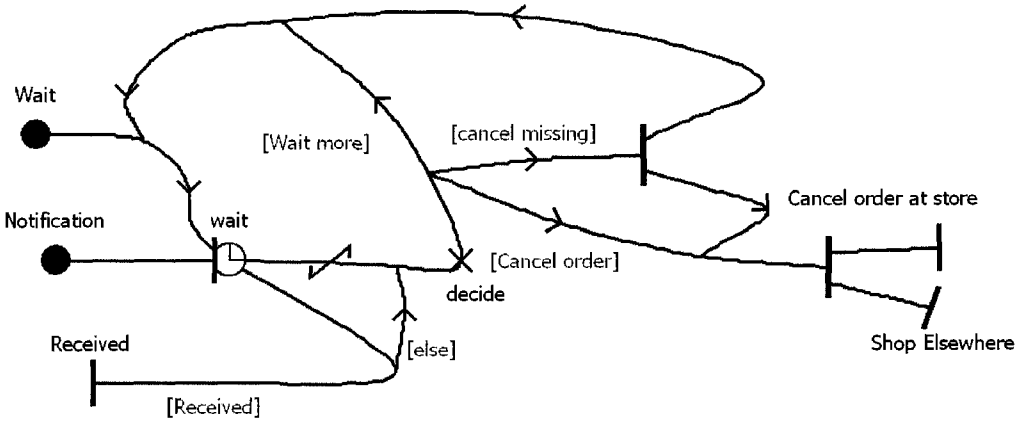


Figure 56 Wait for Order Plug-in

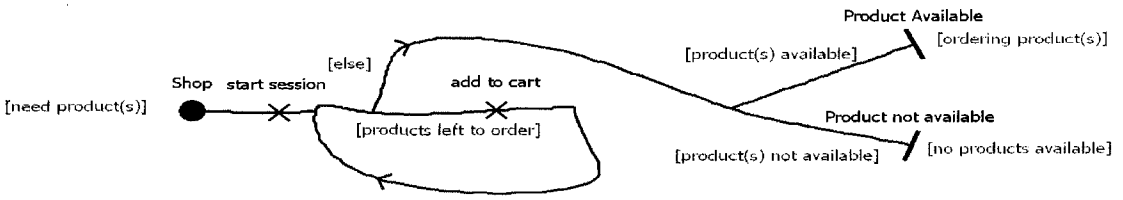


Figure 57 Shop Plug-in

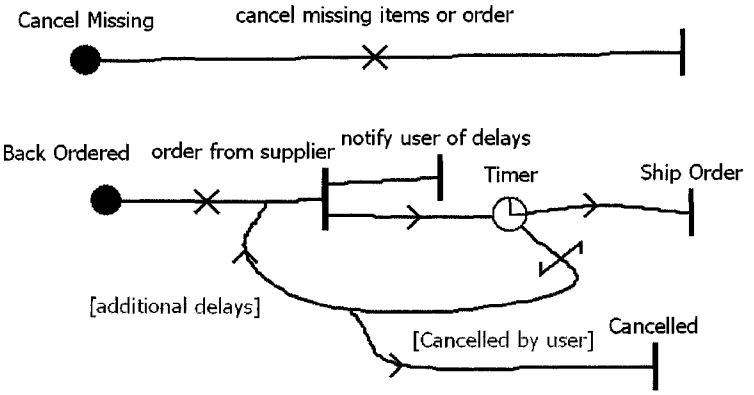


Figure 58 Wait for Supplier Plug-in

Finally, Figure 58 is the plug-in map located inside the *Wait for Supplier* stub. Two disjoint paths are included. The *Cancel Missing* start point is bound to the cancel incoming path, which originates at the *Wait for Order* stub whereas its related end point (unnamed) is not bound to any outgoing path: the path will not return to the top level map after *Cancel Missing Items or Order* is invoked. The *Back Ordered* start point is bound to the path coming from *Reserve Available Items* and the *Cancelled* end point is bound to

the outgoing path that leads to the *Release Items* responsibility. Both the *Notify User of Delays* and *Ship Order* end points are bound to outgoing paths that lead to responsibilities of similar names.

The process which is explained is not overly complex. For example, there is no discussion of payment processing in order to keep the example simple. However, the model is expressed in such a way to cover a large breadth of scenario traversal features, making it an ideal example to demonstrate the scenario traversal algorithm and the MSC export. An example of a superfluous modelling construct is everything concerning the warehouse employee in this model. The employee is not of much interest in the main scenarios and the model could be simplified with responsibilities on the main path instead of what is currently in place. However, one could imagine this model as being in the first steps of evolution from a black-box web order towards a white-box view of the internal business processes in place at the warehouse.

Table 6 Case Study Data Model

Name	Description	Type
IsArrivingSoon	is the product arriving soon?	Boolean
IsCustomerPatient	will the customer wait for the back-ordered products?	Boolean
IsProductAvailable	is the product available in the warehouse?	Boolean
IsWarehouseOutOfStock	is the warehouse out of stock?	Boolean
OrderedProductCount	how many products were ordered?	Integer
OrderToProcess	is there an order to be processed?	Boolean
ProductShipped	has the product been shipped?	Boolean
ProductsToOrder	the number of products to be ordered	Integer
ReceivedMissingProduct	has the warehouse received the missing product from their supplier?	Boolean
UserDecision	what will the user do if products are missing?	Enumeration (CANCEL, CANCELMISSING, WAIT)

Table 6 presents the variables that are used in our model during scenario execution. Table 7 presents the responsibility definitions with their accompanying pseudo-code. The Boolean expressions associated with the conditions in this model are straightforward and will not be presented in this chapter.

Table 7 Responsibility Definitions

Name	Description / Pseudo-code
add to cart	adds the selected item to the cart *ProductsToOrder=ProductsToOrder-1;* *if (IsProductAvailable) {* *OrderedProductCount=OrderedProductCount+1;* *}*
cancel missing items or order	cancel the order or the missing items, depending on the customer's preference *if (UserDecision==CancelMissing)* *IsProductAvailable=true;* *else // if UserDecision is Cancel or wait* *IsProductAvailable = false;*
count	count the received products
gather products	gather the ordered products in the warehouse *IsProductAvailable=!IsWarehouseOutOfStock;*
increment inventory	increment the inventory with the received products *IsWarehouseOutOfStock=!ReceivedMissingProduct;* *IsProductAvailable=ReceivedMissingProduct;*
order from supplier	order backordered products from the supplier
process order	process the order that was received. *OrderToProcess=true;* *ProductShipped=false;*
rejoice	don't worry, be happy
release items	release the items that were reserved for a particular order because it was cancelled
reserve available items	reserve on-hand items for a particular order (which was delayed)
ship order	ship the order to the customer *ProductShipped=true;*
start session	start the shopping session *OrderedProductCount=0;*
submit final order	submit your order and payment details to the web store

This particular example is of great interest for this case study because it covers almost all the modelling constructs available when creating scenarios.

1- Responsibilities with attached pseudo-code
2- Or-forks with conditions and or-joins
3- Concurrency via and-forks; (and-joins are not used in this model).
4- Static stubs.
5- Dynamic stubs. One in particular has no plug-in map.
6- Start points with preconditions, end points with post-conditions
7- A waiting place and a timer unblocked by incoming paths, with timeout path
8- A timer unblocked by conditions, with timeout path
9- Use of enumerations, integer and Boolean variables.
10- Included scenarios
11- Scenario end points that must be reached
12- Scenario preconditions, scenario postconditions.

5.2. Modelled Scenarios

Seven scenarios were created to cover different behaviours modelled in this system. One of these is a simple, re-usable base with default variable initializations and other elements that it is included by the six other scenarios. Further details can be found in Appendix B.

The first and simplest scenario covers the case where the product cannot be found in the store; the user simply leaves and goes elsewhere. A more interesting scenario is the normal order scenario which is defined in Figure 59 and can be seen highlighted in Figure 60, Figure 61, and Figure 62. The scenario definition includes the Base scenario; the greyed-out elements are inherited from this scenario.

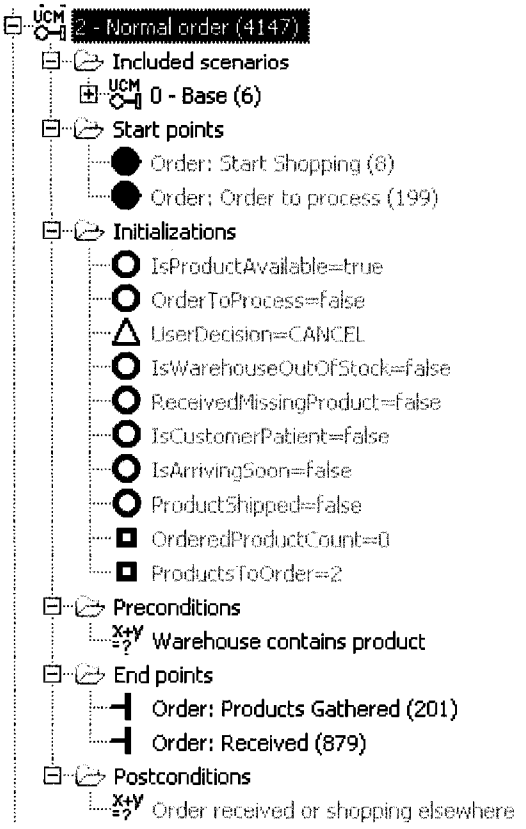


Figure 59 Normal Order Scenario Definition

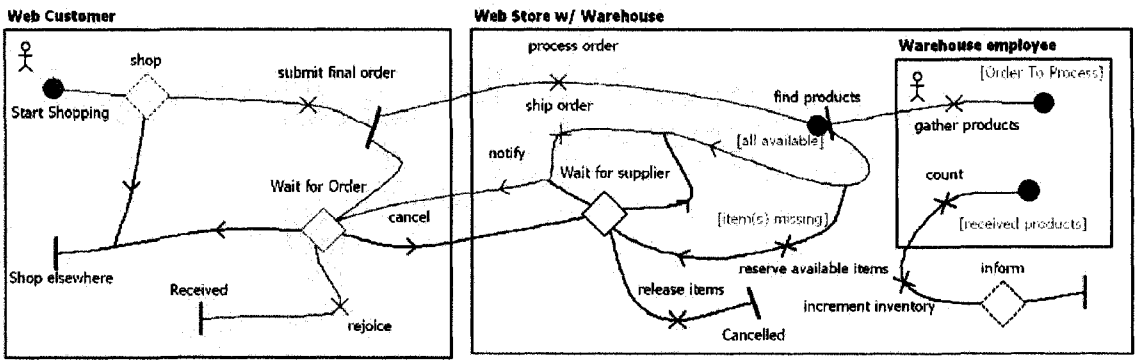


Figure 60 Normal Order Scenario Execution (Root Map)

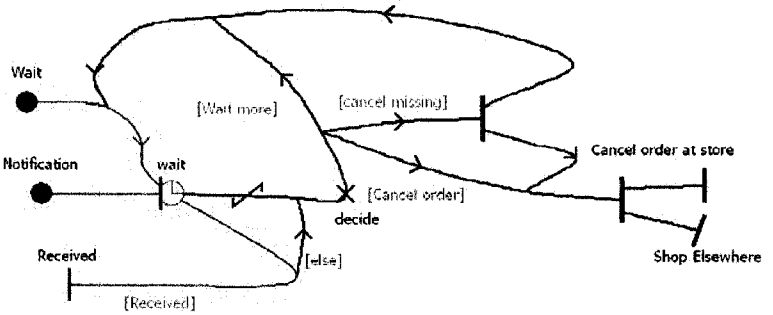


Figure 61 Normal Order Scenario Execution (Wait for Order Plug-in Map)

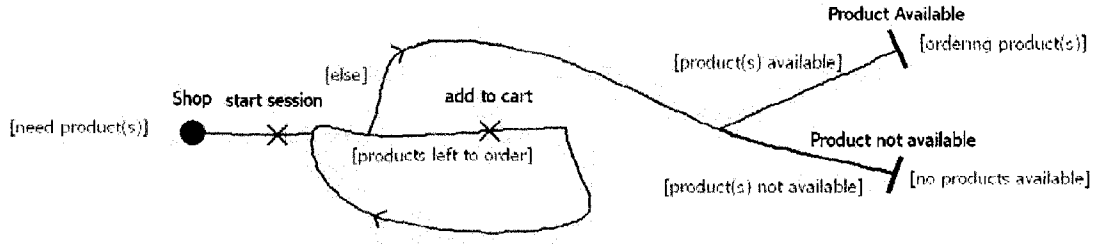


Figure 62 Normal Order Scenario Execution (Shop Plug-in Map)

Once an order is submitted, the path is divided into two concurrent processes. First, the order reaches the warehouse where it is to be processed. In parallel, the customer waits for the order to arrive or a notification that there are supply problems. The first branch blocks at the *find products* waiting place while an employee gathers the products.

An interesting construct here is the *Order To Process* pre-condition on the related start point. The normal order scenario launches two scenario start points when executed: the *Start Shopping* start point and *Order To Process*. The conceptual meaning of this behaviour is that the employee is present even while there are no orders to process. Additionally, the *Order To Process* pre-condition is false at the start of the scenario while the customer is shopping. Hence, because the default scenario traversal algorithm is patient (see 3.4) by default, this start point is blocked at least until the *Process Order* responsibility is executed (where the Boolean flag is set to true). This scenario also works when the algorithm is impatient because of the order of the start points in the scenario definition; had they been inversed, the algorithm would have thrown an error because of an invalid precondition. Because of the depth first nature of the default scenario traversal algorithm, however, the execution has to block on the *Find Products* waiting place before another path is attempted.

Once the products are found, they are shipped and the customer receives a notification inside the *Wait for Order* plug-in map. This unblocks the timer, returns back to the parent map where the customer happily receives his order.

As mentioned previously, our model is built using included scenarios. The six other scenarios all include the same base scenario which defines the start points and variable initializations. Initializations can be overridden, but the all the other elements defined in the base scenario are simply included. Even though the behaviour of the base scenario is the same as the normal scenario, it is not desirable to have the other scenarios

include the normal scenario because this prevents the creation of useful scenario post-conditions and reached end points (because they would have to be true for all scenarios).

There are three similar scenarios that represent the events when the products are backordered: One case with the customer waiting until the products arrive, one case with the customer cancelling the whole order, and one case with the customer cancelling only the backordered products but waiting for the rest. The case where the customer is patient is shown Figure 63, Figure 64, and Figure 65. The plug-in located in the shop stub is the same as in Figure 62.

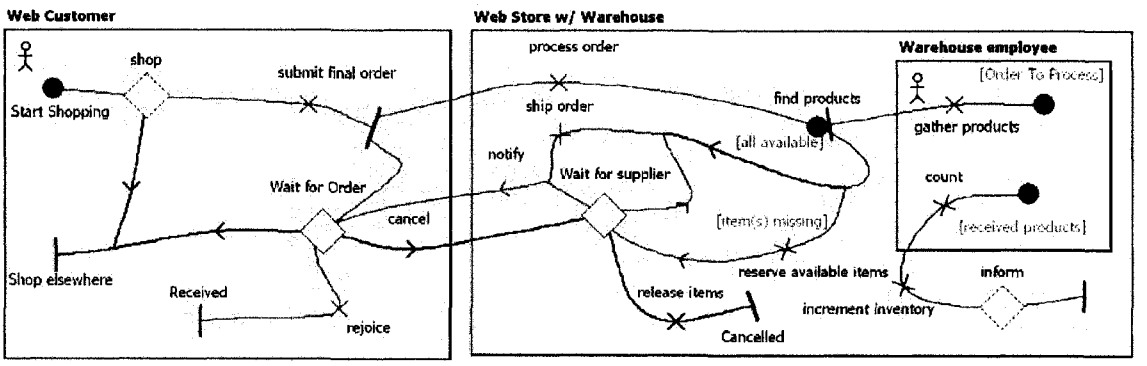


Figure 63 Backordered Products with Patient Customer (Root Map)

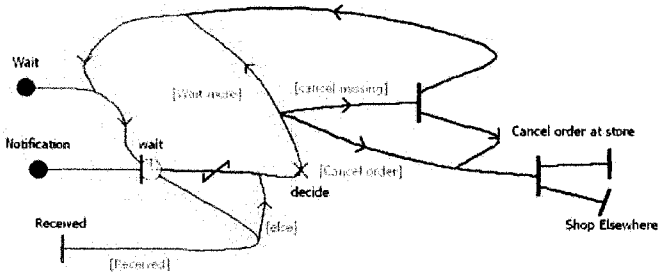


Figure 64 B.O. Products, Wait for Order Plug-in

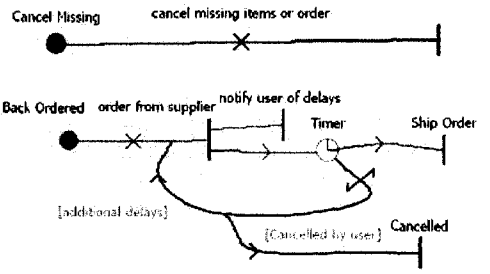


Figure 65 B.O. Products, Wait for Supplier

The interesting construct here is the Timer in the *Wait for Supplier* plug-in which is paused until the order arrives (and the inventory is incremented). It should be noted that the timer is unblocked by a variable becoming true here as opposed to a path arrival. Furthermore, note that even though the *Inform* stub is empty, a default path is assumed to exist because the stub has only one input and one output. The final scenario that was modelled is the case where the supplier never ships the product to the warehouse and the customer is patient. An infinite loop occurs and is detected by the traversal algorithm.

5.3. Multiple Scenario Execution

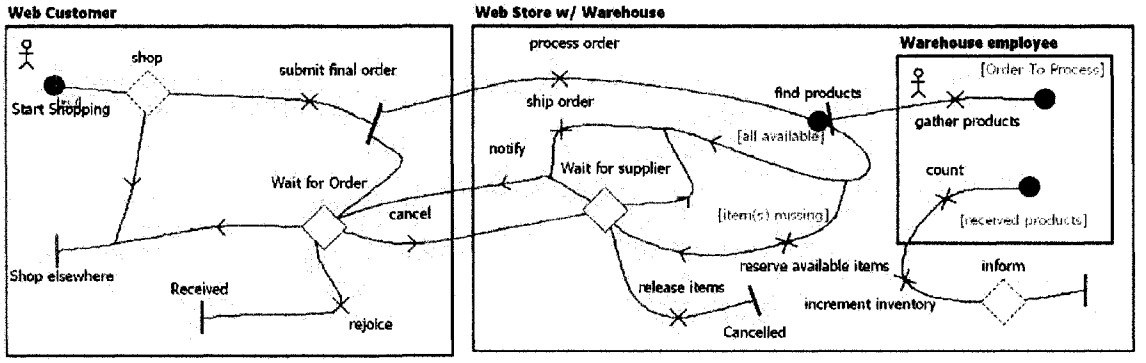


Figure 66 Execution of All Scenarios

Figure 66 highlights the situation where all scenarios are run to verify scenario coverage, as described in 3.3.3. The designer can peruse the list of maps to see that all paths are traversed at least once, and can verify how many times the engine traversed a path by hovering over an element and looking at the hit count. Furthermore, because errors and warnings are cumulated during the traversal of each scenario, the modeller obtains a quick overview of any breaking changes that might have recently been made.

5.4. Flattened Scenario

As discussed in Chapter 4, the first algorithm in the MSC export process is the UCM flattening algorithm which eliminates all alternatives (including loops) and stubs. An example of such a flat scenario, generated from our normal order scenario, can be seen in Figure 67.

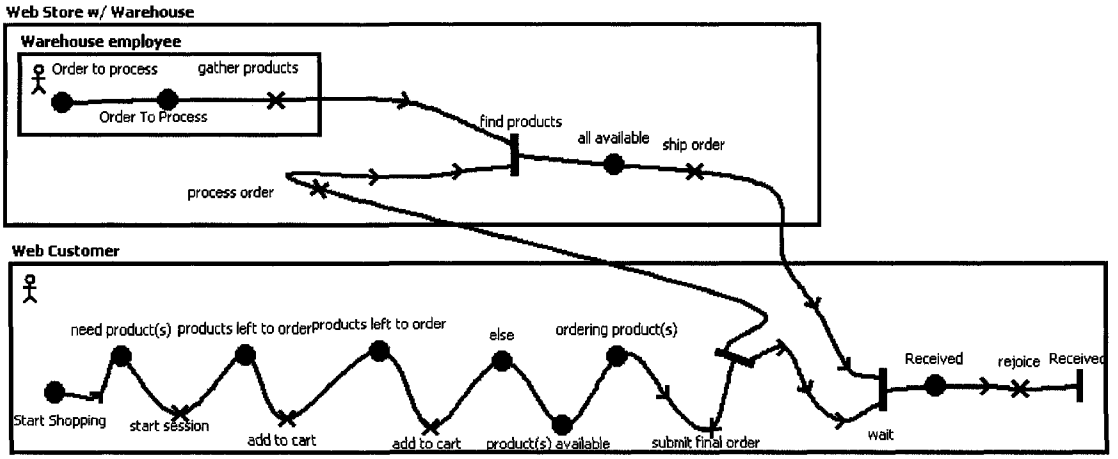


Figure 67 Flattened Normal Order Scenario

5.5. Well-Nested Scenario

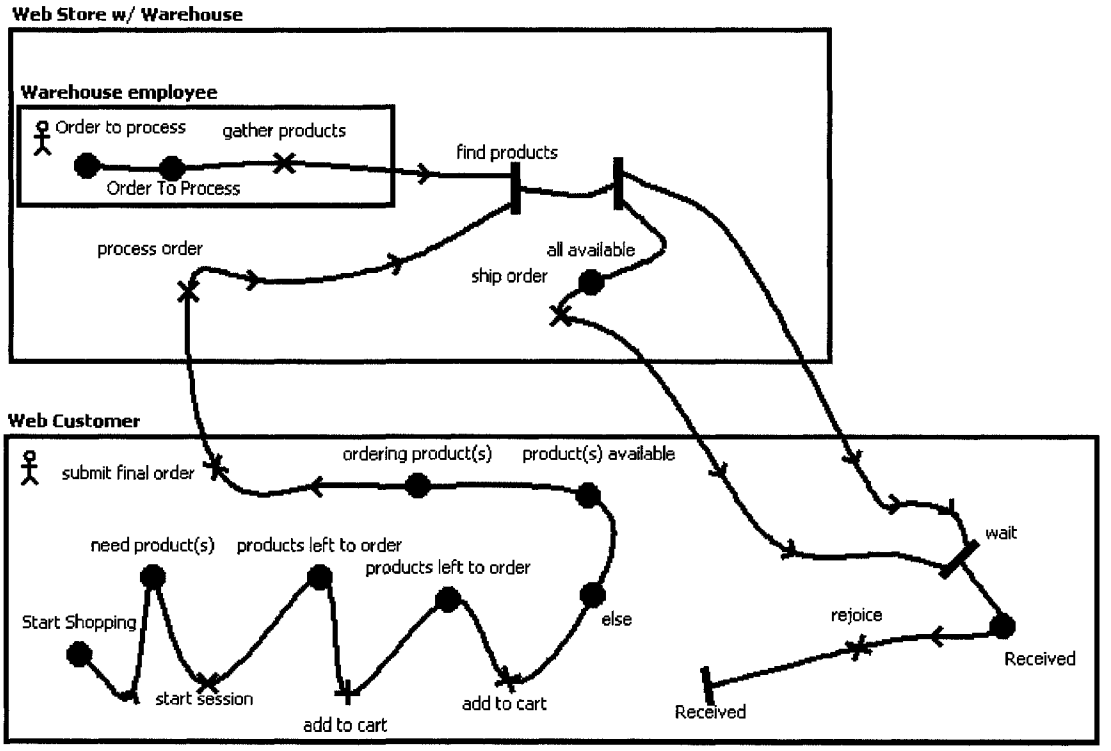


Figure 68 Well-Nested Flattened Normal Order Scenario

The flattened UCM is check for well-nestedness according to the rules described in section 4.3.6 by the second algorithm in the MSC export process. Since it is not well nested, the model is transformed and additional concurrency constraints are imposed to ensure that it can be expressed in a linear form (such as an MSC). A well-nested version can be

seen in Figure 68. Here, the three consecutive direction arrows carrying plug-in traversal events (as metadata) were constrained to be executed after *process order* instead of in parallel.

5.6. Generated Message Sequence Chart

Finally, the scenario generation algorithm builds a UCMScenarios model instance by traversing the generated well-nested URN model shown in Figure 68. This output can then be visualized in the MSC Viewer that is packaged with jUCMNav, as seen in Figure 69. We observe that the start point precondition is evaluated before the start point on the *Web Customer* instance. The lengthy sequence of conditions and action is repeated until the order is submitted to the *Web Store w/ Warehouse*, which processes the order and informs the *Web Customer* that they must wait for a notification. The MSC then shows the *Warehouse Employees* gathering products. Logically, this makes sense in our context, but looking at the intermediate UCMs, it might be seen as though these actions were done in parallel. However, as discussed in 3.5.9, start points are launched in sequence in the order in which they are specified in the scenario definition; therefore the MSC follows the same convention. In the parallel block, we see that the timer is set in parallel to the order's shipment. Looking at this MSC, we see that a race condition could occur in this context: the notification could be received by the customer before setting the *wait* timer. This is indeed problematic, but looking at the source UCM, we see that this is also the case in the original model. This never happens in the UCM because of the way the traversal algorithm supports concurrency, but that is due to the implementation details of the and-fork. Hence the MSC is representative of the source UCM and this visualization reveals a potential issue.

5.7. Chapter Summary

This chapter presented a web ordering UCM example and the intermediate steps leading to the generation of a Message Sequence Chart for a particular scenario. This simple case study covered most of the notational elements and situations encountered in typical UCM

models. Additional information on the other scenarios of this case study can be found in Appendix B.

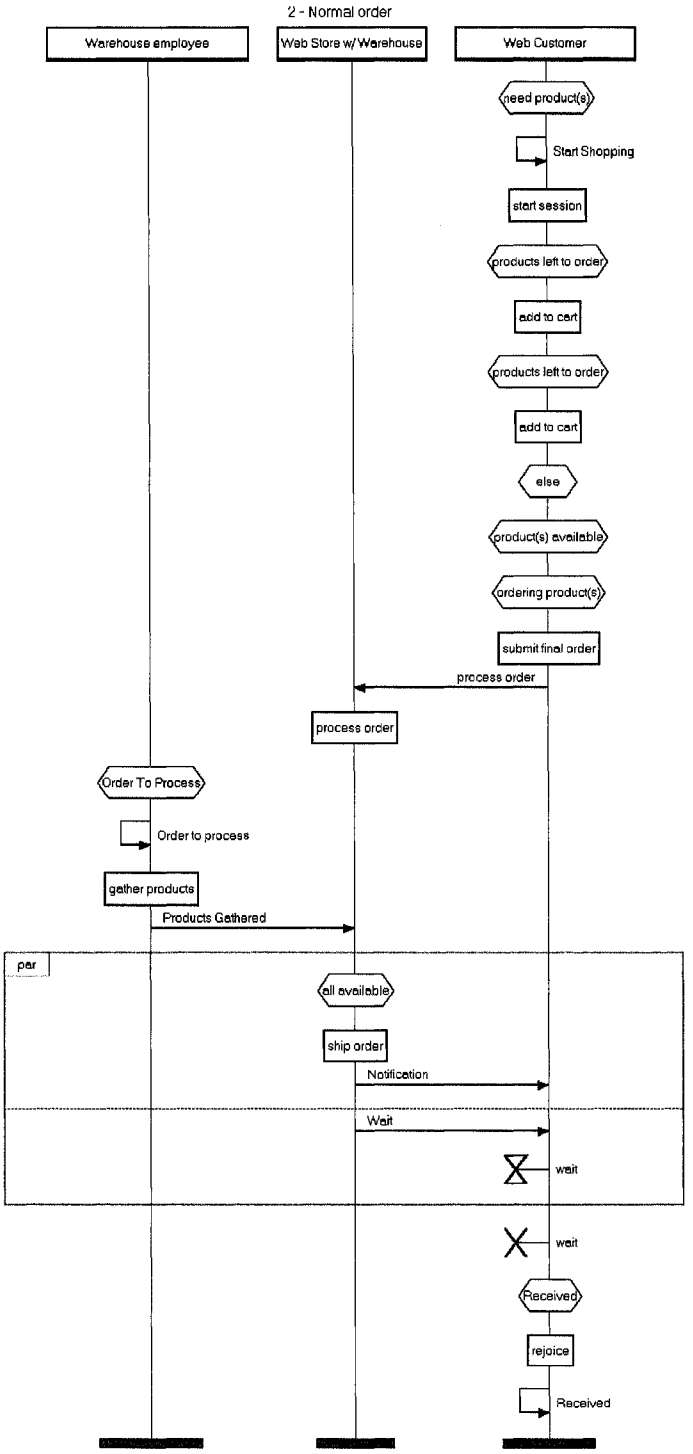


Figure 69 MSC Generated from Normal Order Scenario

Chapter 6. Discussion

The discussion first focuses on additional arguments in favour of the good extensibility of the framework developed by adding a new plug-in to jUCMNav enabling the import of use cases. Then, several semantic variation points offering alternatives to the default UCM semantics and traversal algorithm used so far are presented.

6.1. Framework Extensibility: Import from Textual Use Cases

This section discusses the realization of Task G in section 1.2.

6.1.1 Problem Statement

Both textual use cases and Use Case Maps are used in the early stages of requirements engineering. Use cases are widely used and closely related to UCMs. However, no tool has been developed to facilitate the generation of UCM from use cases. The implementation of a bootstrapping mechanism that generates UCMs from textual use cases will help leverage the work already produced by software engineers. The main idea is to provide the means to try out the notation on non-trivial examples and play with it without having to build everything from scratch.

Given time constraints, and because this thesis is not focused on natural language processing (NLP), it would be impossible to implement anything that would work on free-form textual use cases and import those into jUCMNav. Therefore, a simple prototype was built that creates UCMs from use case models created in a particular tool, namely UCEd [39]. The transformation itself requires a deep understanding of the various UCEd concepts; as this is not relevant in the context of this thesis, the reader is invited to review [26] or the original project report for more detailed explanations. Only a summary of the construct mappings from UCEd use cases to jUCMNav UCM models will be presented.

The goal of this section is to demonstrate the extensibility of jUCMNav's import/export facilities with a complex example. This example is still only a proof-of-concept prototype and is not as robust or well-tested as the algorithms presented in Chapter 3 or Chapter 4. Furthermore, as the scenario traversal algorithm was not implemented when this prototype was created, it does not use the analysis features introduced in this thesis.

6.1.2 Background, Context, and Review of Related Literature

UCEd uses an ad-hoc natural language processing implementation to parse the textual use cases written using the tool and transform them into a more formal representation. UCEd allows software engineers to add more information to use cases than what would conventionally be expected, mainly through its domain model. Because of this, it can generate state models that realize the use cases and perform simulations on these state models [9]. Although the subset of structured English that is allowed in the tool is very limited, it represents a good portion of well-written use cases. UCEd is also implemented in Java and relies on the Eclipse framework (Figure 70).

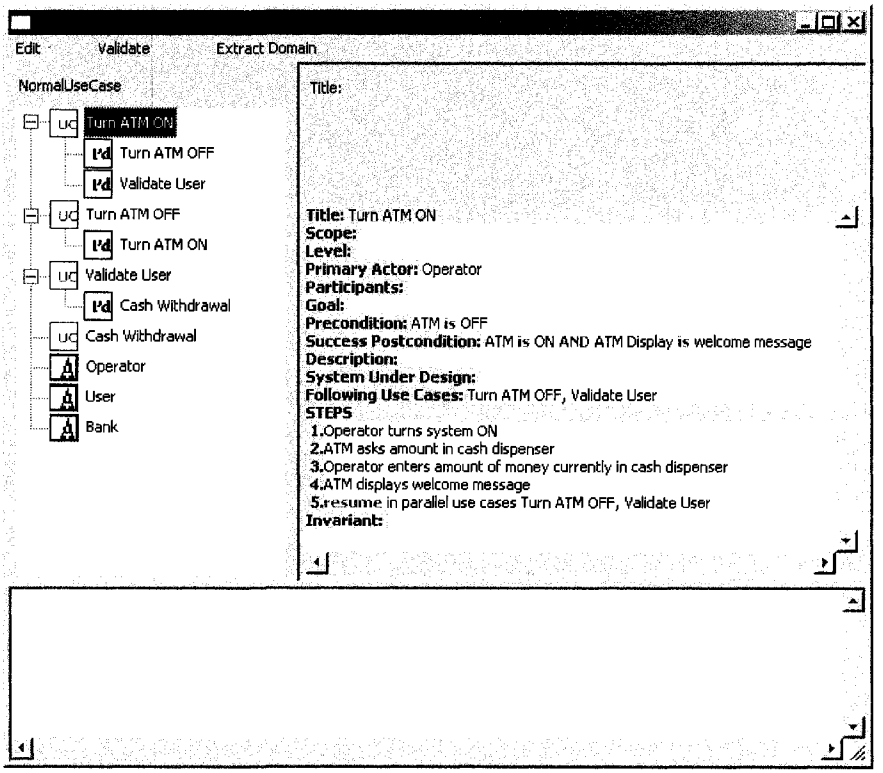


Figure 70 Screenshot of UCEd

UCEd was chosen because it was developed here at the University of Ottawa and because both UCEd and jUCMNav are used in the requirements engineering course that is offered in the third year of the software engineering program. Furthermore, open source alternatives were hard to come by, maybe because textual use cases are so unstructured that most people type them directly in a word processing software.

When this prototype was implemented, jUCMNav did not support scenarios (see Chapter 3) nor did it have a data model. This prototype helped push the introduction of integers and enumerations into the data model, an enhancement upon the previous Boolean-only model used by UCMNav. Therefore, the prototype does not create valid conditions; they are simply textual equivalents of the more abstract representation.

When talking about use cases, the reference on the subject is unequivocally *Writing Effective Use Cases* [11] by Alistair Cockburn. Albeit no format is actually imposed for writing use cases, people tend to adhere to a certain style. Along with different use case writing templates, Alistair Cockburn presents guidelines that should be followed to maximize the benefit of use cases and answers many questions that come to mind when first introduced to use cases. UCEd's use case template is inspired from these guidelines.

The translation of textual use cases into pro-cases, a formal (textual) notation, is described in [30]. The paper distinguishes between different use case steps using NLP. The article is a good introduction to the NLP involved at the lower level, but since his pro-case notation does not support inclusions, other sources were needed in order to transform a set of interrelated use cases into a set of UCMs. Furthermore, he does not try to parse conditions (which are used in alternatives, for example). For these concepts, [40] describes the conversion of textual use cases (from UCed) to finite state machines. The key to being able to automate the conversion is to use simple English, namely the Subject-Verb-Direct Object-Preposition-Indirect object (SVDPI) pattern when writing use case steps [11].

UCEd has a wide feature set; only a few of these features were taken into consideration for the implementation of the prototype.

Actors: The domain model defines the different entities that can be used in use cases. UCEd calls these *Concepts*. There are *System Concepts* and regular concepts. The

former represents the system and the latter, actors. UCEd refines these into sub-concepts, sub-components and instances.

Primary successful scenario: This is a sequence of interactions (sequence of use case steps) between a user and the system that usually occurs.

Alternatives: After a certain step in the use case, if a condition is evaluated to be true, the scenario branches off to this alternative.

Redirection actions: A certain step can redirect to another (*Goto step 3*); this is most commonly used in alternatives to return to the main scenario. Note that UCEd does not allow redirections inside alternatives, which is a good thing to keep the use cases readable.

Use case inclusion: One use case can include another. The UCEd project file contains multiple use cases and defines their relationships.

Use case extensions: One can define an extension point in a use case. Optionally, the system analyst can define extensions that will add behaviour to the use case containing the extension point.

Various conditions: Primary successful scenarios have pre and post conditions, alternatives have conditions, use case extensions can also be conditional and a use case step can be of the type *IF User Password is Invalid THEN System displays error message.*

Time constraints: UCEd allows conditions to contain time constraints such as BEFORE 60s User accepts the terms and conditions.

Any * alternatives: Alternatives can be defined as available after a specific step or after any step.

Operation effects: Operation effects are changes in the use case state when certain operations are performed. Operation are defined (in the domain model) on different Concepts. (Added/withdrawn conditions).

6.1.3 Methodology

The main theoretical challenge to support the automated conversion of textual use cases to graphical UCMs is the low-level natural language processing required to interpret sentences as use case steps. In addition, without a predefined use case template (using free-

form Microsoft Word documents), automation would be practically impossible. Luckily, use cases are well-structured in UCEd and the NLP is handled by the tool. Hence, the work done by the jUCMNav's import plug-in can be summed up as a transformation from one graph structure (UCEd internal use case model) to another (UCMs). In addition to the use case model, we also use UCEd's domain model which is semi-automatically generated from the use case model.

A technical difficulty that was avoided was re-implementing a mechanism which reloads the UCEd object-oriented model from the project file. The UCEd JAR file was simply added to the build path and the prototype uses the tool's input layer. Another technical difficulty is the visualisation of the generated UCM, in which the elements do not have x and y coordinates. Once generated, an automatic layout is performed on all UCMs using the auto layout mechanism already present in jUCMNav. Therefore, the plug-in focuses on the central problem: transforming use cases into Use Case Maps. Creating a syntactically valid UCM model is itself a technical challenge. Because jUCMNav was designed to be extensible, the set of commands (command design pattern) that is used by the editor can be re-used by the import mechanism.

Two UCEd modules were used to implement the transformation algorithm: the use case module and the domain module. UCEd semi-automatically extracts domain information from the use cases. The import mechanism imposes that both the use case and domain model must have passed UCEd's validation algorithms.

The implementation of a jUCMNav import plug-in is very straightforward as it follows the extension mechanism inherent to Eclipse. The extension plug-in must register itself as an implementation of jUCMNav's import extension point (Figure **71**). A few settings are defined in the plug-in's *plugin.xml* file and the *IURNImport* interface must be implemented. The plug-in is given a file and is responsible for converting it into a URN-spec model instance.

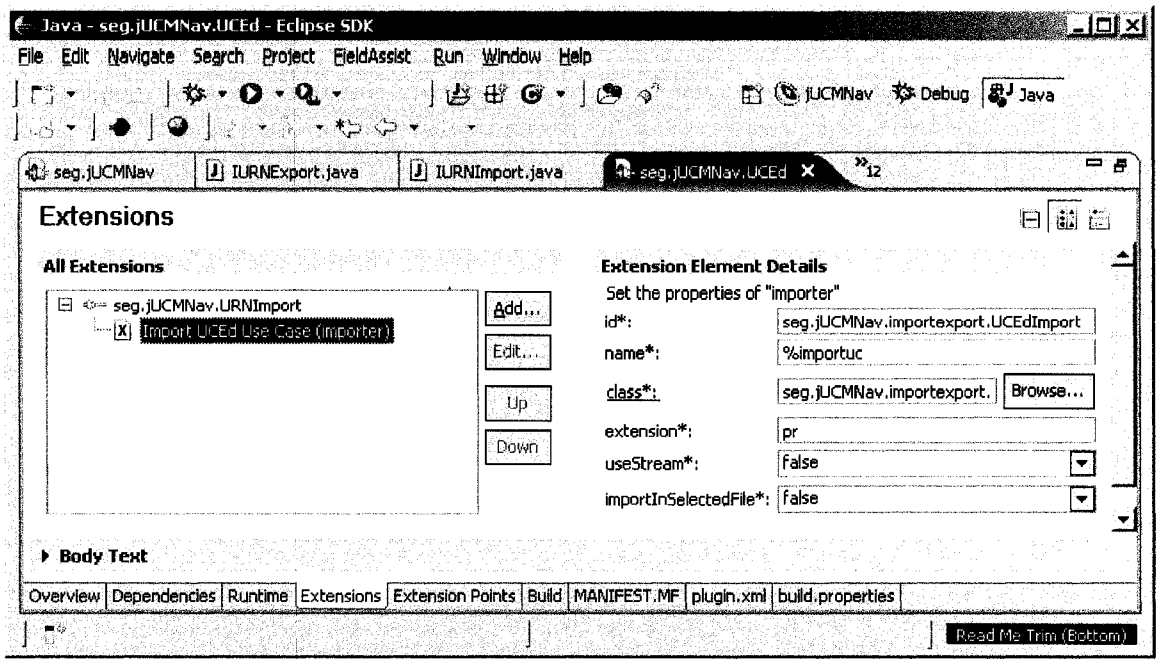


Figure 71 UCEd Import Plugin.xml

6.1.4 Transformations by Example

Actors and Primary Successful Scenario

The simple use case shown in Figure 72 contains all the different actors that are supported by UCEd. The result of the transformation is shown in Figure 73: the structure of the various entities is much clearer in the visual notation. The general transformation rule is that every entity in the UCEd domain tree is mapped to a UCM component definition. Furthermore, every instance of a subject in the SVDPI pattern (Subject-Verb-Direct Object-Preposition-Indirect Object) of a use case step is mapped to a UCM component reference. Even though the UCEd domain model could associate operation effects to certain actions, the prototype focused on the visual aspects of the transformation and does not generate equivalent pseudo-code.

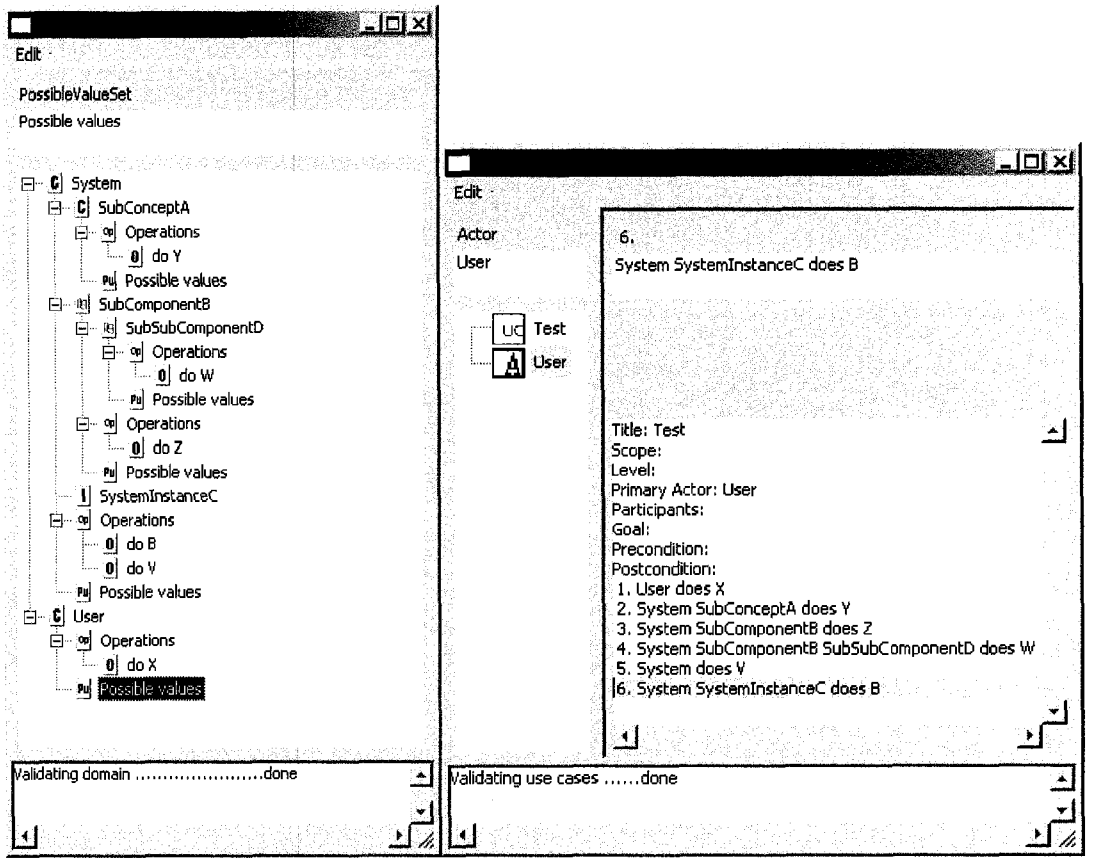


Figure 72 Source Domain Model and Use Case (in UCEd)

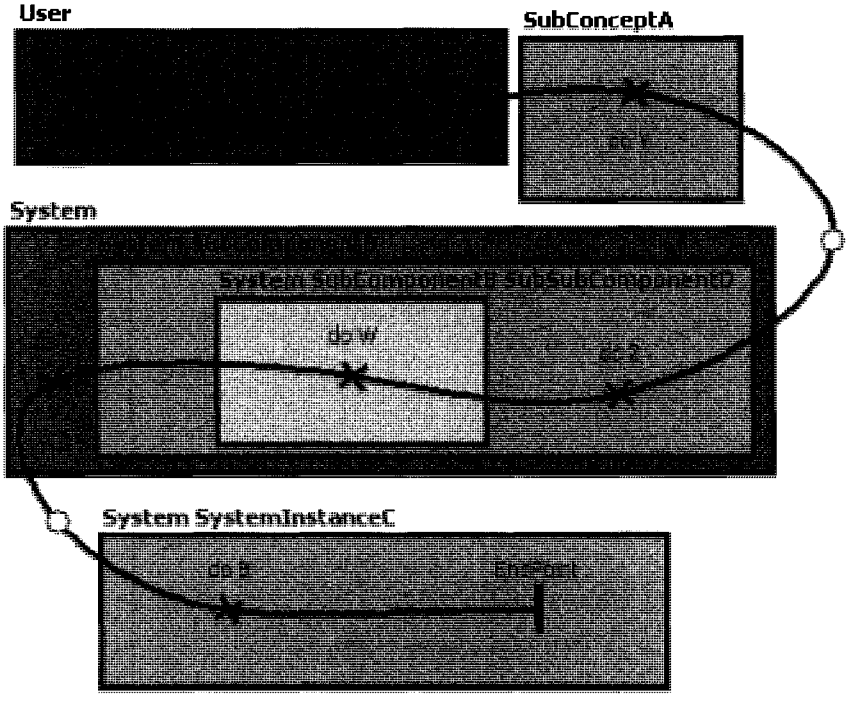


Figure 73 Generated UCM

Alternatives

```
Title: General Scenario
Precondition: jUCMNav is not UCM loaded
Postcondition: jUCMNav is UCM loaded
1. User Jason clicks import button
2. jUCMNav GUI displays file list
3. User Jason selects file
4. UCEd loads Use Case
5. jUCMNav ImportExport UC2UCM converts UC into UCM
6. jUCMNav loads UCM
7. jUCMNav GUI displays UCM
3. a. User file is not found
3. a. 1. Goto step 2
4. a. UCEd Use Case is not valid
4. a. 1. jUCMNav GUI displays error message
```

Figure 74 Use Case Illustrating Alternatives

Use case alternatives (3.a. and 4.a) in Figure 74 are mapped to or-forks in UCMs (see Figure 75) with the alternative condition being mapped to the or-fork outgoing path condition. The use of pre/post conditions are also seen in this example; these conditions are associated with their respective start and end points. The transformation module also converts inline conditions (IF ... THEN ...) to or-forks, as can be seen in Figure 76. Finally, the GOTO construct is used at step 3.a.1 of Figure 74. This generates an and-join in the target UCM. Although jUCMNav does not allow the designer to create explicit infinite loops (loops with no outgoing path), these can be created in UCEd (GOTO the previous line). Since the import mechanism uses the commands built for use in the editor, infinite loops are automatically prevented. It is therefore clear that an efficient use of jUCMNav import/export framework's increases robustness, ensures syntactic legality, and reduces model development time.

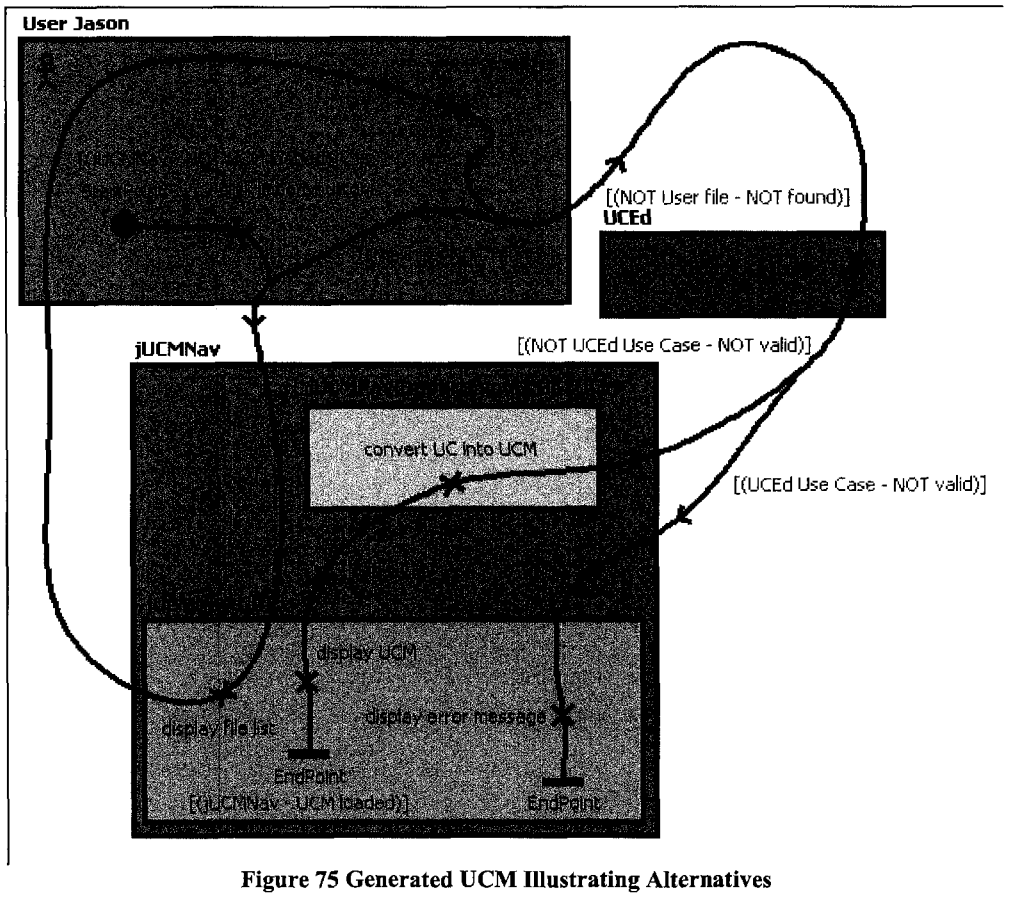


Figure 75 Generated UCM Illustrating Alternatives

Use Case Inclusion and Extension

```
Title: General Scenario
1. User Jason clicks import button
2. User Jason selects file
3. include Load Use Case
4. jUCMNav ImportExport UC2UCM converts UC into UCM
5. jUCMNav loads UCM
ExtensionPoint==> loaded
6. jUCMNav GUI displays UCM

Title: Load Use Case
Precondition: UCEd ProjectModel is NOT loaded
Postcondition: UCEd ProjectModel is loaded AND UCEd ProjectModel
is valid
1. UCEd loads Use Case
2. IF UCEd Use Case is not valid THEN jUCMNav GUI displays error
message

Title: Logging
PART 1. At Extension Point loaded
1. jUCMNav logs usage
```

Figure 76 Use Case Illustrating Inclusion and Extension

Figure 76 presents a more complex set of textual use cases which make use of both inclusion (via the *include* verb) and extensions (via extension points). The mapping for inclusion is straightforward as this is conceptually equivalent to a static stub. However, although extension points are modelled using a dynamic stub, their semantic meaning differs. Indeed, in use cases, extension points model locations where behaviour can be added. There can be a number of extension use cases which all do behaviour in some undetermined order. In the UCM notation, on the other hand, the dynamic stub only supports the execution of one plug-in amongst a set, depending on the plug-in selection condition. The prototype simply binds the extension use cases to the dynamic stub without attempting to resolve this problem. This semantic variation was also encountered by the scenario traversal algorithm; see section 6.2.10 for more information. Furthermore, because extension points do not require extension use cases to be present, the dynamic stub could be left empty. This mapping motivated the adoption of the default behaviour implemented for empty stubs (see section 3.5.7).

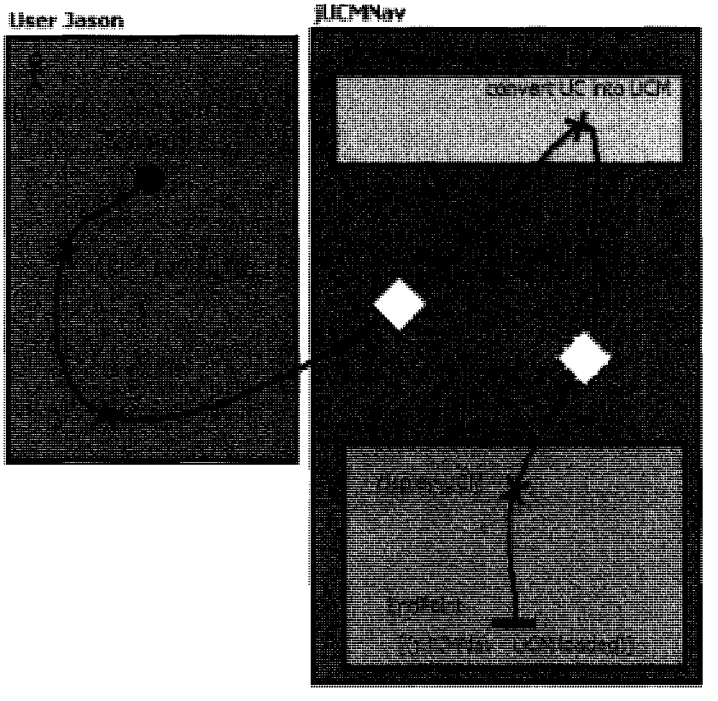


Figure 77 Generated Top-Level UCM Illustrating Inclusion and Extension

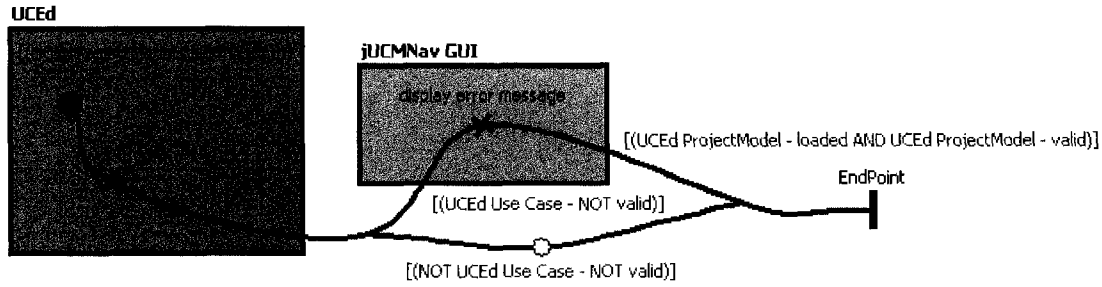


Figure 78 Generated Load Use Case Plug-In Map

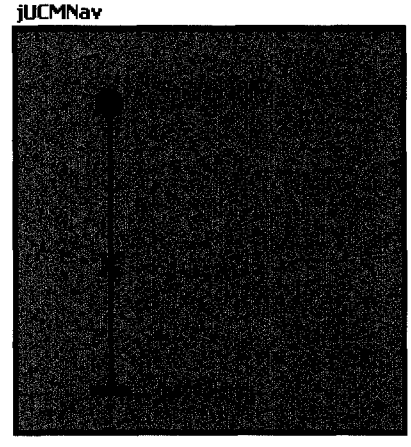


Figure 79 Generated Loaded Plug-In Map

Time-Constraints and Concurrency

Timing conditions are one of the features that presented the greatest technical challenge. The prototype contains a simple solution that works with a very limited number of cases, but would not scale to the complexity of UCEd's features. In UCEd, use case steps can define that a certain step will occur BEFORE or AFTER a certain interval of time. Furthermore, alternative conditions can also use timing constraints, combining BEFORE, AFTER and regular conditions as defined previously.

Use Case Maps include a timer construct which is a special waiting place where the scenario is paused until another path activates the timer. The timer also specifies a timeout time after which it stops waiting and the original scenario goes down the timeout path. However, jUCMNav's meta-model currently does not support any way to define that actual timeout duration, except perhaps with metadata. And even if it did, representing BEFORE and AFTER implies creating secondary paths that activate the timer as de-

scribed by the corresponding conditions in UCEd. This is far from being visually intuitive for those who are not familiar with UCMs.

```
Title: General Scenario
Primary Actor: User Jason
1. User Jason clicks import button
2. User Jason selects file
3. BEFORE 60 sec jUCMNav ImportExport UC2UCM converts UC into UCM
4. jUCMNav loads UCM
5. jUCMNav GUI displays UCM
2. a. AFTER 60 sec
2. a. 1. jUCMNav GUI displays error message
```

Figure 80 Use Case Illustrating Time Constraints

For this reason, this feature was simplified to a lone timer, with an expressive name such as shown in Figure 81 (WAIT AT MOST 60 seconds). Figure 80's use case step 3 does not declare what happens if the time constraints are not respected; the alternative adds this behaviour.

However, note that the alternative defines a timeout path condition that it is the logical opposite of the timer's timeout duration. This may not always be the case (an alternative could be taken if "BEFORE 60 s and AFTER 10s AND User Jason is convicted felon"), and because logical expression manipulation was out of scope, an assumption was made: if an alternative exists on a use case step that uses a timer, the alternative will use the timeout path. If many of these alternatives exist, the timeout path will be forked.

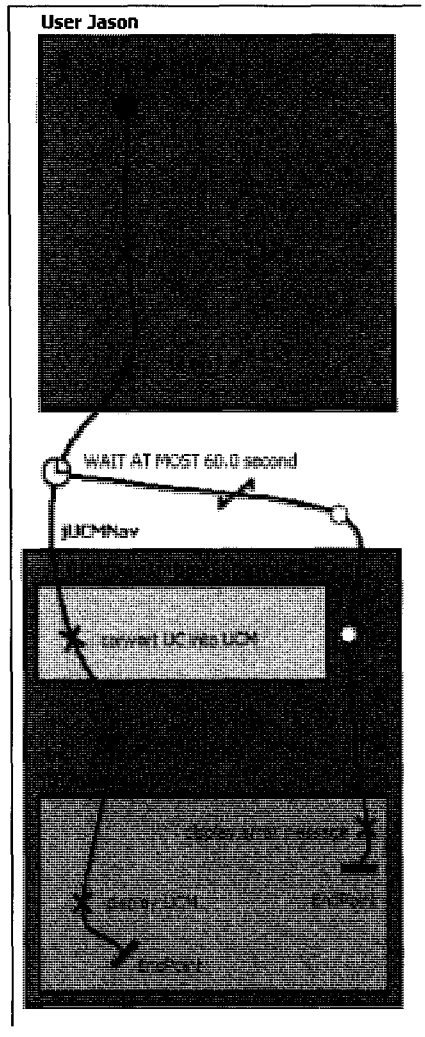


Figure 81 Generated UCM Illustrating Time Constraints

In a static UCM model, there exists too great a gap between UCEd's time-based elements (used in state machine generation) and the UCM timer for a perfect transformation to be defined. Individually, the concepts make sense in their context, but the notion of time in UCMs is probably required to allow for a breakthrough in this direction (for instance, time extensions for UCMs are proposed in [18]). Because of this realization, the scenario traversal algorithm was built to be able to support two distinct expressions on the node connections which leave the timer (see 3.5.6). Although this was not available at the time of the prototype's implementation, one could now use the conditions expressed in the use case model as conditions on both the continuation and the timeout paths.

Any * Alternatives

```
Title: New
1. System displays welcome screen
2. User Jason clicks start
3. System displays information
* 1. System is quit requested
* 1. a. System displays welcome screen
```

Figure 82 Use Case Illustrating Any * Alternatives

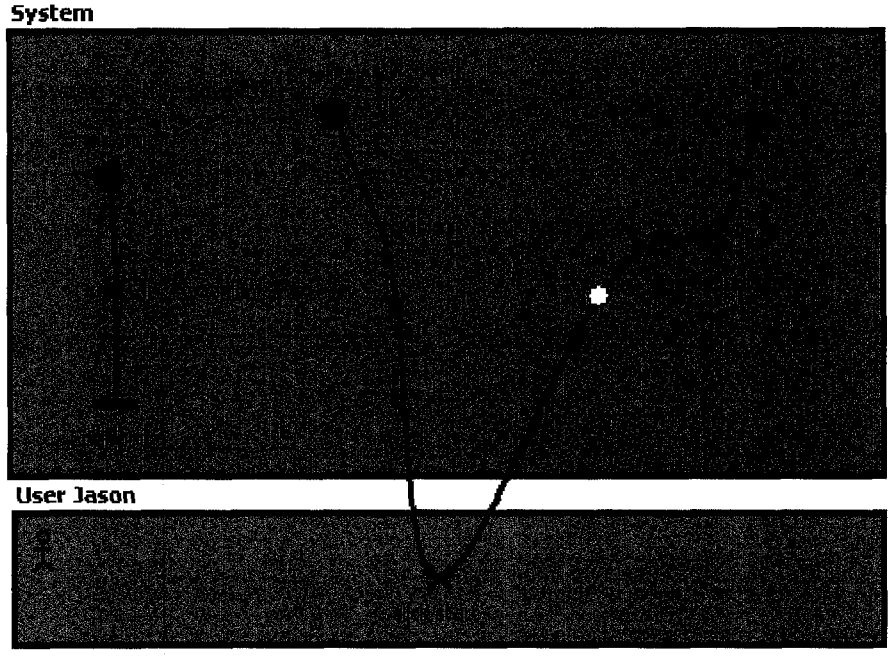


Figure 83 UCM Illustrating Partial Support for Any * Alternative

UCEd allows the creation of *Any * Alternatives*: alternatives that can occur at any point during the execution of the primary scenario. The UCM notation does not yet have such a construct but there have been discussions concerning the concept of a UCM exception. The transformation algorithm still partially supports this concept by creating a second path on the same UCM (see Figure 83) with a precondition to the start point. Although this would not interrupt the second path's execution, it is a step in the right direction.

6.1.5 Discussion and Conclusion

This section's goal was to demonstrate the extensibility of jUCMNav's import/export facilities with a complex example. The implementation was done after the release of jUCMNav v1.0 in parallel to the work being done to support the full URN notation in

jUCMNav (which was introduced in v2.0) as a course project in CSI5180: Natural Language Processing. Prior to this work, the only kind of transformation that was supported was exporting images or layout information. Now, an extensible transformation framework is available and new import/export plug-ins have been written by various contributors:

- Import UCEd Use Case
- Export Linear Scenarios (See Chapter 4)
- Export Message Sequence Charts (See Chapter 4)
- Export to Telelogic Doors (See [27])
- Export Core Scenario Model
- Export HTML Report
- Import/Export GRL Catalog
- Export GRL Strategies

At first glance, the textual use case import plug-in for jUCMNav may not have appeared to be related to scenario execution (Chapter 3) which is the core subject of this thesis. However, as can be seen in the examples presented in section 6.1.4, the creation of this prototype improved the quality of the scenario traversal algorithm in addition to its main goal of perfecting the import/export framework available in jUCMNav.

A few key observations were made which directly impacted its implementation:

- Support for integers and enumerations (matches UCEd domain model)
- Support for the patient traversal algorithm preference (see 3.4) (for any * alternatives)
- Support for complex timers and waiting place continuation/timeout expressions is desirable, as opposed to waiting for a triggering path only. (see 3.5.6)
- The traversal algorithm does not stop at empty stubs (see 3.5.7)

Furthermore, a few more observations on features that could be added to jUCMNav:

- Support for multiple activated stub plug-ins.
- Support for hierarchical component definitions
- Support for associating responsibility definitions to component definitions, creating a component "interface".
- Support for component inheritance.

Finally, thanks to jUCMNav's import/export framework, it is much easier to create model transformations that it ever was with UCMNav. The framework allows developers to extend jUCMNav for their own needs and have full access to the thousands of lines of code already written against the EMF meta-model. As can be seen with the UCEd import plug-in, the infrastructure is limited to a minimum, allowing a greater investment in the transformation itself. The export mechanism is very similar to the import one presented in this section. jUCMNav can also be extended by contributing menu items which also have full access to jUCMNav's infrastructure.

6.2. UCM Semantic Variations

Use Case Maps have a generally easy-to-understand structure but there are some aspects that are not defined precisely, especially with regards to how the scenario traversal mechanism should behave. Section 3.3.3 discussed how we can create different algorithms but the default algorithm has to pick one semantic meaning. This section records the different semantic variations and which meaning was implemented in the default algorithm. Building a list of such semantic variations was identified as Task E in section 1.2 and contributes positively to the achievement of offering better tool support for UCM scenarios.

6.2.1 Semantic Variation #1: Start Point Preconditions

Traversing a start point simply implies going to the single subsequent node on the path. However, all is not as simple as it appears because start points can have preconditions.

Semantic variation: What should be done when a start point precondition is false?

a) Abort path traversal and notify of error
b) Pause until it becomes true, notify of error if this never happens
c) Ignore and log warning

The default implementation does b) but there is a preference (see 3.4) that allows for a).

6.2.2 Semantic Variation #2: Multiple Or-Fork Branches Are True

Semantic variation: What should be done when multiple or-fork branches evaluate to true?

a) Pick the first one and give warning (what defines the first?)
b) Pick one branch randomly and give warning
c) Launch all true branches in sequence (what sequence?)
d) Launch all true branches in parallel
e) Abort and notify of error (non-determinism detected)
f) Keep the alternatives in the end-result (e.g., MSCs have the ALT construct)
g) Pause until only one becomes true, notify of error if this never happens

The default implementation in jUCMNav does either a) or b), depending on the deterministic user-defined preference (see 3.4). It was decided that the or-fork should keep its exclusive-or meaning, but this is definitely question of opinion strongly influenced by the legacy implementation available in UCMNav (which only supports e).

6.2.3 Semantic Variation #3: No Or-Fork Branch Is True

Semantic variation: What should be done when no or-fork branch evaluates to true?

a) Abort and notify of error (deadlock)
b) Pause until one becomes true, notify of error if this never happens
c) Pick one branch randomly and give warning

In the original UCMNav option a) was taken, but because the user-defined preference of having a patient algorithm was already implemented for preconditions (see 3.4 and 3.5.1), the default implementation behaves as b). If the algorithm is configured as impatient, it will then behave as UCMNav does.

6.2.4 Semantic Variation #4: Not All Paths Arrive at And-Join

Semantic Variation: If some of the incoming branches never arrive, what happens?

a) Block and throw an error
b) Continue and warn

Because and-joins are used for synchronization in Use Case Maps and all branches must be synchronized; the default implementation uses a). It should be noted that certain workflow notations allow for more complex rules such as n-out-of-m arrivals but this is currently not the case with Use Case Maps (see [33]).

6.2.5 Semantic Variation #5: And-Join Memory

If we consider the path arrivals as tokens on incoming branches, does the and-join remember what arrived and at what moment? Of course, it has to wait minimally until a token arrives at each incoming branch before continuing, but what happens when a token arrives on the same branch multiple times before the and-join continues?

Semantic Variation: Do and-joins have memory?

a) Yes. If multiple tokens arrive on the same branch, they are queued.
b) Yes. If multiple tokens arrive on the same branch, the arrival count is remembered.
c) No. Multiple arrivals are ignored.

The default implementation uses b) as only arrival counts are remembered. If the tokens do not contain any data, a) and b) are equivalent, but this is not the case in the default scenario traversal. The token (a *TraversalVisit*) does contain context information such as the traversed PluginBindings. A simple and effective test was implemented to verify if all branches have arrived by comparing the *TraversalResult* for the incoming branches with the outgoing branch. The path can only continue if the incoming branch with the smallest number of traversals is one more than the number of times the outgoing node connection was traversed.

```
if (minimum arrival count on arrival branches == outgoing branch traversal count +1)
        unblock();
```

6.2.6 Semantic Variation #6: Simultaneously Blocked Path Node Instances

Semantic Variation: Can there be multiple instances of *TraversalVisit* representing the same path node in the waiting list simultaneously?

a) Yes. Multiple arrivals on the same branch launch a second instance blocked at the same place. Instances are queued for token reception (FIFO).
b) Yes. Same as above but LIFO.
c) Yes. Same as above but random token reception.
d) No. If the and-join has no memory, additional arrivals are ignored.
e) No. Keep track arrival counts (as in 6.2.5) and throw errors if they do not match up at the end of the execution.
f) No. Keep track arrival counts (as in 6.2.5) but do not do anything if they do not match up at the end of the execution.
g) No. Keep track of arrivals counts (as in 6.2.5) but only throw error if the last arrival did not unblock the and-join.

The default implementation uses g) because at any point in time, all of the *TraversalVisits* in the waiting list represent distinct path nodes. A path node is blocked only if it is not already in the waiting list. To clarify the default implementation, here is a simple example featuring an and-join with two incoming branches (see Figure 84).

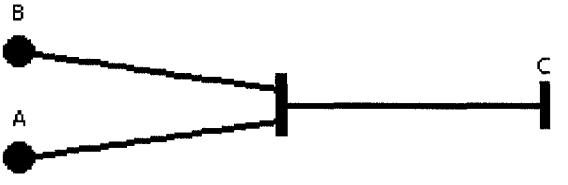


Figure 84 Simple And-Join

- Case 1) If each incoming branch is hit once, the outgoing branch is taken and no errors are thrown.
  - Arrivals: A, B.
  - A=1 and B=1 implies C=1
- Case 2) If any incoming branch is never taken, the outgoing branch is not taken and an error is thrown. (if traversal ends and nothing else arrived)

- Arrivals: B, B, B....
- A=0 and B=N implies C=0 and error because still blocked

- Case 3) If the first incoming branch is hit twice, followed by one hit on the second incoming branch, the outgoing branch is taken once and no errors are thrown.
  - Arrivals: A, A, B
  - A=2 and B=1 implies C=1 but no error because the last token unblocked it
- Case 4) If the first incoming branch is hit once, followed by two hits on the second incoming branch, the outgoing branch is taken once and an error is thrown.
  - Arrivals: A, B, B
  - A=1 and B=2 implies C=1 and error because the last token did not unblock it.

However, even though it was simple to develop, the current implementation does have its flaws. When a token arrives at a blocked and-join, the blocked and-join's context is reviewed. If the new token comes from a different parent stub, it will have a different set of plug-in bindings in its *TraversalVisit*. The two sets are merged, to subsequently return to the parent stubs at the end of the path (see 3.5.8). Because we do not allow multiple blocked *TraversalVisit* instances to refer to the same path node, this is problematic in Case 3 above.

Imagine Figure 84 where A is bound to two different parent maps (A1 and A2) and B is bound to two other parent maps (B1 and B2). If we have the following arrivals: A1, A2, B1, B2. C is first reached after the arrival from B1. Since only gather counts and accumulate plug-in binding contexts (instead of keeping FIFO queues), the context when the token reaches C1 references maps A1, A2 and B1. Then, when B2 arrives, because of our simple algorithm, it can continue but when it reaches C, the context is only B2. With FIFO queues, one would get contexts A1, B1 (first two tokens to arrive on the two incoming branches) then A2, B2. This can lead to unintended behaviour. This issue is left unresolved and might be addressed by future enhancements to the default algorithm.

6.2.7 Semantic Variation #7: Both Timeout and Continuation Paths Active

Semantic variation: What should be done when both the timeout path and continuation path of a timer evaluate to true?

a) Pick the continuation condition
b) Pick the timeout condition
c) Pick one branch randomly and give warning
d) Launch all branches in sequence (what sequence?)
e) Launch all branches in parallel
f) Abort and notify of error

This is similar to a semantic variation presented in 6.2.2 with the additional constraint that it makes no sense for a timer to continue and timeout simultaneously. Therefore, the default implementation gives priority to the continuation path (option a)). If both conditions are false, the waiting place or timer is placed in the waiting list and treated similarly to an and-join. It waits for the arrival of the triggering path, the path that is connected to it, if one is available. The arrival of this connected path will launch the continuation path. If it never arrives, as mentioned in 3.5.6, the timeout path will be taken or an error will be thrown if none exists. It is to be noted that every time that a waiting place or timer is attempted, the conditions are evaluated. This means that a waiting place or timer can be blocked for a period of time and released because the *UcmEnvironment* has changed state.

6.2.8 Semantic Variation #8: Waiting Place and Timer Memory

If a token arrives on the incoming branch while the waiting place or timer is already blocked, what happens?

Semantic Variation: Do waiting places and timers have memory?

a) Yes. Tokens are sent to queued instances (FIFO).
b) Yes. Same as above, but LIFO.
c) Yes. Same as above, but random blocked instance.
d) No. Subsequent arrivals are ignored.

Section 6.2.5 clarified that and-joins have memory in relation to the incoming branches. This is not the case for waiting places and timers. Although this inconsistency

might appear odd, the commonality between both situations is that the same path node cannot be added simultaneously to the waiting list (see 6.2.6). Improvements are possible in this direction. Hence, the default scenario traversal algorithm does d).

6.2.9 Semantic Variation #9: Continuation of Unblocked Waiting Place or Timer

Semantic Variation: What should be done when the connected triggering path arrives (first) at an unblocked waiting place or timer?

a) Throw an error; inform of race condition.
b) Launch the continuation path and warn.
c) Warn of the race condition, wait for the regular path to arrive and unblock it as soon as it does.

The default implementation keeps things simple by informing the user of a race condition (option c)). In summary, waiting places and timers are treated very similarly to or-forks and and-joins, although there are a few distinctions. This implementation differs from the original UCMNav version where timers had timer timeout variables, which indicated if the timer should timeout when reaching it (and whether it did timeout, when evaluated later on). jUCMNav's implementation is much more generic and flexible, although a few explanations are required with regards to the evaluation order (see 3.5.6).

6.2.10 Semantic Variation #10: Multiple Active Plug-in Bindings

Semantic variation: What should be done when multiple plug-in binding conditions in a dynamic stub are true?

a) Pick the first one and give warning (what defines the first?)
b) Pick one bound branch with true condition randomly and give warning
c) Launch all bound branches with true conditions in sequence (what sequence?)
d) Launch all bound branches with true conditions in parallel
e) Abort and notify of error
f) Keep the alternatives in the end-result (MSCs have the ALT construct)

The default implementation does either a) or b), depending on the deterministic user-defined preference, just like for or-forks. However, it is much less clear that stubs should conserve their exclusive-or meaning. For example, section 6.1.4 converts use case extensions to dynamic stubs, creating a placeholder for any number of plug-in maps to be executed in an undetermined (and unimportant) order. Furthermore, many workflow patterns would make more efficient use of the concept of stub, as will be discussed in 7.2.1.

6.2.11 Semantic Variation #11: No Active Plug-In Bindings

Semantic variation: What should be done when no plug-in binding condition is true in a dynamic stub?

a) Abort and notify of error
b) Pause until one becomes true, notify of error if this never happens
c) Pick one bound branch randomly and give warning
d) Skip over the stub

The default scenario traversal algorithm does either a) or b), depending on the patient user-defined preference. The only case in which it does d) is if the stub does not contain any plug-ins, as discussed in 3.5.7.

6.2.12 Semantic Variation #12: Multiple Plug-In Bindings in Current Context

Semantic variation: If there are multiple plug-in bindings (for end-points connected to stub exit paths) in the current context, what should be done?

a) Fire all of them in parallel
b) Fire all of them in sequence (which sequence?)
c) Fire only one (which one?)

The default scenario traversal does a). To clarify how this situation could occur, think of a simple map containing an and-join (see Figure 84). This map is a plug-in, used in two different stubs on two other maps. Each of the join's incoming branches is bound to a dif-

ferent parent map. The scenario starts in the two different parent maps, drills down their respective stubs, synchronizes at the and-join and reaches the end point. In this situation, the default implementation behaves similarly to having two tokens united at the and-join until the end-point is reached, before being separated once again. Both parent maps are informed that the work in the stub is complete. If splitting off into parallel paths after synchronizing in the plug-in seems contradictory, the workaround is simply to only bind the end point with one parent.

6.2.13 Semantic Variation #13: Same Stub Exit Path Fired Multiple Times

Semantic variation: If a plug-in binding is traversed multiple times to reach a plug-in map, and the map synchronizes these requests before terminating, how many times is the stub's exit path taken?

a) Only once
b) Once for each time the traversal drilled down the stub.

In this situation, the algorithm diverges from the initial concept and only fires the stub's exit path once (option a)). In retrospect, it might be desirable to offer a consistent view with regards to the previous semantic variation, but the current implementation is conceptually similar to an and-join. Originally the variation used was option b) but undesirable side-effects were observed when a UCM was refactored and the and-join was put inside a stub. As this is a common scenario, priority was given to option a).

6.2.14 Semantic Variation #14: False Postconditions

Semantic variation: If an end point's postcondition is false, what should be done?

a) Abort and report error
b) Continue and warn
c) Warn once and pause until it becomes true, notify of error if this never happens
d) Silently pause until becomes true, notify of error if this never happens

Postconditions are seen as being critical verifications in the traversal algorithm. Hence, the traversal stops if a postcondition is not respected in the default scenario traversal algorithm (option a)), regardless of the patient algorithm flag.

6.2.15 Semantic Variation #15: How Are Start Points Launched?

Semantic Variation: How are start points launched?

a) In parallel
b) In sequence (which order?)

Conceptually, many articles talk about launching UCM start points in parallel. However, both the original UCMNav implementation and the work presented in Chapter 3 launch start points sequentially (option b). The scenario definition creates an ordered list of start points, which allows duplicates. The new algorithm takes the first one and does depth-first processing on it until blocked. When this occurs, the next one is taken until none remains. Parallel execution can still be modeled using start point preconditions, if the patient user-defined preference is activated. Even though the start points are launched sequentially, the flattened UCM could be interpreted differently when exporting to MSC (Chapter 4). Indeed, the MSC export algorithm could treat scenario start points as being run concurrently: this would be an interesting parameter to add.

6.3. Integration of UCM Scenarios and GRL Strategies

During the development of the scenario traversal algorithm presented in Chapter 3, an interesting opportunity was revealed: the bi-directional integration of UCM Scenarios with GRL Strategies. This section presents the general concept of what was prototyped. Figure 85 and Figure 86 present two different activated strategies in a trivial GRL model.

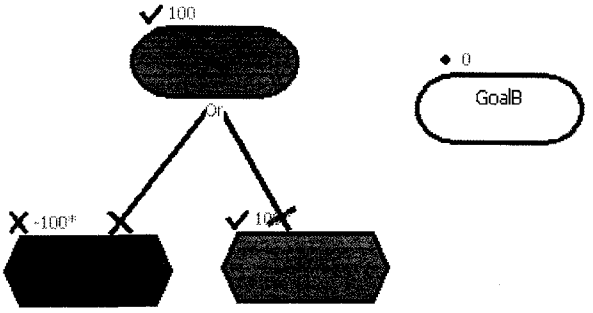


Figure 85 Trivial GRL Model with Active Strategy

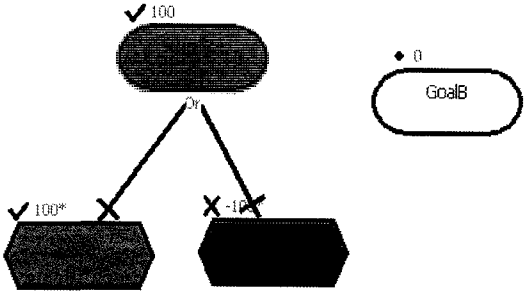


Figure 86 Trivial GRL Model with Second Active Strategy

Typically, the requirements engineer builds such a goal model to represent alternative solutions. Then, separately, a UCM model is created and the different scenarios and architectures are created. The UCM model contains a variable representing which GRL task is active to model the execution flow. This leads to duplication of effort as a Boolean variable must be created for each different task to know if it is active or not. Furthermore, if the scenario behaves differently depending on the satisfaction of certain goals, the results of the strategy execution has to be interpreted and reflected into a variable initialization. This represents much additional work for a tool which is supposed to offer an integrated view of scenarios with goals. As a first step to solving this redundancy issue, a tighter integration between GRL strategies and UCM scenarios was created and added to jUCMNav.

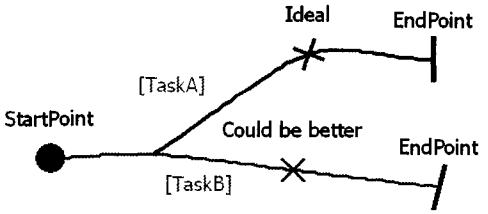


Figure 87 Trivial UCM with Active Scenario

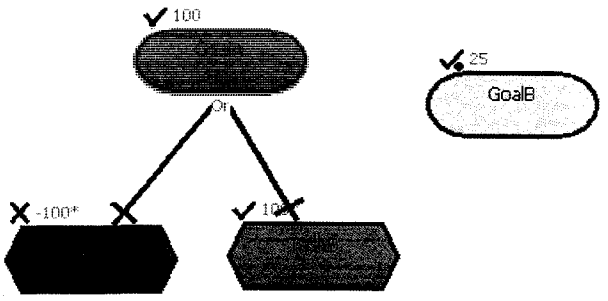


Figure 88 Scenario Traversal Impacts Strategy Evaluations

After the activation of a GRL strategy, the example shown in Figure 87 has an enhanced data model: Integer variables are automatically created for each GRL intentional element definition (see the corresponding option in the preferences of Figure 31). Conditions can then refer to these variables, prefixed by *_GRL_*. For example, the formal condition associated to the bottom or-fork branch is *_GRL_TaskB==100* whereas it is *_GRL_TaskA==100* for the top one. Hence, the scenario traversal algorithm behaves different depending on the active strategy.

Another interesting capability involves the update of these GRL variables in responsibilities. For example, the two responsibilities presented in the UCM have pseudo-code that modifies the automatically created variables (e.g.,*_GRL_GoalB=25;*). At the end of the scenario's execution, the values of these variables are re-assigned to the GRL strategy as can be seen in Figure 88. Any intentional element can have its evaluation read/written; these valuations are not saved permanently with the model elements and therefore no feedback loops (possibly infinite) can be created. In the prototype, if the evaluation value is written back to an element which is not a leaf in the GRL graph, the GRL evaluation propagation algorithm will override the value and the impact of the scenario will not be visible.

6.4. Chapter Summary

This chapter presented satellite work related to the core subjects of this thesis (Chapter 3 and Chapter 4). A plug-in for transforming textual use case to graphical UCM models was presented, illustrating the extensibility of the import/export infrastructure offered by jUCMNav while helping with the elicitation of some requirements for the scenario tra-

versal mechanism. The scenario traversal algorithm's semantic variations were also listed to enable the future creation of other traversal algorithms. Finally, a prototype was created to integrate GRL strategies to UCM scenarios in order to simplify the redundancy involved in modelling scenarios that depend on or impact strategies.

Chapter 7. Conclusions

This thesis presented enhancements to jUCMNav that strengthen the tool support for the analysis and transformation of UCM models. The contributions extend the work done in the course of my bachelor's software engineering capstone project which focused on improving tool support for designing (editing) UCM models [28]. This chapter reiterates the thesis' goals and contributions before introducing future enhancements to be considered for jUCMNav and the User Requirements Notation itself.

7.1. Goals and Contributions

Figure 89 recalls the thesis goals and tasks presented in section 1.2 with annotations that indicate how the chapters of this thesis are related to the various tasks.

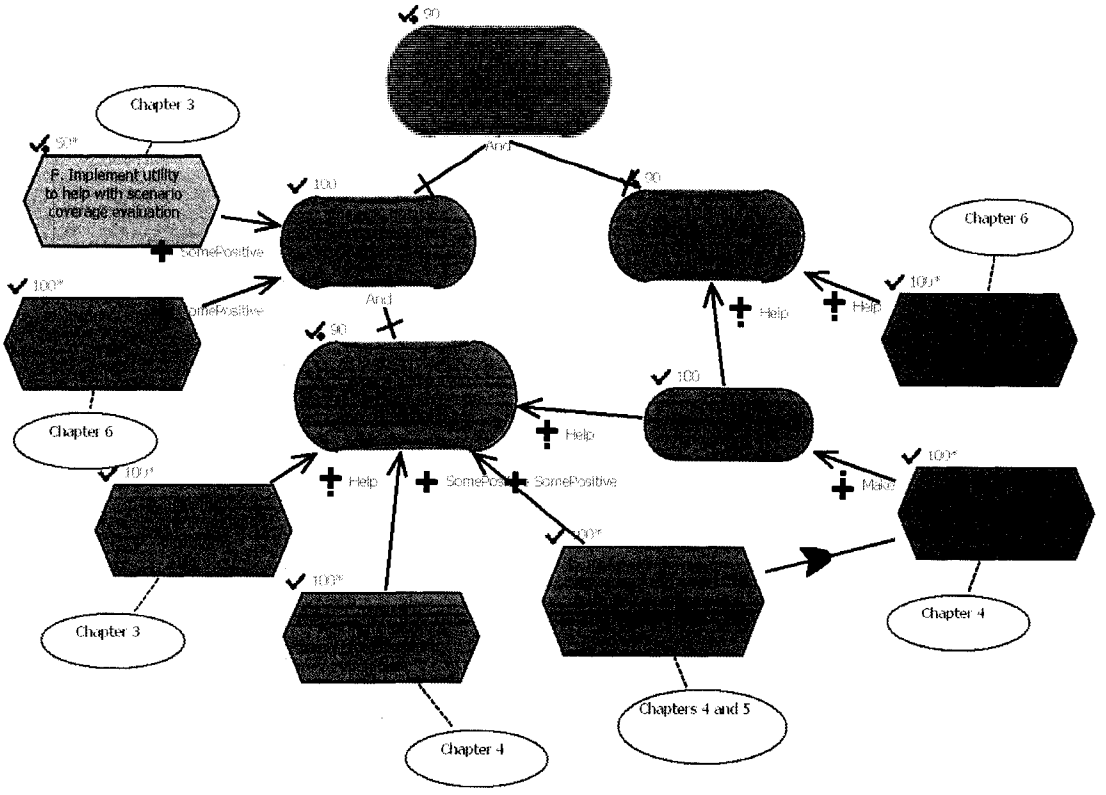


Figure 89 Thesis Goals and Contributions with Corresponding Chapters

A detailed list of contributions was declared in section 1.3. In summary:

- Vastly improved, completely re-engineered, UCM scenario traversal
- Extensible import and export plug-in framework
- Identification of UCM traversal semantic variation points
- Better integration between GRL strategies and UCM scenarios
- Illustration via a case study, and additional validation with test cases
- Four conference publications [25][26][27][37]

Finally, as lead developer for jUCMNav, it was a pleasure to see the tool used and validated in a number of different contexts. It is rewarding to see the software being used by, and actually help, a large number of people, especially after contributing over 100 000 lines of code to such a project.[2]

- Graduate and undergraduate courses at the University of Ottawa, Carleton University and other institutions around the world (Brazil, Canada, China, Germany, Hungary, India, Norway, UK, USA, etc.)
- Supporting infrastructure for various publications and theses made by peers and other researchers, including: Daniel Amyot, Gunter Mussbacher, Jean-François Roy, Sepideh Ghanavati, Pengfei Chen, Alireza Pourshahid, Jacques Sincennes, Michael Weiss.
- Industry: Albeit limited in scope, we received feature requests from users in industry.

7.2. Future work

Although a large number of small, incremental enhancements could be made to the contributed work, this section focuses on the more important changes that could be made.

7.2.1 Suggested Improvements for the Scenario Traversal Algorithm

The scenario traversal algorithm would benefit from an iteration focused on parameterization based on the defined semantic variations or simply alternative scenario traversal algorithms. An especially interesting feature would be true concurrent execution which

[2] See StatSVN statistics: http://jucmnav.softwareengineering.ca/jucmnav/statsvn/. StatSVN (http://www.statsvn.org) is an open source project for which I am also lead developer.

could detect more race conditions in UCM models (because of the determinism imposed by the depth first algorithm).

In addition, the pseudo-code could be enhanced to support external operations such as writing to a message log, reading from / write to a file or invoking a URL. Even though the focus of UCMs is not to become an executable notation, this would be interesting in the context of high-level testing based on UCMs [8].

A few productivity enhancements might also be interesting, such as find/replace in pseudo-code and automated scenario generation using reverse engineering techniques.

7.2.2 Suggested Improvements to the MSC Export Plug-In

The MSC export plug-in could greatly be enhanced to support various customizable parameters defining message synthesis techniques and the actual message contents. Documenting the various semantic variations related to the interpretation of UCMs as MSCs would be a pre-requisite to this parameterization. Another interesting project would be to expand the MSC viewer packaged with jUCMNav to become an MSC Editor with focus on forward requirements engineering. With appropriate traceability links, message exchanges could be refined once and remembered even though the MSC model would be regenerated. jUCMNav could become a more complete requirements engineering package, although this work should be seen as a long term objective.

Bisgaard Lassen et al. proposed an approach to generate process descriptions (at the level of UML activity diagrams, Petri Nets, YAWL, or BPEL) from MSCs [9]. In essence, their transformation is the reverse of ours and could easily be adapted to cover UCMs as a target notation. Combining both approaches could enable a roundtrip transformation process.

7.2.3 Suggested Improvements to URN and jUCMNav

Section 6.1.5 discussed enhancements to the UCM definition structure. Currently, URN models contain a list of component definitions and a list of responsibility definitions. It would be interesting to allow for an optional restructuring of these elements. Components could be defined as being sub-components of another component and responsibilities could be associated with components. Other enhancements with regards to inheritance,

instantiation, and abstraction could be desirable, but the simple enhancements listed above would allow for simple model validation. By defining how the elements should be structured and what responsibilities can be executed inside these components, the model could be checked for inconsistencies. For example, if a component is accidentally unbound from its parent, there is no visual indication that this is the case: having an enhanced domain model with a UCM model validation mechanism would easily find these issues. With a few more enhancements, this domain model could even be used for the generation of UML class diagrams from UCMs.

As described in detail in [33], enhancements could be made to the UCM notation in order to bring it in line with other workflow notations [46]. A summary of desirable changes are presented in Table 8. Simply put, many new workflow patterns could be supported by the addition of a few new path node attributes and changes to the traversal algorithm. The proposed changes are related to the various semantic variations listed in 6.2.

As for the tool itself, jUCMNav is due for an iteration focused on scalability and enhanced productivity. Albeit very easy to use, the tool does not currently scale well with a large number of diagrams where the tab-based interface and tree-outline becomes problematic. Furthermore, copy-paste and refactoring capabilities (extract to stub) are become increasingly needed.

Finally, the tool currently supports both UCM and GRL but both notations could be integrated more tightly. There is a possibility for greater synergy and traceability between goal models and their implementation in scenarios. Such potential is demonstrated in section 6.3 but merits further investigation. A simple URN modelling methodology must first be defined which would specify the typical use cases where one builds a scenario model that is strongly related to a goal model in the context of validating alternatives. An explicit link between UCM scenarios and GRL strategies is required as the desired behaviour is probably the visualization of impact on a scenario of a set of GRL strategies. Furthermore, the prototype currently does not make use of the URN links that can be created in jUCMNav nor does it take into consideration actor satisfaction levels. Integration of these two elements as parameters used in the scenario traversal algorithm could help create linear UCMs (or exported MSCs) where the component structure depends on a GRL strategy.

Table 8 Workflow Pattern Suggested URN/jUCMNav Enhancements

Workflow Pattern	Required notation / tool changes
Multiple Choice (paths)	Or-forks in jUCMNav are exclusive-or only. To support multiple choice workflow pattern, the most explicit solution is to add conditions on and-fork branches. The traversal algorithm would fire (in parallel) only the branches that are active. This could also be done implicitly on or-forks (traversal preference).
Multiple Choice (plug-in maps)	New attributes should be added to dynamic stubs to represent a "synchronizing stub". This stub would no longer have an exclusive-or plug-in selection mechanism. It could launch multiple plug-ins in parallel. Simply put, the addition of an expression which determines how many plug-in maps must terminate (all, the first, N plug-ins, etc.) before continuing execution will satisfy all these patterns.
Synchronizing Merge	
N-out-of-M Join	
Discriminator	
Multiple Instances without synchronization	Use the replication factor of a component to execute multiple parallel instances of the same responsibility.
Multiple instances with a priori design time knowledge	Plug-in bindings should be enhanced to support a replication factor expression. This expression is evaluated at runtime and determines the number of simultaneous instances to launch.
Multiple instances with a priori runtime knowledge	
Deferred Choice	A specialized traversal mechanism is required to support these patterns. The interleaved parallel routing is currently an implicit side-effect of the traversal algorithm.
Interleaved Parallel Routing	
Milestone	
Cancel Activity / Case (cancellation pattern)	Improved abort construct / UCM Exceptions.

7.2.4 Suggested Improvements to the Use Case Import Plug-In

The textual use case import plug-in could be enhanced to generate a set of scenarios, the relevant variables and pseudo-code now that these capabilities are present in the tool. This would open the door to a Use Case to MSC transformation, although these could also possibly be generated directly from the state machines generated by UCEd.

The work presented in 6.1 could be expanded to perform the opposite transformation: generating textual use cases from visual UCMs. This would be a great way to document UCM models and could even make use of the various algorithms implemented for the MSC Export presented in Chapter 4. The logical next step would be to aim for round-trip requirements engineering by integrating a Use Case editor directly in jUCM-Nav and immediately seeing the impact of a change in either model in a split view.

Finally, textual use cases offer Use Case Diagrams for an abstract overview of the relationships between the various scenarios. Although UCEd does not support Use Case diagrams, automatically reverse-engineering such a diagram from a UCM model would be a tremendously valuable enhancement to the tool. There has already been some indirect work in this direction with the Concerns View implemented in the context of Aspect-Oriented URN [34].

7.3. Chapter Summary

This chapter revisited the thesis' goals and contributions. Furthermore, possible future work related to the diverse subjects that were covered by the contributions was presented. With the contributions presented in this thesis, the groundwork is now set for another full iteration of work on jUCMNav. First, the individual UCM and GRL notations should be strengthened to better compete with the other major workflow / goal modelling notations. Second, the competitive advantage that URN offers, the integration of a scenario and a goal modelling notation, should be made more apparent with deeper ties between the two sub-notations. Finally, round-trip requirements engineering using UCMs as the missing piece of the puzzle should be explored in greater depths.

References

All Web references last accessed in August 2007 unless stated otherwise.

[1] Alexander, I. and Stevens, R.: *Writing Better Requirements*. Addison-Wesley Professional, 2002.

[2] Amyot, D.: Introduction to the User Requirements Notation: Learning by Example. In: *Computer Networks*, 42(3), 285-301, 21 June 2003.

[3] Amyot, D., Cho, D.Y., He, X., and He, Y.: Generating Scenarios from Use Case Map Specifications. In: *Third International Conference on Quality Software (QSIC'03)*, November 2003, 108-115

[4] Amyot, D. and Eberlein, A.: An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. *Telecommunications Systems Journal*, 24:1, 61-94, September 2003.

[5] Amyot, D., Echihabi, A., and He, Y.: UCMEXPORTER: Supporting Scenario Trans-formations from Use Case Maps. In: *NOuvelles TEchnologies de la RÉpartition (NOTERE'04)*, Saïdia, Morocco, June 2004. http://ucmexporter.sourceforge.net

[6] Amyot, D., Farah, H., and Roy, J.-F.: Evaluation of Development Tools for Domain-Specific Modeling Languages. In: R. Gotzhein, R. Reed (Eds.) *SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modelling*, Kaiserslautern, Germany, May 2006. LNCS 4320, Springer, 183-197.

[7] Amyot, D. and Mussbacher, G.: On the Extension of UML with Use Case Maps Concepts. In: *3rd International Conference on the Unified Modeling Language (<<UML2000>>)*, York, UK, October 2000. LNCS 1939, Springer, 16-31.

[8] Amyot, D., Roy, J.-F., and Weiss, M.: UCM-Driven Testing of Web Applications. In: A. Prinz, R. Reed, and J. Reed (Eds.) *12th SDL Forum (SDL 2005)*, Grimstad, Norway, June 2005. LNCS 3530, Springer, 247-264.

[9] Bisgaard Lassen, K., van Dongen, B.F., and van der Aalst, W.M.P.: Translating Message Sequence Charts to other Process Languages using Process Mining. *BETA Working Paper Series, WP 207*, Dept. Technology Management, Technische Universiteit Eindhoven, The Nederlands, March 2007. http://ga1717.tm.tue.nl/wiki/publications/beta_207

[10] Boyko, A., Boyko, T., Kovalenkov, M., and Abumohammad, T.: *MSC Viewer*. CSI 4900 project, SITE, University of Ottawa, April 2005. Now packaged with jUCMNav.

[11] Cockburn, A.: *Writing Effective Use Cases*, Addison-Wesley Professional, USA, January 2000, 270p.

[12] Buhr, R.J.A.: Use Case Maps as Architectural Entities for Complex Systems. In: *IEEE Trans. on Software Engineering*, Vol. 24, No. 12, Dec. 1998, 1131-1155.

[13] Eclipse: *The Eclipse Platform*, http://www.eclipse.org/. Accessed April 2007.

[14] Eclipse: *Eclipse Modeling Framework (EMF)*, http://www.eclipse.org/emf/. Accessed April 2007.

[15] Eclipse: *Graphical Editing Framework (GEF)*, http://www.eclipse.org/gef/. Accessed April 2007.

[16] Friedman, D., Wand, M., Haynes, C. T., and NetLibrary Inc.: *Essentials of Programming Languages,* MIT Press, USA, 2001.

[17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, USA, 1995.

[18] Hassine, J., Rilling, J. and Dssouli, R.: Formal Verification of Use Case Maps with Real Time Extensions, In: R. Gotzhein, R. Reed (Eds.) *SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modelling*, Kaiserslautern, Germany, May 2006. LNCS 4320, Springer, 99-114.

[19] ITU-T: Recommendation Z.120 (04/04) Message Sequence Chart (MSC). Geneva, Switzerland, 2004.

[20] ITU-T: Recommendation Z.150 (02/03), User Requirements Notation (URN) – Language Requirements and Framework. Geneva, Switzerland, 2003.

[21] ITU-T, URN Focus Group: *Draft Rec. Z.151 – Goal-oriented Requirement Language (GRL)*. Geneva, Switzerland, Sept. 2003

[22] ITU-T, URN Focus Group: *Draft Rec. Z.152 – UCM: Use Case Map Notation (UCM)*. Geneva, Switzerland, Sept. 2003. http://www.UseCaseMaps.org/urn/

[23] java.net: JavaCC™: *JJTree Reference Documentation*. 2007. https://javacc.dev.java.net/doc/JJTree.html

[24] jUCMNav: *jUCMNav Wiki*, http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/WebHome. Accessed April 2007.

[25] Kealey, J., and Amyot, D., *Enhanced Use Case Map Traversal Semantics*. In: 13th System Design Language Forum (SDL'07), Paris, France, September 2007. LNCS 4745, Springer, 133-149.

[26] Kealey, J. and Amyot, D.: Towards the Automated Conversion of Natural-Language Use Cases to Graphical Use Case Maps. In: *2006 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE06)*, Ottawa, Canada, May 2006. 2377-2380.

[27] Kealey, J., Kim., Y., Amyot, D., and Mussbacher, G.: Integrating An Eclipse-Based Scenario Modeling Environment With A Requirements Management Sys-

tem. In: *2006 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE06)*, Ottawa, Canada, May 2006, 2432-2435.

[28] Kealey, J., Tremblay, E., Daigle, J.-P., McManus, J., Clift-Noël, O., and Amyot, D.: jUCMNav: une nouvelle plateforme ouverte pour l'édition et l'analyse de modèles UCM. In : *5ième colloque sur les Nouvelles Technologies de la Répartition (NOTERE 2005)*, Gatineau, Canada, August 2005. 215-222

[29] Liang, H., Dingel, J., and Diskin, Z.: A Comparative Survey of Scenario-based to State-based Model Synthesis. *5th Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06)*, Shanghai, China, May 2006, ACM Press, 5-12.

[30] Mencl, V.: Deriving Behavior Specifications from Textual Use Cases, In *Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE04)*, Linz, Austria, September 2005, 331-341.

[31] Miga, A.: *Application of Use Case Maps to System Design with Tool Support.* M. Eng. Thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, October 1998.

[32] Miga, A., Amyot, D., Bordeleau, F., Cameron, D., and Woodside M.: Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. In: *Tenth SDL Forum (SDL'01)*, Copenhagen, Denmark, June 2001. LNCS 2078, 268-287.

[33] Mussbacher, G.: Evolving Use Case Maps as a Scenario and Workflow Description Language. In: *Anais do WER07 - Workshop em Engenharia de Requisitos*, Toronto, Canada, May 2007,. 56-67.

[34] Mussbacher, G, Amyot, D., and Weiss, M.: Visualizing Early Aspects with Use Case Maps. To appear in: *LNCS Journal on Transactions on Aspect-Oriented Software Development*, Springer, 2007.

[35] Object Management Group: *Unified Modelling Language Specification Version 2.1.1,* http://www.omg.org , Accessed June 2007.

[36] Roy, J.-F.: *Requirement Engineering with URN: Integrating Goals and Scenarios.* M. Sc. Thesis, School of Information Technology and Engineering (SITE), University of Ottawa, Ottawa, Canada, January 2007.

[37] Roy, J.-F., Kealey, J., and Amyot, D., Towards Integrated Tool Support for the User Requirements Notation. In: In: R. Gotzhein, R. Reed (Eds.) *SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modelling*, Kaiserslautern, Germany, May. LNCS 4320, Springer, 198-215.

[38] Rumbaugh, J., Jacobson, I., and Booch, G.: *Unified Modeling Language Reference Manual,* 2nd edition, Addison-Wesley Professional, 2004, 752 p.

[39] Somé, S.: *Use Cases based requirements engineering with UCEd, UCEd User Guide*, http://sourceforge.net/projects/uced/ Accessed June 2007.

[40] Somé, S.: Beyond Scenarios: Generating State Models from Use Cases. In: *ICSE 2002 Workshop Scenarios and state machines: models, algorithms, and tools*, May 2002.

[41] Störrle, H.: Semantics of Control-Flow in UML 2.0 Activities. In: Bottoni, P., Hundhausen, C., Levialdi, S., und Tortora, G. (Eds.), *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Rome, Italy, 2004. IEEE Computer Society, 235–242.

[42] Tjong, S.F., Hartley, M. and Berry, D.: Extended Disambiguation Rules for Requirements Specifications. In: *Anais do WER07 - Workshop em Engenharia de Requisitos*, Toronto, Canada, May 2007. 97-106.

[43] UCM User Group: *UCMExporter*, http://ucmexporter.sourceforge.net/ Accessed June 2007.

[44] UCM User Group: *Use Case Maps Navigator 2 (UCMNav)*, http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/UCMNavDownload Accessed June 2007.

[45] UCM User Group: *UCM Virtual Library,* http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/UCMVirtualLibrary Accessed June 2007.

[46] Workflow Patterns Initiative: *Workflow Patterns website*, 2007. http://www.workflowpatterns.com

[47] Zhao, T.C. and Ovenmars, M., *XForms, a GUI toolkit for X*, http://savannah.nongnu.org/projects/xforms. Accessed June 2007.

Appendix A: Interfaces

This appendix contains various interfaces used by jUCMNav's scenario traversal algorithm. A good understanding of these interfaces is required in order to write a custom scenario traversal algorithm with the least amount of work.

```
public interface IScenarioTraversalAlgorithm {

    /**
     * Initialize the algorithm.
     *
     * @param env
     *            the environment in which to run the scenario
     * @param scenario
     *            the scenario to be executed.
     */
    public void init(UcmEnvironment env, ScenarioDef scenario);

    /**
     * Initialize the algorithm.
     *
     * @param env
     *            the environment in which to run the scenario
     * @param group
     *            the scenario group to be executed.
     */
    public void init(UcmEnvironment env, ScenarioGroup group);

    /**
     * Initialize the algorithm.
     *
     * @param env
     *            the environment in which to run the scenario
     * @param ucmspec
     *            run all scenarios in ucmspec
     */
    public void init(UcmEnvironment env, UCMspec ucmspec);

    /**
     * Adds a list of {@link ITraversalListener} to the current internal listener list.
     *
     * @param newListeners the new listeners
     */
    public void addListeners(Vector newListeners);


    /**
     * Erase any traversal results we may have obtained.
     */
    public void clearTraversalResults();
```

```
    /**
     * Returns the traversal result for a certain element.
     *
     * @param obj
     *            the element
     * @return the traversal result or null if it does not exist.
     */
    public TraversalResult getTraversalResults(EObject obj);


    /**
     * Execute the scenario in its environment.
     * - Perform initializations
     * - Verify preconditions
     * - Execute the traversal algorithm
     * - Verify postconditions
     * Caller should build warnings using {@link #getWarnings()}
     *
     * @throws TraversalException
     *             fatal errors are returned as traversal exceptions.
     */
    public abstract void traverse() throws TraversalException;

    /**
     * The warnings accumulated during the execution.
     *
     * @return A vector of String instances.
     */
    public abstract Vector getWarnings();

    /**
     * The results of the traversal.
     *
     * @return A HashMap of EObject -> TraversalResult.
     */
    public abstract HashMap getResults();

}
```

Figure 90 IScenarioTraversalAlgorithm

```
public interface ITraversalListener {

   /**
    * Ran some code in the scenario's environment.
    *
    * @param visit Where did we run the code?
    * @param code The code that was run.
    */
   public void codeExecuted(TraversalVisit visit, String code);

   /**
    * A condition was evaluated in the scenario's environment.
    *
    * @param visit Where did we run the code? If is null, this is a scenario pre/post
    *        condition.
    * @param condition What condition did we evaluate?
    * @param result Was it true or false?
    */
   public void conditionEvaluated(TraversalVisit visit, Condition condition,
                                  boolean result );

    /**
     * We are going from a stub to a plugin.
     *
     * @param visit which stub in which thread
     * @param inb which inbinding are we traversing
     */
   public void drillDown(TraversalVisit visit, InBinding inb);
```

```
 /**
  * We are going from an end point back up to its parent map.
  * @param visit which end point in which thread
  * @param outb which outbinding are we traversing.
  */
public void drillUp(TraversalVisit visit, OutBinding outb);

 /**
  * We have left a waiting place / timer. If it was blocked before, it is not anymore.
  * @param visit what did we leave?
  * @param becauseOfCondition because of a condition (true) or because of a path
  *        arrival (false)
  */
public void leftWaitingPlace(TraversalVisit visit, boolean becauseOfCondition);

/**
 * When traversing this element, we started a new thread. The ThreadID can be obtained
 * by querying the TraversalVisit.
 *
 * It might be issued from some merger, see {@link #threadsMerged(List, int)} and
 * {@link #threadSplit(int, List)}
 *
 * @param visit Where did we start the thread?
 */
public void newThreadStarted(TraversalVisit visit);

/**
 * We aborted a node, kicking it out of the waiting list; it will never be
 * attempted again.
 *
 * @param visit The node that was aborted
 */
public void pathNodeAborted(TraversalVisit visit);

/**
 * We are attempting a node.
 *
 * @param visit The node that is being attempted.
 */
public void pathNodeAttempted(TraversalVisit visit);

/**
 * We are blocking a node by pushing it into our waiting list.
 * @param visit The node being blocked.
 */
public void pathNodeBlocked(TraversalVisit visit);

/**
 * We are unblocking a node, pulling it out of the waiting list. This can either mean
 * it was unblocked by some other traversal element or we are attempting it and might
 * push it back onto the waiting list.
 * @param visit The node being unblocked.
 */
public void pathNodeUnblocked(TraversalVisit visit);

/**
 * We have successfully traversed this element.
 *
 * @param visit The node that was traversed.
 */
public void pathNodeVisited(TraversalVisit visit);

 /**
  * While traversing, a certain thread died. It might be merged into something else,
  * see {@link #threadsMerged(List, int)} and {@link #threadSplit(int, List)}
  *
  * @param threadID which thread?
  */
public void threadDied(int threadID);
```

```
    /**
     * Multiple threads were merged into a single one.
     *
     * @param oldThreadIDs a list of Integers, reprensenting dead thread ids.
     * @param newThreadID the  new thread id
     */
    public void threadsMerged(List oldThreadIDs, int newThreadID );

    /**
     * A single thread was split into multiple threads.
     *
     * @param oldThreadID the original thread.
     * @param newThreadIDs a list of integers representing the new threads.
     */
    public void threadSplit(int oldThreadID, List newThreadIDs);

    /**
     * A timer has timed out.
     *
     * @param visit which timer?
     * @param becauseOfCondition because of a condition (true) or because was forced to
     *        timeout (false)
     */
    public void timerTimeout(TraversalVisit visit, boolean becauseOfCondition);


    /**
     * All scenarios were traversed and the algorithm will not be sending anything more
     * to this listener.
     */
    public void traversalEnded();

    /**
     * The algorithm has finished traversing a scenario.
     *
     * @param env the {@link UcmEnvironment} in which it was traversed.
     * @param scenario the scenario that was run.
     */
    public void traversalEnded(UcmEnvironment env, ScenarioDef scenario);

    /**
     * The algorithm is starting to traverse a scenario.
     *
     * @param env the original {@link UcmEnvironment}
     * @param scenario the scenario to be ran.
     */
    public void traversalStarted(UcmEnvironment env, ScenarioDef scenario);
}
```

Figure 91 ITraversalListener

Appendix B: Case Study – Additional Details

This chapter provides additional details on the case study presented in Chapter 5. Please refer to the case study for explanations. The case study is freely available on jUCMNav's wiki site [24].

Additional Scenario Descriptions and Definitions

Figure 92 is the base scenario which is included by all other scenarios. It is not intended to be executed but is presented here for completeness. The customer starts shopping and the warehouse employee awaits an order. Variables are initialized with default values and overridden in the other scenarios. A simple postcondition is present to ensure that the scenario did not block in mid-process.

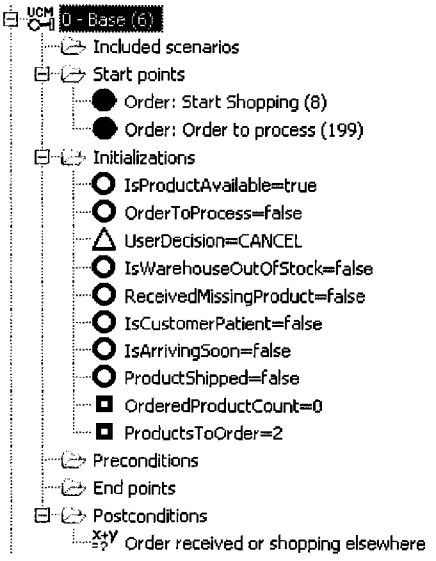


Figure 92 Scenario Definition: Base Case

Figure 93 represents a scenario where the user arrives to shop but can't find anything worth purchasing; the customer leaves without placing an order.

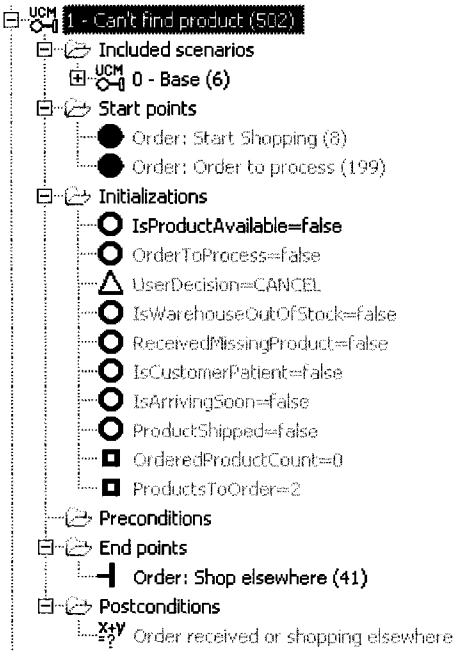


Figure 93 Scenario Definition: Can't Find Product

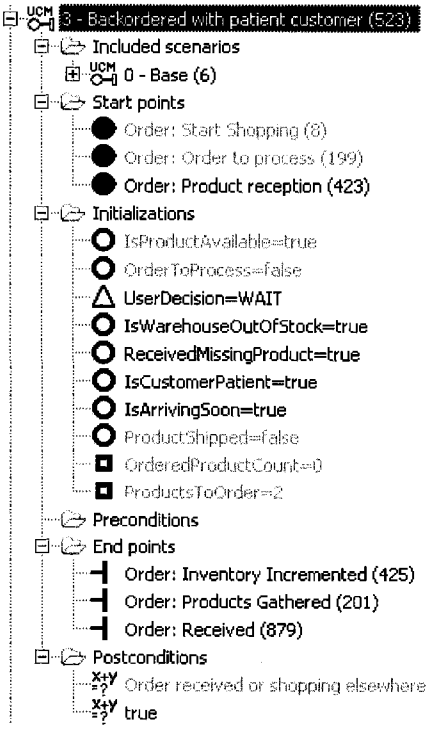


Figure 94 Scenario Definition: Backordered with Patient Customer

Figure 94 shows the case where at least one of the items that the customer has ordered is backordered. The customer is patient and decides to wait for it to arrive, regardless of how long it should take. On the other hand, Figure 95 shows the case where the customer cancels the backordered products but since the warehouse has some items in stock, the customer will still order the in-stock items.

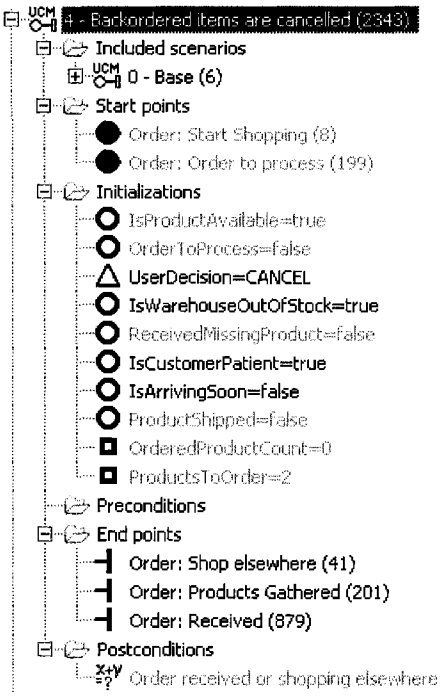


Figure 95 Scenario Definition: Backordered Items Are Cancelled

Figure 96 represents the case where the customer is so unhappy that some items are backordered that he cancels the whole order. Moreover, the scenario also represents the case when all ordered items are backordered and the customer is impatient. Finally, Figure 97 represents an infinite loop: the customer is patient but the warehouse never receives the backordered product and hence never fulfills the order.

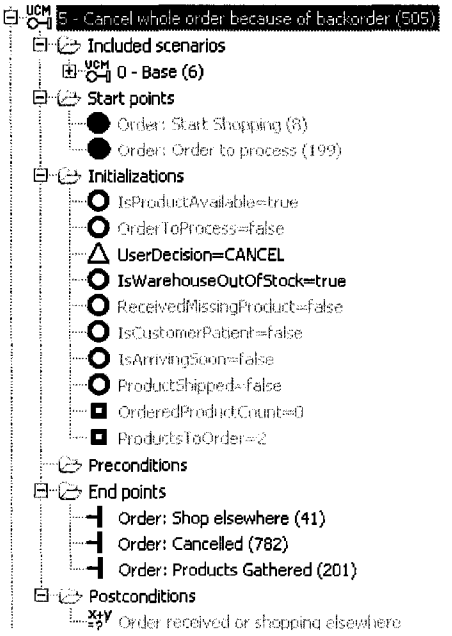


Figure 96 Scenario Definition: Cancel Whole Order Because of Backorder

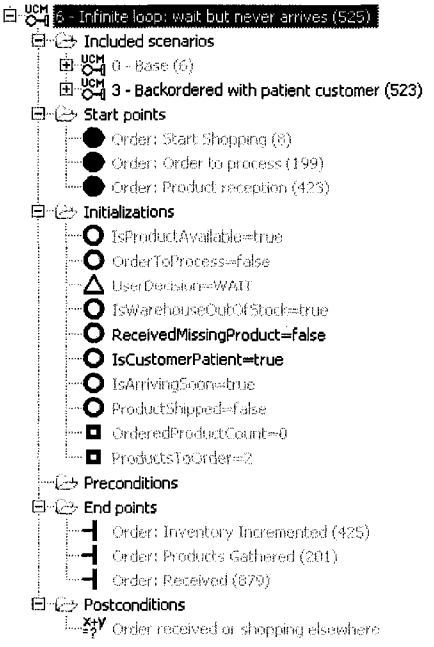


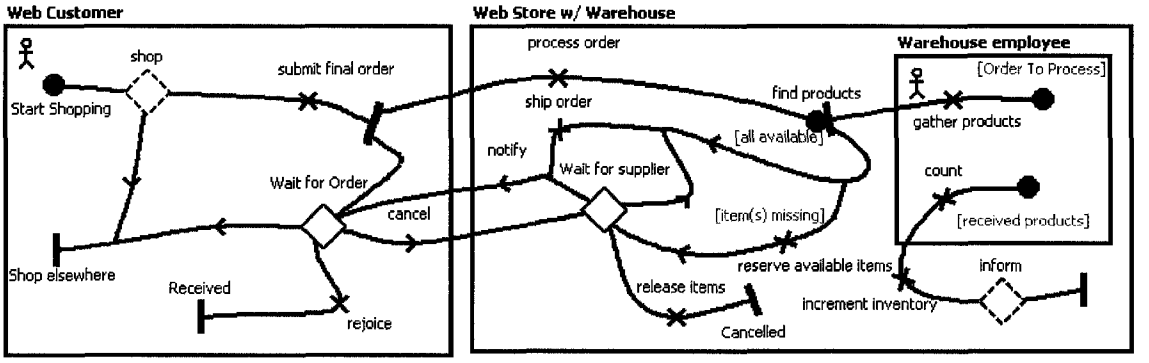
Figure 97 Scenario Definition: Infinite Loop Because Product Never Arrives

Scenario Execution

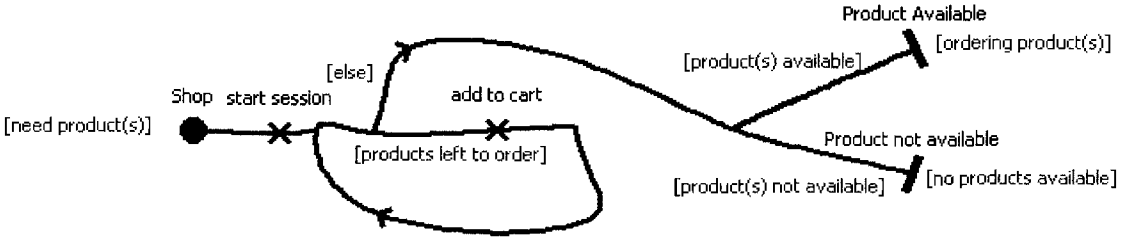
Except for scenario 1, the shop plug-in is the same as the one presented in Chapter 5.

Scenario 1 – Can't find product

Parent map:

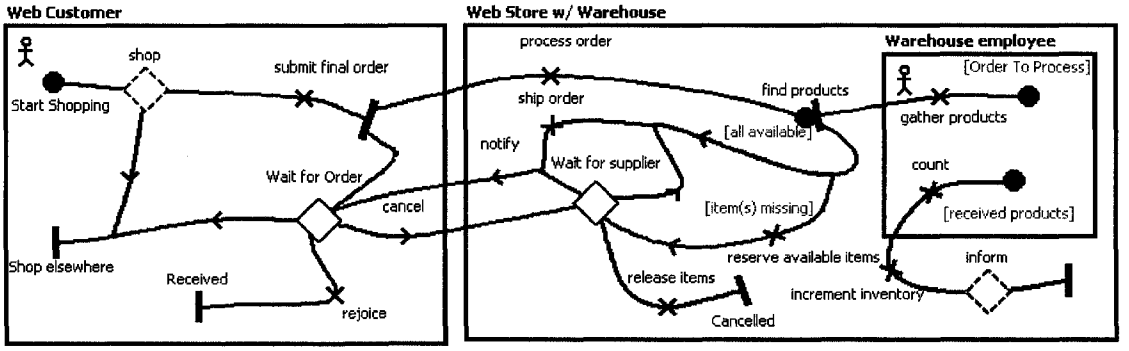


Shop:

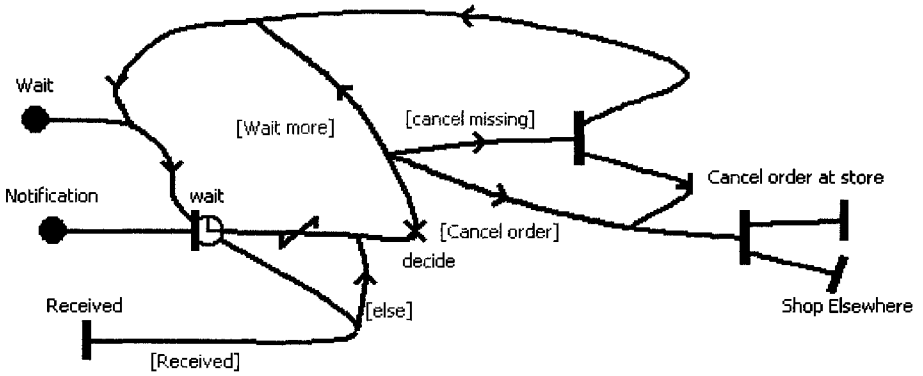


Scenario 4 – Backordered Items Are Cancelled

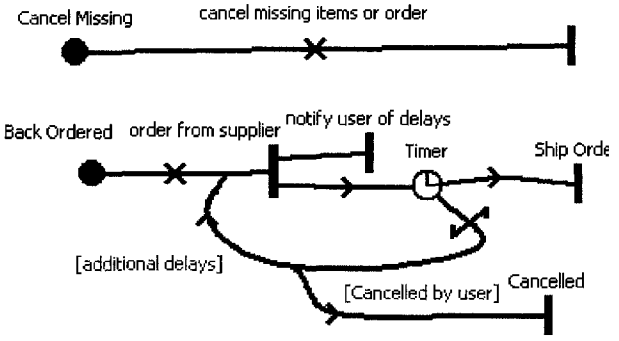
Parent Map:



Wait for Order:

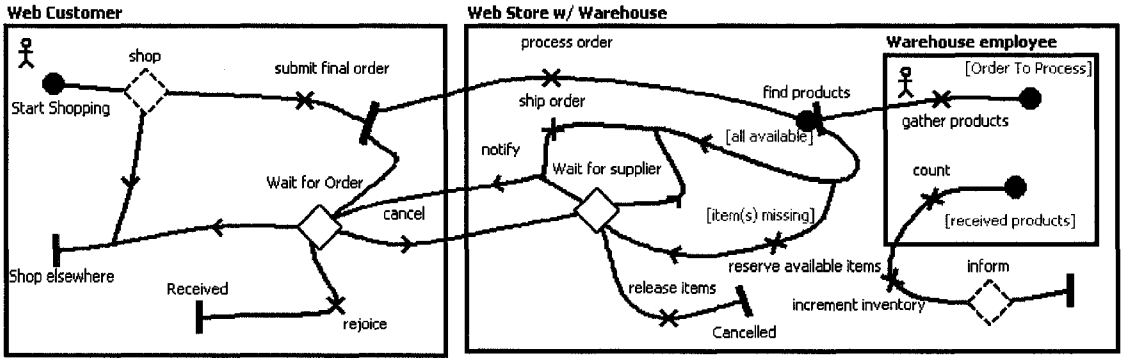


Wait for Supplier:

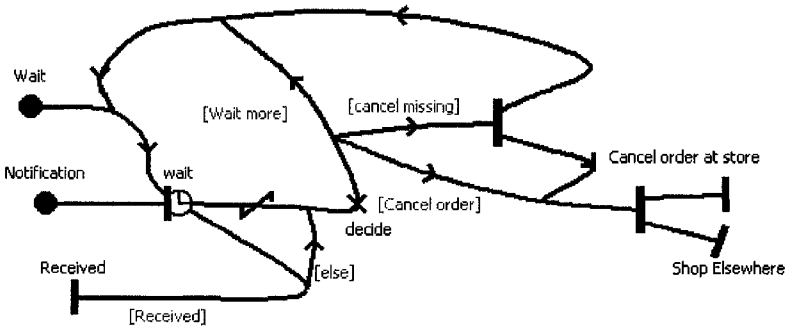


Scenario 5 – Cancel Whole Order Because of Backorder

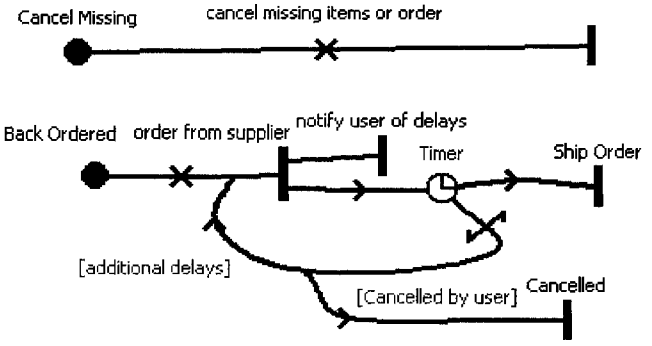
Parent Map:



Wait for Order:

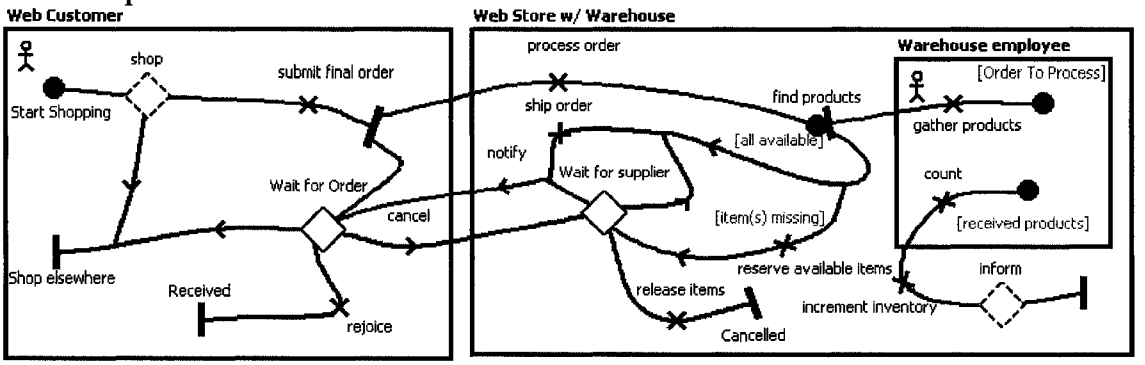


Wait for Supplier:

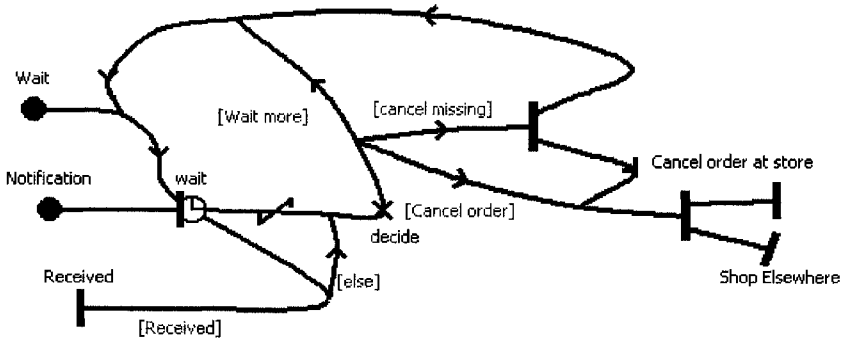


Scenario 6 – Infinite loop: wait but never arrives.

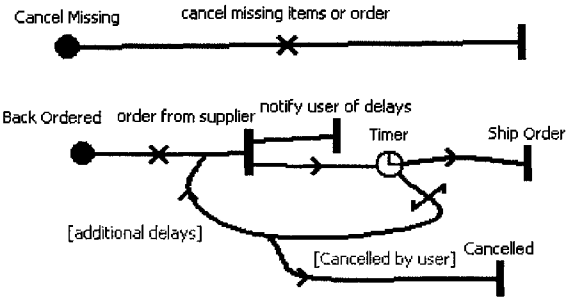
Parent Map:



Wait for Order:

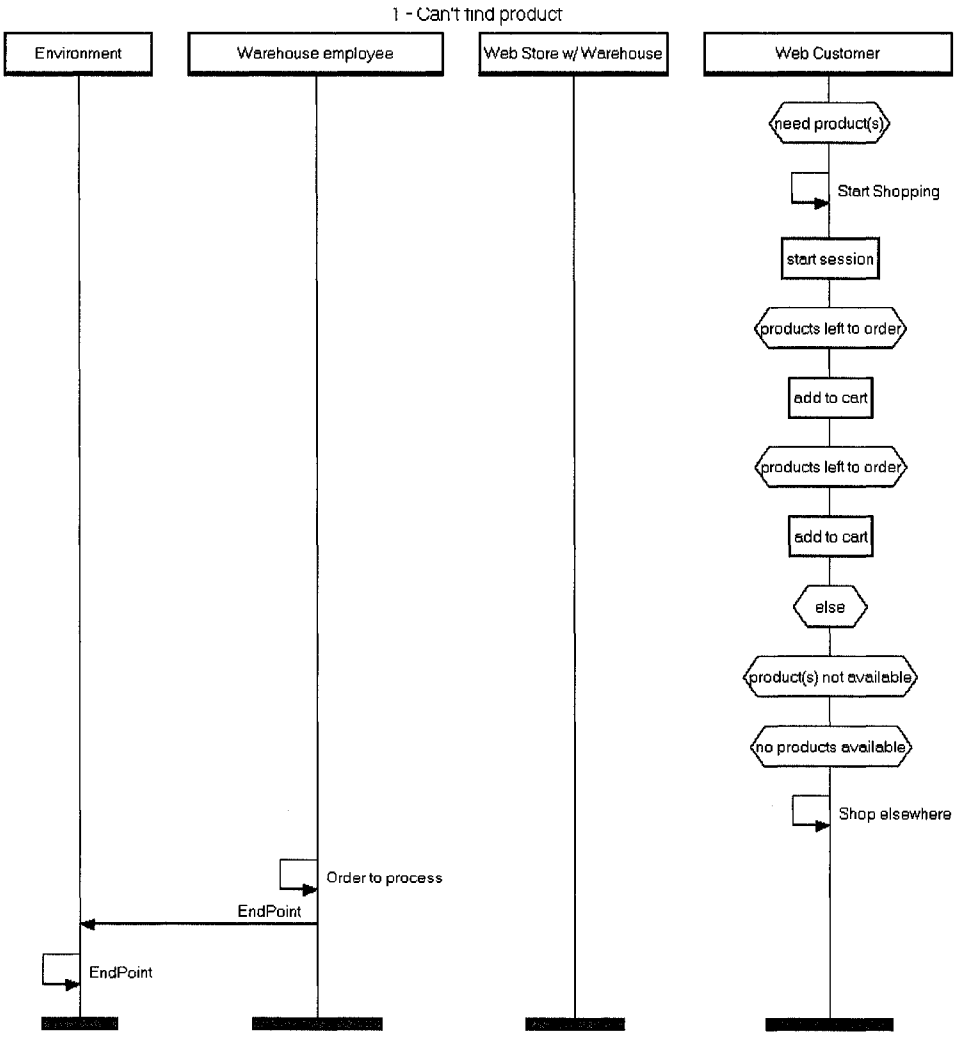


Wait for Supplier:

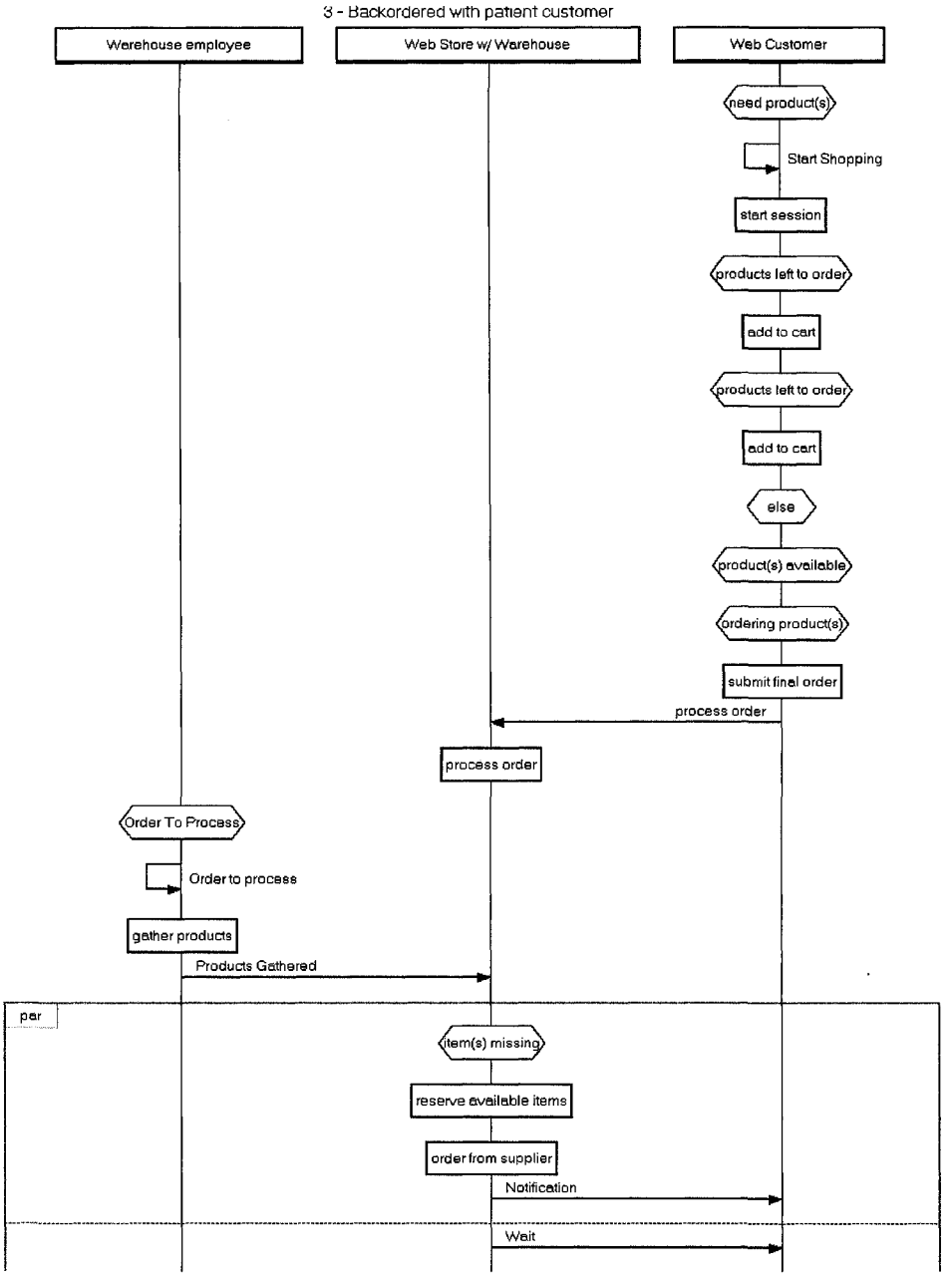


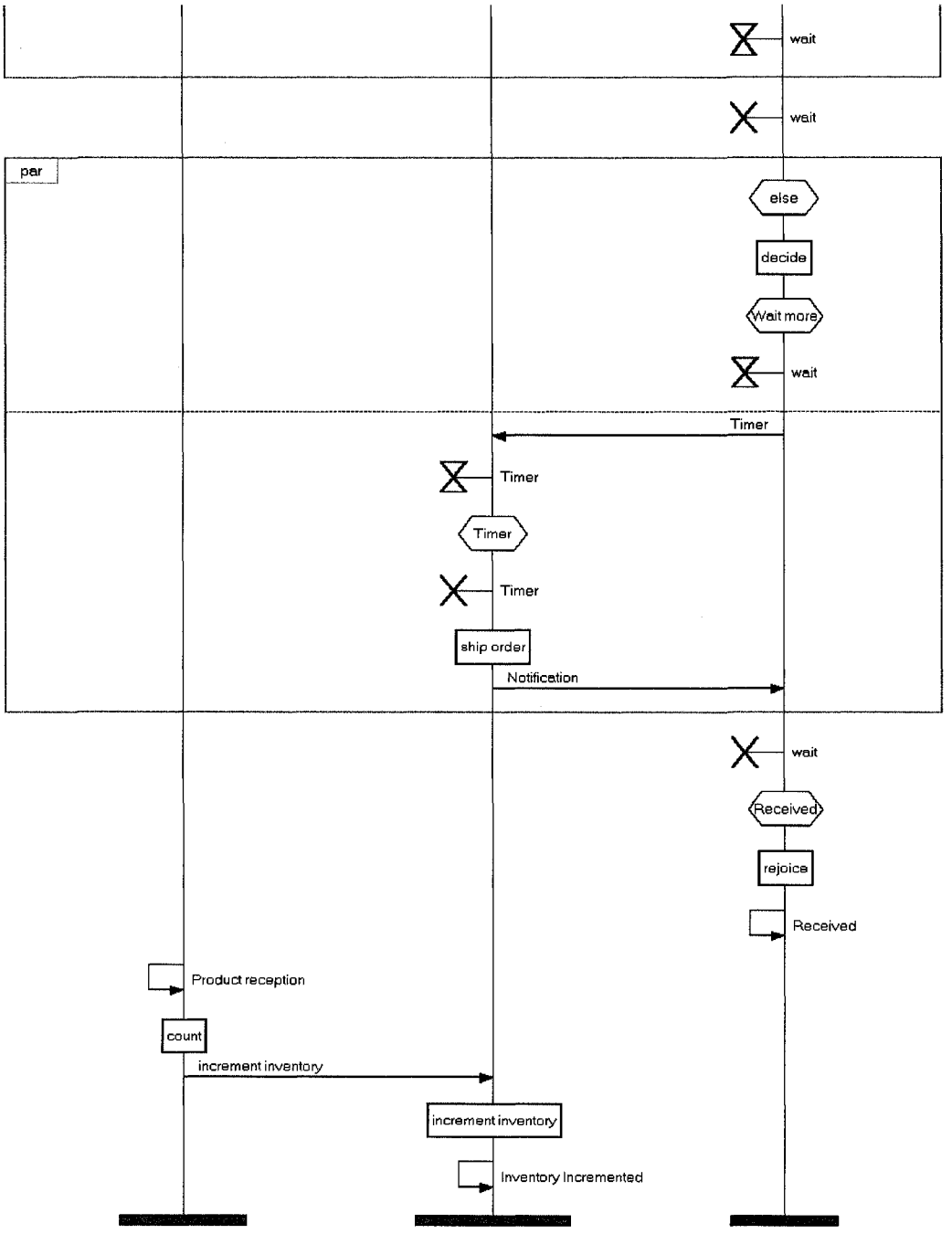
Generated Message Sequence Charts

Scenario 1 – Can't find product

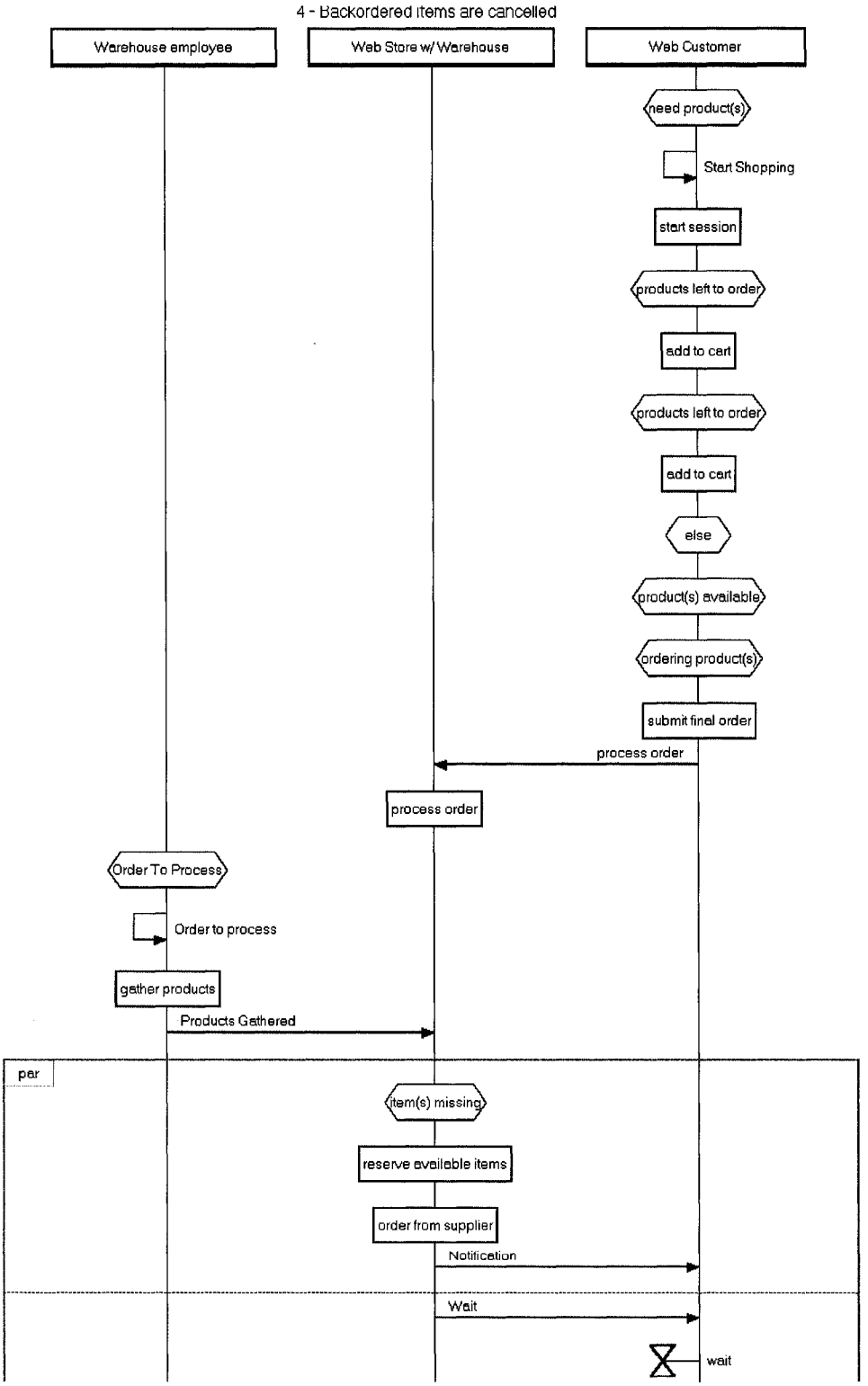


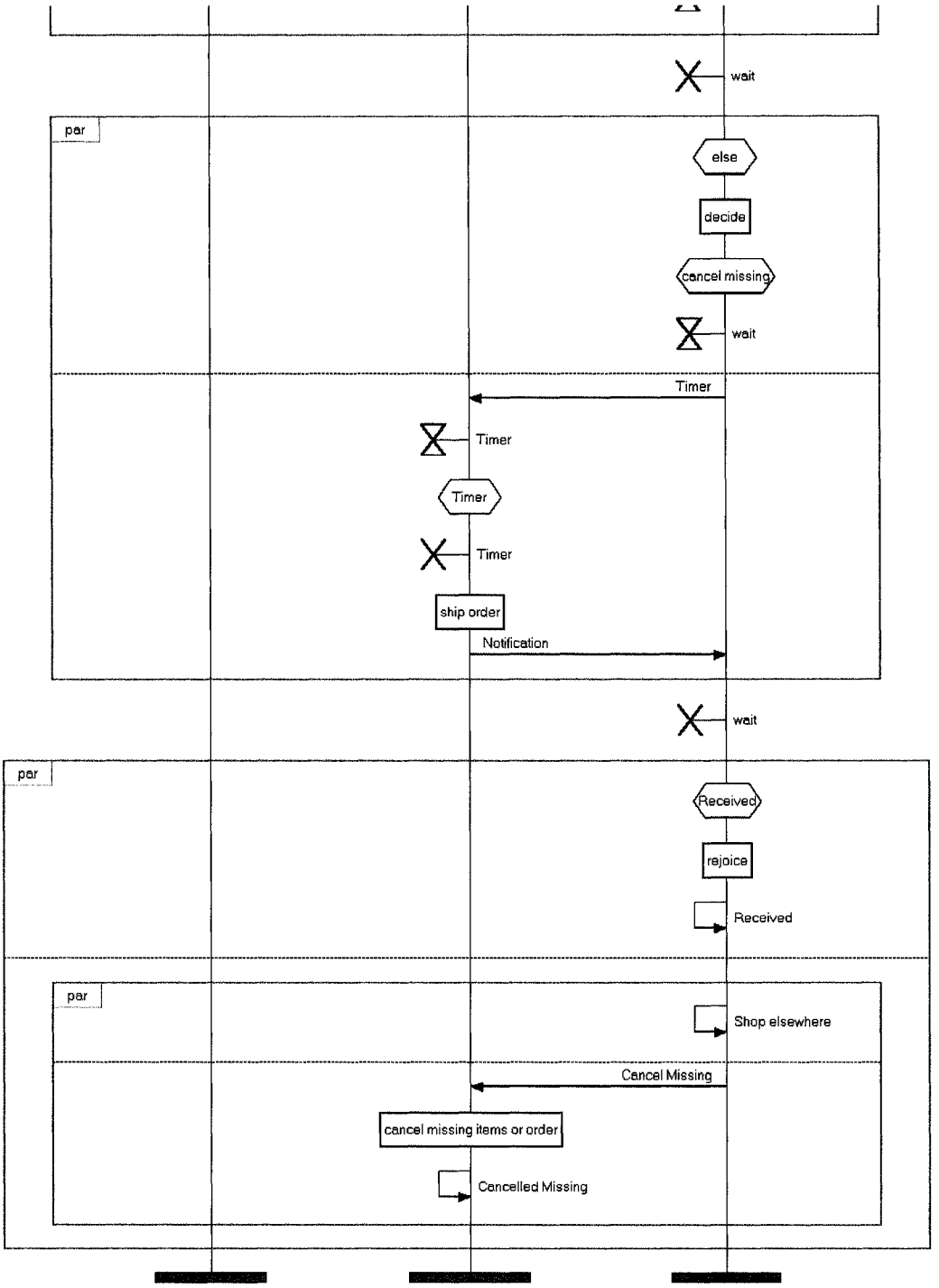
Scenario 3 – Backordered with patient customer



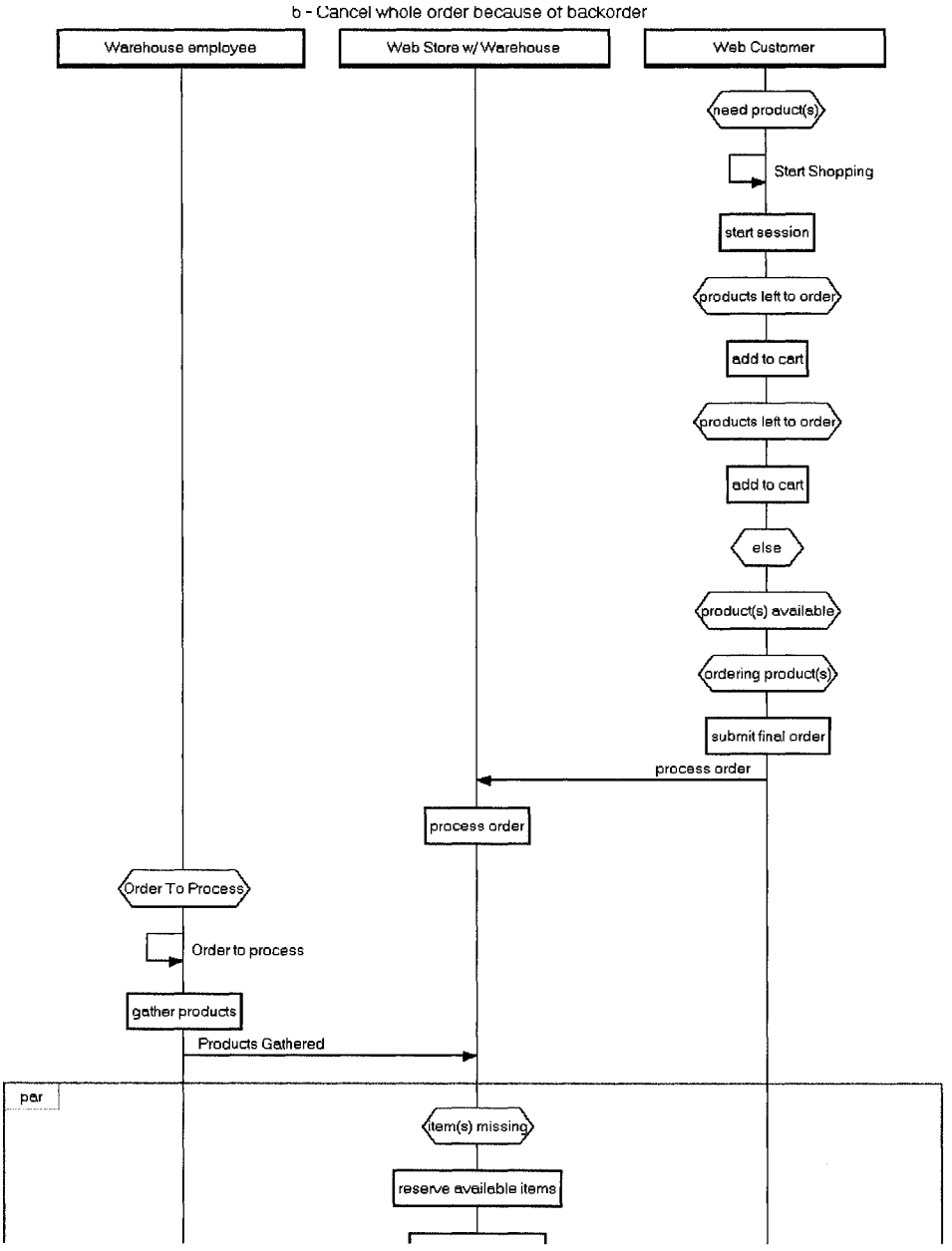

wait
wait
par
else
decide
Wait more
wait
Timer
Timer
Timer
Timer
ship order
Notification
wait
Received
rejoice
Received
Product reception
count
increment inventory
increment inventory
Inventory Incremented

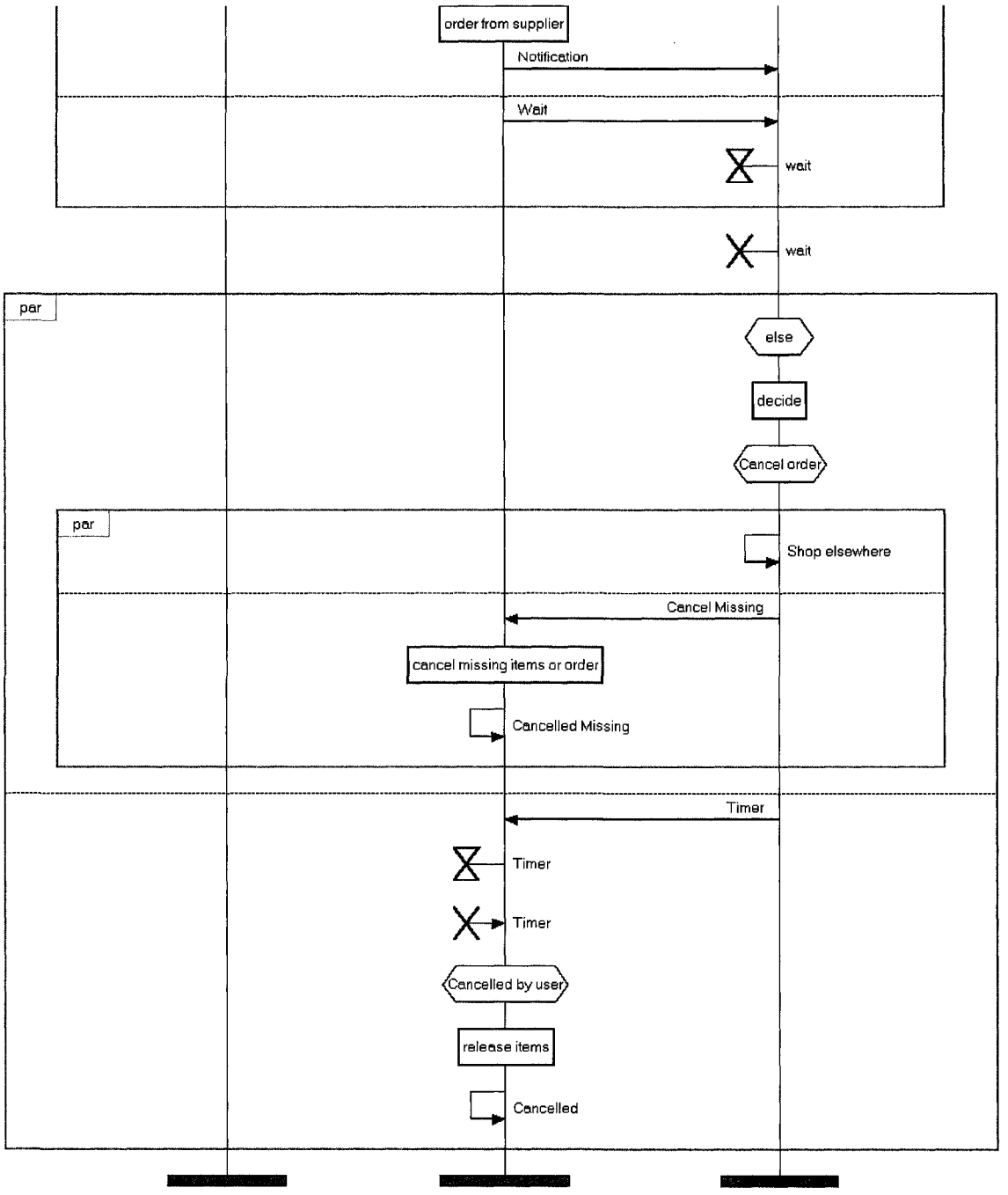
Scenario 4 – Backordered Items Are Cancelled




wait
par
else
decide
cancel missing
wait
Timer
Timer
Timer
Timer
ship order
Notification
wait
par
Received
rejoice
Received
par
Shop elsewhere
Cancel Missing
cancel missing items or order
Cancelled Missing

Scenario 5 – Cancel Whole Order Because of Backorder




order from supplier
Notification
Wait
wait
wait
par
else
decide
Cancel order
par
Shop elsewhere
Cancel Missing
cancel missing items or order
Cancelled Missing
Timer
Timer
Timer
Cancelled by user
release items
Cancelled

Scenario 6 – Infinite loop: wait but never arrives. (interrupted after 25 attempts)