APPLYING STATIC SLICING TO UCM REQUIREMENTS SPECIFICATIONS

BY

TAHA HUSSEIN BINALIALHAG

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

SOFTWARE ENGINEERING

DECEMBER 2016

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **TAHA HUSSEIN BINALIALHAG** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING**.

Thesis Committee

[illegible]

Dr. Jameleddin Hassine (Adviser)

M Alshayeb 2/5/2017

Dr. Mohammad Alshayeb (Member)

[illegible] April 26, 2017

Dr. Sami Zhioua (Member)

[illegible] 2/5/2017

Dr. Khalid Al-Jasser
Department Chairman

[illegible]

Dr. Salam A. Zummo
Dean of Graduate Studies

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

14/5/17

Date

To my parents

ACKNOWLEDGMENTS

All thanks to Allah almighty who gave me the ability, will, determination, knowledge, and patience to complete my thesis.

I would like to acknowledge, with deepest appreciation and gratefulness, the support, encouragement, valuable time and guidance given to me by Dr. Jameleddine Hussine, who served as my major advisor and mentor. Thereafter, I am deeply indebted and grateful to Dr. Mohammad Alshayeb, and Dr. Sami Zhioua, my committee members, for their guidance, and support. I am grateful to Dr. Daniel Aymot for his valuable support and cooperation.I would like to thank KFUPM and Hadramout Establishment of Human Development for giving me the opportunity to pursue my M.S. degree. I would like thank my friends Majdi Bin Salman, Hassan Alkaf, and Omar Jafar for the great support they gave me. My sincere thanks to my precious family for the support, and prayers during my academic journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME:	Taha Hussein Binalialhag
TITLE OF STUDY:	Applying Static Slicing to UCM Requirements Specifications
MAJOR FIELD:	Software Engineering
DATE OF DEGREE:	December 2016

Requirement Specification is getting more attention as a crucial stage in software development life cycle. As requirements descriptions evolve, they become more sophisticated. Hence they rapidly become difficult to understand and to maintain. Therefore, developing methods to assist the comprehension, and maintenance of requirements specification has gained more importance. The Use Case Maps (UCM) language, part of the standard ITU-T User Requirements Notation (URN), is a visual modelling notation that aims at describing requirements at a high-level of abstraction. A UCM specification is used to integrate and capture both functional (based on causal scenarios that represent behavioral aspects of a system) and architectural (system components bound to functional scenarios) aspects. As UCM models evolve, they rapidly become hard to understand and to maintain. In this *thesis, we propose a slicing technique for the Use Case Maps language. The goal of the proposed work in this thesis is to assist maintainers in understanding a UCM requirements specification with respect to a particular maintenance task.*

CHAPTER 1

INTRODUCTION

Program Slicing, proposed by Weiser [2], is a reduction technique used to decrease the size of a program source code by keeping only the lines within a program that are related to the execution of a specific slicing criterion specified by the user. Program slicing has been initially used as a debugging approach in order to enhance software comprehension [3, 4]. Furthermore, slicing has been used in other applications such as testing [5, 6, 1], program integration and differencing [7], reverse engineering [3], software maintenance [8], dead code removal [9, 10], integration [7], program segmentation [11], model checking [12], and garbage collection [13]. The large number of techniques for program slicing has led to many surveys [14, 15, 1, 16, 17, 18, 19]. As software modeling gained in popularity and became a well-accepted practice in industry in many application domains, researchers have refined reduction techniques and moved from focusing only on reducing source code to applying reduction techniques to software models. Models are used to describe different facets of a system, e.g., static structure, dynamic behavior, etc., at various levels of abstraction, throughout the system development life cycle. As models grow in size, they become difficult to understand, debug, and maintain, hence impractical [20]. To address a maintenance task, it may be required to analyze a model with respect to one specific functionality, feature, scenario, etc. [21]. Therefore, reduction techniques are required in order to reduce and simplify models with respect to an element of interest. Various types of slicing techniques and criteria are needed because there can be various applications that need slices with different properties. In the recent years, the application of slicing techniques has been extended to include diverse software artifacts [22] including requirements models [23, 24, 25], formal specifications [26, 27], and software architecture [28].

1.1 Motivation

The estimated cost of software maintenance ranges from 50% to 80% of the overall software budget [29, 30]. The tasks applied during software maintenance can be categorized into four types: adaptive, perfective, preventive, and corrective [31]. One of the most vital facets of software maintenance is to comprehend the software in order to apply changes to it. Understanding a program requires studying its documentation and source code in order to achieve an adequate level of comprehension for a particular maintenance task [32]. The process of program comprehension is time-consuming, and during the maintenance phase, program comprehension consumes between 62% and 47% of the overall time for corrective and adaptive tasks, respectively [33], and overall, it is reported that understanding the software at hand can take up to 60% of the effort of software engineering [34]. Consider that we want to maintain the C-program taken from [35] and shown in figure 1 to be maintained.The program is used to read the marks of an exam and produce the number of passed scores, number of failed scores, rate of passes , rate of *excellent*, *very good*, and *good* scores, and average score. However, when executing the program, the output of average score is extremely low.

At this point, the developer is required to review the code and understand the reason of this unexpected output. This maintenance task might be time consuming since the developer will have to read many lines of code which are not related to the bug. Obviously, the effort spent on inspecting lines of code that do not have any impact on *average* value is a wasted effort. Slicing can reduce the size of the program so that inspection is limited to only those lines of code that have impact on the value of *average*. The generated slice is shown in figure 2 where the size of the slice is one third of the original program. Reviewing the program code can be faster after eliminating the unrelated lines. Quicker than reading all lines of code, the developer can detect the bug located in $line8$ in figure 1 ($line4$ in figure 2) where variable $TotalMarks$ is initialized inside the *while* loop instead of being initialized before it.

Requirements gathering and elicitation represent a crucial step in the software system development life cycle. Requirements specifications are getting more attention as new requirements description languages emerged, introducing new analysis techniques. The Use Case Maps (UCM) language, part of the ITU-T standard User Requirements Notation (URN) [36], is a visual modeling technique used to integrate and capture functional requirements as causal scenarios that represent behavioral facets at a high level of abstraction. A UCM model integrates the behavioral view as well as the architectural view of a system. As UCM models evolve over time, they will integrate dozens of maps that are themselves composed of hundreds of constructs, which make such amount of information humanly unmanageable even by notation experts, as stated by Genon et al. [37]. According to the authors [37], one of the main reasons causing this issue is the language lack of horizontal decomposition.

In this Thesis, we aim at helping maintainers understand large and complex UCM requirements specifications. More specifically, we are interested in developing a slicing algorithms for the UCM language in order to help software engineers understand complex UCMs prior to performing a maintenance task; hence increasing their productivity and reducing the cost of typical maintenance tasks.

C-Program 1: A program to be maintained

```
1- Pass = 0 ;
2- Fail = 0;
3- Count = 0;
4- Excellent = 0;
5- VeryGood = 0;
6- Good = 0;
7- while(!eof()){
8- TotalMarks=0;
9- scanf("%d",Marks);
10- if(Marks> = 40)
11- pass = pass + 1;
12- if(Marks<40)
13- Fail = Fail + 1;
14- if(Marks> = 35)
15- Excellent = Excellent + 1;
16- elseif(Marks> = 35)
17- VeryGood = VeryGood + 1;
18- else 19- Good = Good + 1;
20- Count = Count + 1;
21- TotalMarks = TotalMarks + Marks;
22- }
23- printf("Out of %d, %d passed and %d failed",Count, Pass, Fail);
24- PassRate = Pass/Count * 100 ;
25- printf("Pass rate is %d",PassRate);
26- average = TotalMarks/Count;
27- /* point of interest */
28- printf("Average= %d",average);
29- ExcellentRate = Excellent/Count * 100;
30- VgoodRate = VeryGood/Count * 100;
31- GoodRate = Good/count * 100;
32- printf("Excellent rate=%d, VeryGood rate=%d, Good
  rate=%d",ExcellentRate,VgoodRate,GoodRate);
```

C-Program 2: Backward slice with respect to Line 28, variable:*average*

```
1- Count = 0;
2- while(!eof()){
3- scanf("%d",Marks);
4- TotalMarks=0;
5- Count = Count + 1;
6- TotalMarks = TotalMarks + Marks;
7- }
8- average = TotalMarks/Count;
9- printf("Average= %d",average);
```

1.2 Problem Statement

Having sketched the background of our research, we now formulate the problem statement and the main goal of this thesis proposal. The problem statement is denoted as follows:

"As Use Case Maps requirements specifications evolve, they become very complex to comprehend and to maintain. The actual UCM framework tool (jUCMNav) lacks features that facilitate the understanding of selected parts of interest within a UCM specification."

Therefore, the goal of our research is denoted as follows: "The goal of our research is to help the comprehension of complex UCM specifications. More particularly, our goal is to investigate the use of reduction techniques in order to help requirements engineers comprehend models written using the UCM notation."

1.3 Research Hypothesis

Although much work has been done in the use of reduction techniques at the program [15] and the model level [20], applying such techniques to requirements models, remains an open research subject. In this thesis, we apply the well-known program slicing technique to the Use Case Maps language.

The research hypothesis is denoted as follows:

"Slicing techniques can be applied effectively to requirements specifications described using the Use Case Map scenario notation".

1.4 Thesis Approach

We plan to solve the problem by designing a static backward slicing algorithm for the UCM language. The inputs for the algorithm are: a) A particular UCM specification file, and b) Any UCM construct that defines/uses data that will serve as a slicing criterion. The output of the algorithm is a valid UCM specification having all the constructs that affect the selected slicing criterion. The algorithm is implemented within the jUCMNav, the UCM framework tool, which is an Eclipse plugin developed using the Eclipse Modelling Framework (EMF), and the Graphical Editing Framework (GEF).

The implementation of the slicing approach on jUCMNav requires full understandability of the source code of the framework, which is over 150 thousands lines of code.

The implementation of the slicing approach consists of the following seven steps:

1. Define the slicing criterion.

2. Develop the backward traversal mechanism.

3. The traversal mechanism should deal with concurrency(AND-fork and AND-join constructs).

4. The traversal algorithm should deal with hierarchy: The slicing algorithm should be able to traverse plug-ins and stubs at different decomposition levels.

5. The traversal algorithm should be able to identify the related/unrelated

variables within responsibilities, start/end points, OR-forks, timers, and waiting-places.

6. Two types of outputs should be supported, marked UCM and a reduced UCM.

1.4.1 Validation of Thesis Approach

Theoretical Validation

We theoretically validated the research hypothesis by implementing the proposed approach as proof of concept on JUCMNav framework. Moreover, we conducted an experiment to study the impact of using the proposed slicing technique on understandability of UCM specifications.

Evaluation

We evaluated our methodology through its application to many specifications of different sizes and exhibiting large and various sets of UCM constructs as well as implementing the approach on three case studies plus a mock-up system. The resulting slices were valid UCM specifications.

1.5 Contributions

This thesis has the following contributions:

1.5.1 Contribution 1: Design of a static slicing approach for UCM

This thesis proposes a static slicing approach for the UCM language. The proposed approach covers all UCM language constructs. The presented slicing approach also provides the following features:

- It handles model hierarchy presented by UCM stubs.
- It solves loops and inconsistencies within UCM.
- It handles concurrent scenarios.
- It supports two types of outputs: an executable and a marked slice.

1.5.2 Contribution 2: Implementation of the slicing algorithm within jUCMNav framework

The proposed slicing approach is implemented within the jUCMNav tool [38] as a proof of concept.

1.5.3 Contribution 3: Empirical Evaluation and Validation

The slicing approach is evaluated using 3 case studies and one mock model. We have shown that the slicing approach can be applied to UCM specifications with different sizes and structures, and it proved its effectiveness as a reduction technique. The approach is validated by conducting an experiment to prove the impact of using UCM slicing on understandability of UCM specifications.

1.5.4 Contribution 4: Publication

journal paper is in preparation and will be submitted to SoSyM.

1.6 Issues not Addressed in this thesis

We applied backward static slicing technique on UCM. The Other slicing techniques such as dynamic, conditioned, chopping etc. [15, 20] are not considered.

1.7 Outline

The thesis as organized as follows:

Chapter 2: presents the background and related work as well as an overview of Use Case Maps language.

Chapter 3: Describes the presented slicing approach in details.

Chapter 4: Describes how the slicing approach works in jUCMNav tool.

Chapter 5: Discuss the empirical evalutaion of the presented approach.

Chapter 6: Discuss the advantages and shortcomings of the approach.

Chapter 7: States the conclusion and future work.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

In this section, we present a brief background of this research. We start by briefly presenting the Use Case Maps language then reviewing existing slicing techniques.

2.1 Background

In the early stages of common development processes, system functionalities are defined in terms of informal requirements and visual descriptions. Although Semi-formal, scenario driven approaches, however, have raised the awareness and use of requirements engineering techniques, mostly because of their intuitive syntax and semantics. The Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard [36], is a high-level visual scenario-based modeling language that has gained momentum in recent years within the software requirements community. Use Case Maps can be used to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a high level of abstraction, and to provide the stakeholders with guidance and reasoning about the system-wide architecture and behavior. UCMs have been successfully used in describing and validating a wide range of systems [39].

UCMs expressed by a simple visual notation allow for an abstract description of scenarios in terms of causal relationships between responsibilities (×, i.e., the steps within a scenario describing operations, functions, tasks, actions, etc.) along paths allocated to a set of components. UCMs help in structuring and integrating scenarios (in a map-like diagram) sequentially, as alternatives (with OR-forks/joins; —</>—), or concurrently (with AND-forks/joins; ⊣⊢/⊣⊢).

One of the strengths of UCMs resides in their ability to bind responsibilities to architectural components. Several kinds ot UCM components allow system entities (▭) to be differentiated from entities of the environment (웃). Components can be organized hierarchically, i.e., vertical decomposition, through the *component containment* mechanism.

UCM scenarios can be integrated together, yet individual scenarios are tractable through scenario definitions based on a simple data model. Scenario definitions make use of path variables and conditions to identify individual scenarios in an integrated collection of UCMs. Conditions allow the explicit definition of otherwise hidden causal dependencies of path segments. A scenario definition may define the desired start points of the scenario, the end points (where the scenario should end), variables' initialization values in the shared data model of the

URN specification, preconditions, and postconditions that have to be met [36].

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. Path details can be hidden in sub-diagrams called plug-in maps, contained in stubs (presented as *diamonds*) on a path. A plug-in map is bound (i.e., connected) to its parent map by binding the in-paths of the stub with start points (●) of the plug-in map and by binding the out-paths of the stub to end points (▮) of the plug-in map. There are four types of stubs: (1) static stub (solid diamond ◇) can have only one plug-in map, (2) dynamic stub (dashed diamond ◇) may contain multiple plugin-in maps, whose selection can be determined at run-time according to a selection policy, (3) synchronizing stub (rendered with the letter S inside the dynamic stub symbol ◇) is a dynamic stub that synchronizes its plug-in maps before the traversal of the UCM path is allowed to continue past the stub, (4) blocking stub (rendered with the letter B in subscript to the symbol of the synchronizing stub ◇) is a synchronizing stub that does not allow its plug-in maps to be visited more than once at the same time. For a complete description of the Use Case Maps language, interested readers are referred to the ITU-T standard [36].

The most comprehensive UCM tool available to date is the Eclipse plug-in *jUCMNav* [38], a full graphical editor and analysis tool for UCM models. *jUCMNav* is developed using the Eclipse Modeling Framework (EMF), and the Graphical Editing Framework (GEF). In this thesis, we propose to implement our slicing feature within the *jUCMNav* framework.

2.2 Literature Review

2.2.1 Program slicing

Program slicing [2] represents an approach to reduce the size of a particular source code by only locating the portions of the program code related to the execution of a targeted output or function. A slice P' is produced from the original program P by eliminating the code statements that do not contribute to the computation of a specific variable V at some location S, called *the slicing criterion.* The produced slice can give answer to the question: "what program statements potentially affect the value of variable V at statement S?". The observer may not be able to differentiate between the execution of the slice and the execution of the original program since the focus is on the value of the variable V in the statement S. An important feature of program slicing is that the resulting slice preserves the semantics of the original program [2].

Consider the source code of Fig. 2.2.1, calculates the sum s and the product p of a set of integer numbers. The program executes until an upper bound n is reached. Let the slicing criterion be (10, p), i.e., our interest is only in the calculation of the product p in line 10). Fig. 2.2.1(b) illustrates the resulting slice.

Many program slicing techniques have been introduced in the literature [40, 15]. In what follows, we briefly introduce them.

(a) Original program	(b) Static slice for (10, p)	(c) Dynamic slice for (10, p, n=0)
1. $read(n)$	1. $read(n)$	1.
2. $i := 1$	2. $i := 1$	2.
3. $s := 0$	3.	3.
4. $p := 1$	4. $p := 1$	4. $p := 1$
5. $while\ (i < n)$	5. $while\ (i < n)$	5.
6. $s := s + i$	6.	6.
7. $p := p * i$	7. $p := p * i$	7.
8. $i := i + 1$	8. $i := i + 1$	8.
9. $write(s)$	9.	9.
10. $write(p)$	10. $write(p)$	10.

Figure 2.1: A simple program and its corresponding static and dynamic slices [1]

Static vs. Dynamic slicing

Program slicing, introduced by Weiser [2], is considered as static since it does not consider any specific input for the target program to be sliced. The produced slice preserves the behavior of the program for all possible inputs. Korel and Laski [41] proposed the notion dynamic slicing, that preserve the behavior of a program not only with respect to the slicing criterion but also with respect to a particular input, i.e., particular runtime execution of the program. Hence, the slicing criterion is extended by a third item, representing the value of the input. The size of a dynamic slice is considerably reduced compared with the one produced by applying static slicing.Fig. 2.2.1(b) illustrates the resulting dynamic slice based on the slicing criterion (10, p, n=0). Only one statement contributes to the computation of the output in statement 10.

The produced slice of 2.2.1(b) can be considered as a static slice since it is independent from inputs of the program and correctly calculates p given any possible execution. On the other hand, if the focus is on the statements having an effect on the slicing criterion for a specific execution only, then a dynamic slice is computed. A third item is then extended to the selected slicing criterion, that is, the program inputs. A dynamic slice (see Fig. 2.2.1(c)) shows the program execution when the input of the variable is n = 0.

Forward slicing vs Backward slicing

Having selected a slicing criterion, forwards or backwards traversal can then be applied on the program starting from that slicing criterion. The result of forward slicing technique is a slice containing code statements of the original program that are affected by the slicing criterion whereas the result of backward slicing technique is a slice containing all code statements of the original program that have an effect on the selected slicing criterion. The original slicing approach by Weiser [2] is a static backward slicing approach. Backward slices can be used to assist developers identify the portions of the program susceptible to contain bugs. Forward slicing techniques may assist maintainers in predicting those portions of the program possibly affected after a maintenance task is performed [10].

Hybrid slicing

Static slicing technique suffers from the imprecision problem while dynamic slicing targets one specific execution only. To help improve the quality and precision of the produced slices, Gupta and Soffa [42] proposed the notion of hybrid slicing. Hybrid slicing aims at incorporating dynamic information into static slices. Takada et al. [43] and Umemori et al. [44] have introduced *Dependence-cache* which is another type of slicing that combines both dynamic and static information.

Quasi-static slicing

Venkatesh [45] introduced the *quasi-static* slicing technique. A quasi-static slice is built based on an initial prefix of a sequence of input to the original program. Quasi-static slice can be used to study the program behavior when some variables of the input have fixed values while others vary. The quasi-static slice is considered to be the same as a static slice if the values of all variables have no fixed values, whereas, it can be considered to be the same as a dynamic slice if the values of all input variables are fixed.

Conditioned slicing

Conditioned program slicing, introduced by Canfora et al. [46], aims to preserve the original program behavior based on a given slicing criterion for a specified group of execution paths. A group of program's initial states characterizing those paths is identified on the input variables as a first order logic formula. Given a program along with its group of initial states, a symbolic executor is initially used by the slicing algorithm in order to reduce the program by excluding the paths that are infeasible according to those initial states. Next, slicing is applied on the reduced version of the program. As the infeasible paths are eliminated, the resulting slice is more precise than those slices that are produced by traditional slicing techniques.

Chopping

Program chopping is a related slicing operation [47, 48]. A chop contains all the points of a program that are influenced between two locations in the program, called chop source and chop target. A chop can answer the following form of questions: which elements of the program can contribute to transfer effects from the known element (chop source) S to the known target (chop target) T?.

Other types of slicing include, relevant slicing, inter and intra procedural slicing, object-oriented slicing, call mark slicing, interface slicing, pre/post condition slicing, amorphous slicing. For a detailed survey on program slicing, the reader is invited to consult [14, 15, 1, 16, 18, 17].

2.2.2 Model-based Slicing

Slicing has been extended to cover other software artefacts [22] such as software architecture [28], formal specification languages [26, 27, 49] and requirements models [25, 23, 24]. Different types of models require different slicing criteria and produces slices with different properties.

Heimdahl et al. [23] proposed a slicing technique for the Requirements State Machine Language (RSML) specification language. Given a targeted scenario, their proposed technique reduces an RSML specification by keeping only behaviors satisfying the operating conditions of the chosen scenario. The result of this reduction is called *specification interpretation*. Then, the resulting interpretation is reduced/sliced further, based on various elements within the target model, in order to identify the specification portions that affect a specific transition or an output variable. This is achieved via analysis of control and data flow information. Finally, the produced slices are combined arbitrarily through a set of standard operations in order to create a combined slice that contains the needed information. Korel et al. [24] proposed a slicing technique for Extended Finite State Machines (EFSM) models that is based on dependencies analysis. Non-deterministic slicing may be applied on the resulting slice so that it may be sliced further through merging transitions and states to build a non-deterministic EFSM. For a detailed survey about State-based Model slicing, the reader is referred to Androutsopoulos et al. [20].

Zhao [28] proposed a new slicing approach to assist architectural reuse and understanding, called *Reuse Architectural Slicing*. He applied this technique to a system's architectural specification that was written in the architectural description language WRIGHT. To generate a slice of the architecture, information flow graph of the architecture is created and traversed. The result is an architectural description that is reduced and contains lines of ADL code only, which may be relevant to a specific slicing criterion. The used slicing criterion used in [28] is either a group of component ports or a group of connector roles. Stafford et al. [50] proposed an approach, called *chaining*, for dependency of software architecture. The chaining technique extracts a dependency chain, called *links*, from specification elements based on the slicing criterion. The slicing criterion of this technique is a collection of component ports.

Kim et al. [51] proposed a dynamic slicing approach for software architectures in order to improve their understandability. The dynamic architecture slice shows the behavior of the selected parts of the architecture at run-time based on a slicing criterion of interest such as a group of events or resources. The approach was illustrated via an E-commerce system and the run-time execution of its architecture. The software architecture was designed using architectural description language(ADL) of choice. However, the main focus was on event-driven ADLs, particularly, RAPIDE language.

Samuel et al. [52] proposed a dynamic slicing technique for UML activity diagrams used in generating test cases. The first step is creating a flow dependency graph from the input activity diagram. Slices are generated for each predicate residing within the edges of activities, and then test cases are generated for each produced slice. Furthermore, Ray et al. [53] described a conditioned-based slicing technique as a method to generate test cases from UML activity diagrams. The flow dependence graph is constructed from a given UML activity diagram and then conditioned slicing is applied based on a predicate node within the graph, in order to generate test cases. The goal is to reduce the number of generated test cases while preserving all the practically useful ones.

Bouras et al. [54] proposed an approach to assist software merging by slicing sequence diagrams. The approach uses slicing to capture all differences and mappings between elements of the sequence diagrams in order to generate a new version of sequence diagram. The slice is generated by transforming the sequence diagram into a Model Flow Graph (MFG) from which dependencies between elements are identified. The slices are generated for each model element in order to identify all differences when map the slice with MFGs of variants of the original sequence diagram. The result is an MFG that conatins all changes from all diagram variants, and then a new sequence diagram is generated based on the resulted MFG.

Lity et al. [55] applied an incremental slicing technique on delta-oriented software product line (SPL) as a mechanism to identify the impact of changes applied to the model and determine the potential retests required for these changes. Delta modeling is a technique used to model various artifacts of SPL such as finite state machines. The proposed technique detects the differences between slices by identifying the new or changed dependencies which facilitates retesting. Best and Rakow [56] presented a slicing technique for Petri nets model. Petri nets are business process model (BPM) used to model work-flow and business processes the the presented slicing techniques aims to reduce their size to smaller nets with less number of states in order to facilitate validation of business processes.

In an early work, Hassine et al. [25] have proposed a UCM-based static slicing algorithm that produces UCM slices given a specific slicing criterion (end/start point or responsibility). Their approach does not cover all UCM constructs (e.g., different types of stubs such as synchronizing stubs). The slicing criteria in [25] are relaxed in [57] by considering any UCM construct or component. Although the feasibility of the resulting slices is investigated, both approaches [25, 57] are static and do not consider the UCM data flow model.

CHAPTER 3

USE CASE MAPS STATIC SLICING APPROACH

Before presenting the details of our proposed UCM slicing approach, we provide the definitions of UCM slicing criterion and UCM slices:

Definition 3.1 (UCM Slicing criterion) *Let U be a UCM Specification. A slicing criterion* SC *for U is defined as a couple (*Target, Var*) where:*

- Target *may be a start point, a responsibility reference(respRef), an OR-Fork branch, a Timer branch, a Waiting-Place branch, or an end point.*
- Var *represent a set of one or many variables defined or used in* Target*.*

In a UCM specification, variables reside in responsibilities (executable source code), OR-Fork branches (Boolean conditions containing variables), timers (normal and timeout paths have Boolean conditions containing variables) start points (preconditions expressions containing variables), and end points (postconditions expressions containing variables). The other UCM constructs, such as OR-Joins, AND-Forks, AND-Joins, etc., cannot be chosen as a slicing criterion because they don't contain variables.

Definition 3.2 (UCM Slice) *Let U be a UCM Specification and SC a given slicing criterion for U. A UCM slice U' is a reduced UCM that is produced from U by keeping only the UCM parts affecting SC.*

3.1 UCM Slicing Algorithm

Figure 3.1, described as a UCM scenario, illustrates the proposed slicing algorithm, and it is mapped into *Algorithm* 3. The slicing algorithm has several parts; it starts in *Algorithm* 3 in which it invokes all other algorithms discussed in the next sections. The input of the slicing feature is a UCM specification file and a slicing criterion.

As shown in Fig. 3.1, the scenario starts by selecting a UCM construct (illustrated by start point *SelectUCMConstruct*), then invoking the UCM slicing feature (represented by the responsibility *InvokeUCMSlicingFeature*). Figure 3.16 illustrates a snapshot of the menu showing the UCM slicing invocation feature. Next, the user is asked to select the slicing criterion (responsibility *ChooseSlicingCriterion*). The variables enclosed within the selected UCM construct, are then extracted and the user can select zero, many, or all listed variables. Once the slicing criterion is chosen, the three responsibilities *BackwardTraversal* (referring to the UCM backward traversal algorithm), *DependenciesComputation* (referring

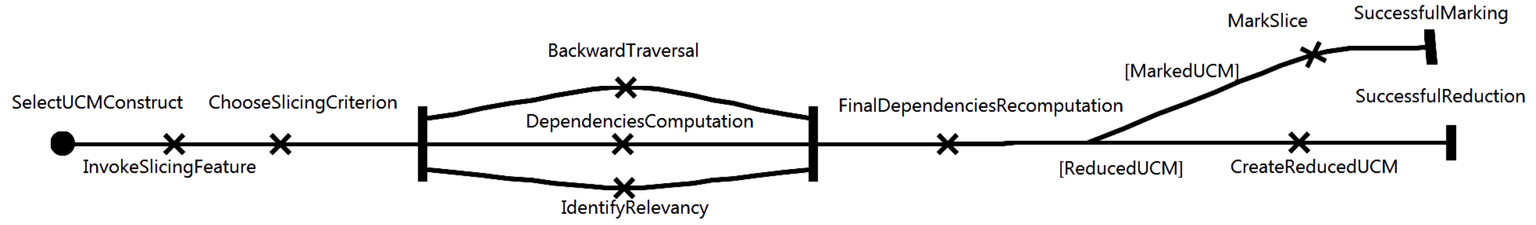


Figure 3.1: UCM Backward Slicing Approach

to the dependencies computation algorithm), and *IdentifyRelevancy* (referring to the identification of retained UCM construct) are performed in parallel (enclosed between an AND-Fork and an AND-Join in Fig.3.1). During the backward traversal, various model elements are visited and data/control flow dependencies are computed based on the selected slicing criterion (see Sect. 3.3). While these dependencies are computed, the set of related/unrelated UCM constructs is being constructed. Once the backward traversal is complete along with the computation of dependencies and the identification of related/unrelated UCM constructs with respect to the slicing criterion, another dependency computation (responsibility *FinalDependenciesRecomputation*) is performed. The aim of this step is to resolve issues related to misidentification of relevant constructs, which is mainly caused by the presence of concurrency between different paths (see Sect. 3.5).

The last step consists of producing the output slice. Using the feature GUI (see Sect. 3.8), the user may choose either a *marked slice* (also known as *closure slice* approach) or a *reduced slice.* A closure slice is displayed by marking the related constructs and paths within the original UCM, while a reduced slice is a new executable UCM obtained after removal of unrelated constructs, paths, components, and scenarios. A discussion about the benefits of each type of output is discussed in Sect. 3.8.In what follows, we detail the aforementioned steps.

Algorithm 3: Slicing Algorithm

```
Input  : UCM model
Output: UCM slice model
stubStack:stack(Stub);
visitedJoins:List(PathNode);
SCNode= select target construct as a SC ;
startingLink= get predecessor link of SCNode;
expression= retrieve code expression from SCNode;
variables= Extract variables from expression;
criterionVariables= get selected variables;
sliceType= get selected slice output type;
invoke backwardTraversal(startingLink, criterionVariables, stubStack,
  visitedJoins) (see Algorithm 6) ;
{Handling Concurrency problem before generating the output slice}
invoke finalComp (see Algorithm 14) ;
if (sliceType == "Remove") then
    {when removal option is chosen}
    invoke RemoveElements() (see Algorithm 16);
else
    {otherwise slice-marking option is selected}
    invoke colorSlice(startingLink) (see Algorithm 17) ;
end
```

3.2 Code Expressions and Extraction of Slicing Criterion Variables

UCM Responsibilities (known as responsibility definitions) define the scenario activities, e.g., actions, operations, or functions to be performed. They can be reused in many places as responsibility references (refereed to them as *RespRef*). The most important attribute of a responsibility definition is the *expression* attribute, which is defined via the URN action language. Expressions define the effect of responsibilities on the global data model of a URN specification. Responsibilities expressions are used to compute data flow dependencies when traversing the UCM model at hand.

RespRef	Expression
R1	x=y;
R2	x=10;
R3	x=y+z;
R4	x=k;
R5	x=j;
R6	j=10;
R7	y=z;
R8	z=1;
R9	z=y+k;
R10	i=10;
R11	k=k+1;
R12	bool_var =false;

R13	if(bool_var)
	bool_var2 =false;

R14	if(a>b)
	{
	x=0;
	m=m-1;
	}
	else
	a=a+1;

R15	x=x+1;
	if(y>z && z<10)
	m=10;
	else
	{
	l=j;
	}

R16	m=10;
	if(z<10)
	{
	m=k;
	}
	else if(l==k)
	k=10;

R17	j=k;
	if(j>=10)
	x=10;
	else
	x=0;
	k=k+1;

R18	y=y+1;
	if(x>y)
	{
	i=j+k;
	if(bool_var)
	a=b;
	}
	else
	x=z;

R19	-
R20	-

Table 3.1: List of RespRefs with their referenced expressions

Expressions are written in the form of C-like code strings. An expression can be one simple statement having an assignment, or it can have multiple statements guarded by predicates in the form of if-else conditions, making the expression look more like a function. A list of responsibility expressions are shown in Table 3.1. These Responsibilities are used throughout thesis work to illustrate different scenarios. Expressions vary from a simple assignment statements such as *R1*, and *R2* (where x and y are Integer variables) to more complex expressions where multiple statements and predicates are used such as *R13* (where *bool_var* and *bool_var2* are boolean variables). The UCM language supports Integer, Boolean, and Enumeration data types.

Once a *RespRef* is selected as a slicing criterion, the expression that corresponds to its responsibility definition is captured as a string, then the comments, if any, are removed. Next, the expression is read and variables are extracted from code statements and predicates. The user can then select the slicing variables from the extracted list.

Figure 3.17 shows a snapshot of the variable extraction GUI, where *R15* is selected as a slicing criterion. The extracted variables are l, j, y, z, m, and x.

3.3 Dependencies Computation

The slicing process consists of a backward traversal of path nodes and tracking dependencies to identify those path nodes within UCM model that may impact the slicing criterion. In our proposed technique, Data and control dependencies are computed directly while traversing the path nodes of the UCM model.

Definition 3.3 (Control flow dependency) *A control flow dependency exists between statements T and $\bar{T}$ if T decides whether $\bar{T}$ is executed or not.*

A control flow dependency is determined by the following path nodes: OR-Forks, Timers, Waiting-places, Start points, and End points. These path nodes affect control dependencies by having conditions that determine whether the subsequent paths are executed or not. In addition, responsibility definitions can also impact the control flow by having conditions within their expressions in the form of if-else conditions, e.g., responsibility *R14* in Table 3.1 has the condition *if(a>b)* that determines whether the statements *x=0*, and *m=m-1* are executed or not. Similarly, a start point can control the execution of all subsequent path nodes by having a *precondition* that determines whether the path is executed or not.

Definition 3.4 (Data flow dependency) *A data dependency exists between statements T and $\bar{T}$, if statement $\bar{T}$ references a variable assigned to or defined in T.*

A data flow dependency is controlled by responsibility definitions via their expressions that impact the global data model when the statements within the expression are executed.

Figures 4 and 5 illustrate the algorithms used to compute the data and control flow dependencies. The algorithm starts by reading and extracting the code within responsibility definitions. Code statements must be read top-down to extract their left- and right- side variables. Since we propose a backward slicing technique, the parsed statements are stacked in a reverse order similar to an ordinary program-based backward slicing approach.

To handle assignments that reside within *if-else* blocks, procedure *Analyze-Condition* (see Fig. 5) is invoked, in order to save the condition(s) to which the statement belongs and to add the predicate variables to the list of relevant variables. It is worth noting that our alogrithm handle infinite number of inner if-else blocks. Table 3.2 shows the stack corresponding to the definition of responsibility *R18*.

Once the statements are in the stack, the left hand side variable of each statement is checked against the criterion variables to determine whether the respRef having that expression is relevant to the slicing criterion or not. The criterion variables are updated accordingly.

Other conditions from path nodes, e.g., OR-Forks, start points, etc., impacting the control flow of their successor model elements, are handled the same way. The variables within those conditions are added to the set of as dependency variables. Finally, based on whether dependencies are detected or not, the predecessor respRef(s) are considered either relevant or irrelevant to the slicing criterion.

3.4 Backward UCM Traversal

Figure. 6 depicts the main steps of the UCM backward traversal algorithm.

The algorithm requires the following parameters:

Algorithm 4: Dependency Computation Algorithm-Part I

```
Input  : Expression of responsibility definition as a string and a list of slicing
         criterion variables
Output: True if the RespRef is relevant, false if it's not
Remove comments from the code string;
$isRelevant = false$;
while Expression string is not empty do
    if (expression starts with an if condition) then
        call AnalayzeCondition algorithm;
    else
        {Otherwise it is a statement}
        statement= tokenize expression and extract the statement ;
        statementVariables_list=extract variables from statement;
        {left side variable is added at index(0) in statementVariables_list }
        if (condition_stack is not empty) then
            {statement within if-else block(s)}
            foreach condition $c \in$ condition_stack do
                extract variable(s) from $c$ and add them to
                  statementVariables_list
            end
        end
        Add statementVariables_list to AllStatements_stack ;
        {statements(formed as a list of variables) are added to this stack
        }
    end
end
{after all statments are stacked bottom-up as variable lists form, they are
  analyzed to compute dependencies}
while AllStatements_stack is not empty do
    statementVariables_list= get the top element in AllStatements_stack ;
    remove the top element of AllStatements_stack ;
    if (statementVariables_list(0) is included in criterionVariables ) then
        {index(0) stores the left side variable of the assignment }
        $isRelevant = true$;
        {Add right side variables and condition variables to the
          criterionVariable list}
        for ($i = 1$, i<size_ of_ statementVariables_list, i=i+1) do
            add statementVariables_list(i) in to criterionVariable;
        end
    end
end
return isRelevant;
```

Algorithm 5: AnalyzeCondition-Dependency Computation Algorithm-Part II

```
Input  : Expression of RespRef as a string, List of Criterion variables
condition= extract condition ;
push condition to condition_stack ;
while condition block is not empty do
    if (expression starts with an if condition) then
        {recursive call}
        call AnalayzeCondition algorithm;
    else
        {Otherwise it is a statement}
        Handle statement similar to 4
    end
end
if there is an else block then
    while else block is not empty do
        if (expression starts with an if condition) then
            {recursive call}
            call AnalayzeCondition algorithm;
        else
            {Otherwise it is a statement}
            Handle statement similar to 4
        end
    end
else
    {if condition has no else block}
    remove the top condition from condition_stack ;
end
```

statement	**left Var**	**Right Var**	**Cond-var**
x=z	x	z	x, y
a=b	a	a	bool_var, x, y
i=j+k	i	j, k	x,y
y=y+1	y	y	-

Table 3.2: Stack of R18 code statements used to compute dependencies

Algorithm 6: Backward Traversal Algorithm

```
Input  : startingLink, criterionVariables, stubStack, visitedJoins
Output: updated criterionVariables
currentLink=startingLink currentNode=getSource(currentLink)
while (currentNode ≠ StartPoint OR stubStack not empty) do
    switch currentNode do
        case RespRef do
            HandleRespRef(currentNode) (see Algorithm 7) ;
        end
        case OrFork do
            HandleOrFork_Timer(currentNode) (see Algorithm 8) ;
        end
        case Timer do
            HandleTimer(currentNode) (see Algorithm 8)
        end
        case WaitingPlace do
            HandleWaitingPlace(currentNode) (see Algorithm 8) ;
        end
        case OrJoin do
            HandleOrJoin(currentNode) (see Algorithm 9) ;
        end
        case StartPoint do
            HandleStartPoint (see Algorithm 10) ;
        end
        case Stub do
            {Any stub type }
            HandleStub (see Algorithm 11) ;
        end
        case AndFork do
            {Handling Concurrency}
            HandleAndFork (see Algorithm 12) ;
        end
        case AndJoin do
            {Handling Concurrency}
            HandleAndJoin (see Algorithm 9) ;
        end
        otherwise do
            {path nodes requiring no action}
            if (VisitedNodes does not contain currentNode) then
                Add(currentNode) to VisitedNodes;
            end
        end
    end
    {continue backward traversal}
    currentLink= getPredecessorLink(currentNode);
    currentNode= getSource(currentLink) ;
end
{Reaching a StartPoint with empty stubStack}
if (VisitedNodes does not contain currentNode) then
    Add(currentNode) to startPoints list ;
    Add(currentNode) to VisitedNodes;
    condition=get StartPoint's Precondition;
    ConditionVariables= getVariables(condition);
    Add(ConditionVariables) to criterionVariables;
end
return criterionVariables;
```

- *startingLink* is the predecessor link of the selected SC. If SC is a respRef or an end point construct, *StartingLink* will be the direct predecessor link of this selected construct. However, when the SC is a branch (a link), such as an OR-Fork branch, *StartingLink* will be the predecessor link of the branch's source, that is, the predecessor link of the OR-Fork, Timer etc.

- The chosen slicing criterion variables.

- *VisitedJoins* is a list of all visited OR-Join path nodes during backward traversal. Each path has its own list, and they are used to detect loops as explained in the following sections.

- *StubStack* represents a stack of stubs that are visited within a particular path during traversal and it is used when traversing plug-in maps, as explained in Sect. 3.4.6.

Since each encountered path node requires a particular procedure, we present each case as a separate algorithm invoked during the backward traversal. Traversal terminates when a *startPoint* is encountered and the *stubStack*, which shows the level of abstraction, is empty.

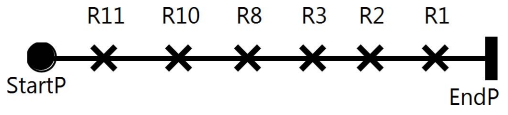
3.4.1 Handling responsibility references

When a respRef is encountered during backward traversal, the expression referenced by this respref is analyzed and data dependency is calculated using the dependency algorithm presented earlier. As a result, the respRef is considered either relevant or irrelevant with respect to the slicing criterion. Algorithm 7 describes the handling of responsibility references.

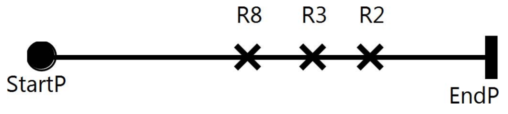
```
Algorithm 7: Handling RespRefs
Input  : A respRef within UCM
Output: True if $respRef$ is relevant to $SC$, false otherwise
if ($respRef \neq SC$) then
    if (respref not in visitedNodes list) then
        Add($respRef$) to $visitedNodes$ list ;
    end
    $respDef$= getDefinition($respRef$);
    $expression$=getExpression($respDef$) ;
    if (expression is not empty) then
        $isRelevant$= dependencyAlg(expression,criterionVariables) Fig. 4;
        if ($isRelevant == true$) then
            Add($respRef$) to $relevantResp$ list ;
        else
            Add($respRef$) to $irrelevantResp$ list ;
        end
    else
        {Otherwise it's an empty respRef}
        Add($respRef$) to $emptyResp$ list ;
    end
else
    {reaching $SC$, means a loop }
    return;
end
```

If a resRef is relevant to the slicing criterion, variables appearing in the right hand side of assignment statements and the ones appearing in the conditions may be added to the set of relevant variables. Figure 3.2 shows a UCM with various responsibility references (Fig. 3.2(a)) and its reduced slice (Fig. 3.2(b)) with respect to the slicing criterion SC= $(R2, x)$.

The referenced expression of *R3*, i.e., x=y+z, impacts the data flow of x, making *R3* relevant to the *SC*. Dependency variables y and z are added to the dependency variables list. Consequently, *R8* is considered relevant since its expression



(a) Scenario with several respRefs



(b) Reduced UCM with SC=$(R2, x)$

Figure 3.2: Handling respRefs

defines the variable z. Responsibilities *R10* and *R11* are considered as irrelevant since they do not impact *R2*. Generating an executable slice also requires the removal of all forward path nodes that come after *SC*, e.g., *R1*.

3.4.2 Handling OR-Forks, waiting places, and timers

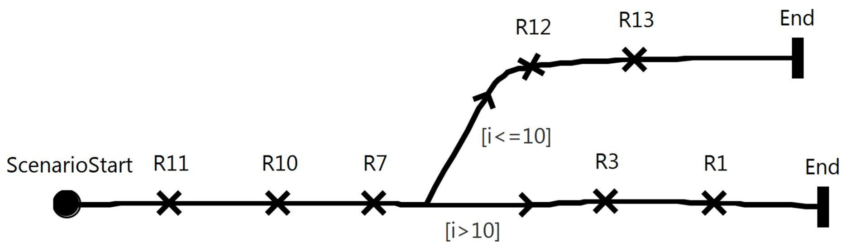
OR-Forks, Timers, and WaitingPlaces are model elements that affect the control flow of their subsequent branches. They may contain conditions that determine whether the subsequent branches will be executed or not. The difference between these constructs is the number of branches attached to them, which serve as alternative flows executed according to the truth and falsity of the condition [36]. An *OR-Fork* has at least two outgoing branches, a *Timer* has two branches at maximum, and a *WaitingPlace* has only one outgoing branch. From dependency computation perspective, the three model elements are handled similarly, as shown in Algorithm 8. The variables enclosed within their conditions are extracted and added to the set of dependency variables. In addition, the algorithm keeps track of the un-traversed OR-Fork branches, so that they can be either deleted if the user opts for a reduced executable slice or unmarked if the user chooses the closure slice approach.

Algorithm 8: Handling Orfork,Timer, and WaitingPlace

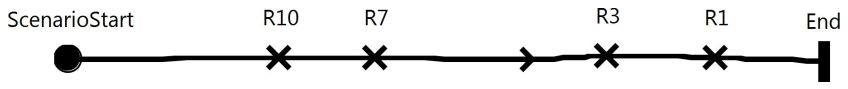
```
Input  : OrFork, Timer, or WaitingPlace within UCM
{Avoid loop when SC is a branch of orfork, timer, or waiting-place construct}
If(currentLink == SC)
return;
condition= getcondition(Input);
if condition $\neq$ empty then
    conditionVariables=getVariables(condition) ;
    Add(conditionVariables) to criterionVariables;
end
if (InputNode not in visitedNodes) then
    {encounter the node for the first time}
    Add(InputNode) to visitedNodes ;
    foreach (link l $\in$ successorLinks(InputNode) do
        Add(l) to unrelatedBranches list;
    end
    {exclude the current link}
    remove(currentLink) from unrelatedBranches list ;
else
    {it's already been visited}
    remove(currentLink) from unrelatedBranches list ;
end
```

Figure 3.3 illustrates an example of handling OR-Forks with respect to the slicing criteria *SC1=(R1, (x,y))* (see Fig. 3.3(b)) and *SC2=(OR-Fork branch(i >10), i)* (see Fig. 3.3(c)). Figure 3.3(a) shows a UCM with an OR-Fork having two outgoing branches.

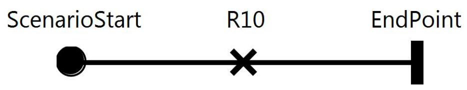
With respect to SC1 and since the branch to which *R1* belongs requires the condition (i>10) to be true in order to be executed, the variable i is added as a dependency variable, and all respRefs that define i are considered as relevant to the slice (i.e., responsibility *R10*). As a result and in addition to *R3* and *R7* (computing x and y), *R10* is kept in computing the slice of Fig. 3.3(b). How-



(a) A UCM scenario with an OR-Fork



(b) Executable UCM slice with SC1=$(R1, (x, y))$



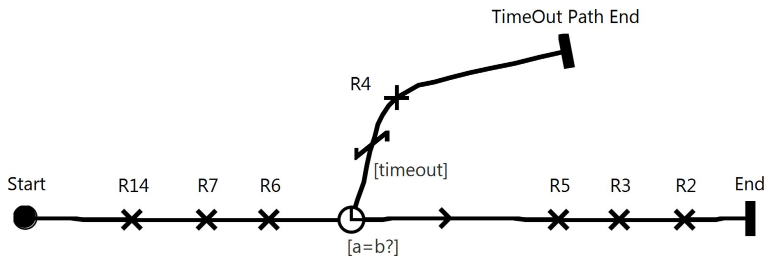
(c) Executable slice with SC2=(Orfork branch(i>10),i)

Figure 3.3: UCM Slicing: Handling OR-Forks

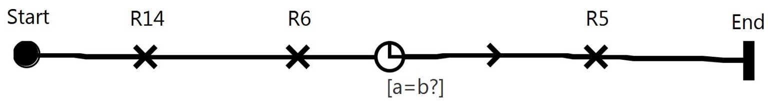
ever, only *R10* is kept in computing the slice of Fig. 3.3(c) since it is the only responsibility updating the value of i.

Likewise, figures 3.4 and 3.5 are examples of scenarios containing a timer and a waitingPlace respectively. In Fig. 3.4, the execution of the timer continuation path, to which the *SC* belongs, depends on the condition *aequalsb*. Similarly, the condition, k>1, determines the execution of the *SC* in Fig. 3.5.

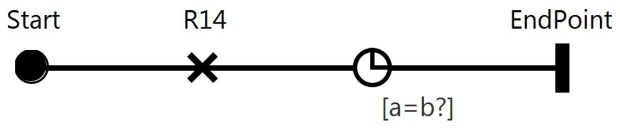
Similar to handling OR-Fork branches as slicing criteria, Fig. 3.4(c) and Fig. 3.5(c) show executable slices when SC are branches of a timer and a waiting-place respectively. Likewise, figure 3.5(d) illustrates the executable slice resulting from applying slicing with the slicing criterion being the end point *End* and the variable i.



(a) A scenario with a timer in UCM

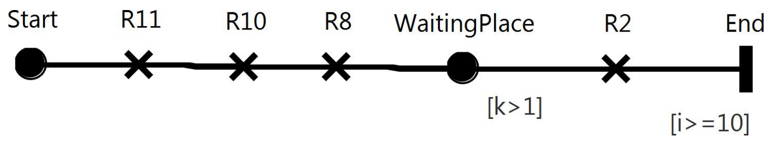


(b) Executable UCM slice with $SC{=}(R5,(x,j))$

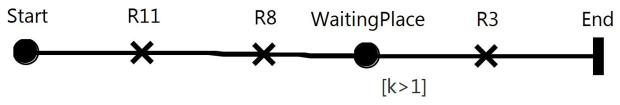


(c) Executable UCM slice with SC=(Branch of a timer $(a = b)$,a)

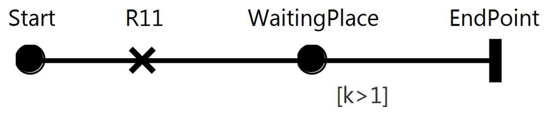
Figure 3.4: UCM Slicing: Handling Timers



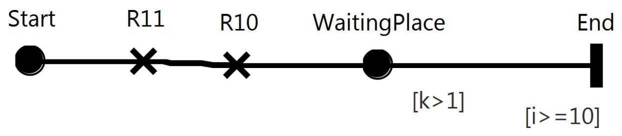
(a) A scenario with a waitingPlace in UCM



(b) Executable UCM slice with $SC{=}(R3,z)$



(c) Executable UCM slice with SC=(Branch of a waiting-place (k>1),k)



(d) Executable UCM slice with SC=(End, i)

Figure 3.5: UCM Slicing: Handling WaitingPlaces

3.4.3 Handling OR-Joins

When an OR-Join is encountered during backward traversal, the incoming predecessor branches are traversed separately. The backward traversal algorithm (see Fig. 6) is invoked recursively for each branch of the OR-Join with the following parameters:

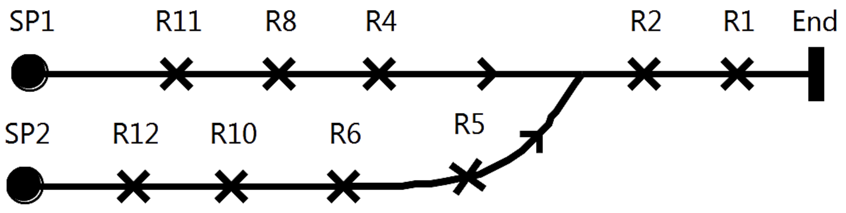
- *startLink* is the branch link.
- *criterionVariables* is the set of dependency variables computed before reaching the OR-Join node.
- *stubStack* is the list of stacked stubs that have been encountered before reaching the OR-Join node.
- *visitedJoins* is the list of the OR-Joins visited before reaching the OR-Join node, used to detect loops within branches.

For each branch, the backward traversal algorithm computes its own dependencies and identifies relevancy based on its local dependency variables. When a branch encounters internal OR-Joins, sub-branches are handled in a recursive manner, forming a tree of child branches. Once a branch traversal is finished, it returns the computed dependency variables, to its parent branch. Finally, all sets of variables emanating from the child branches are merged. Algorithm 9 illustrates the handling of the OR-Join node.

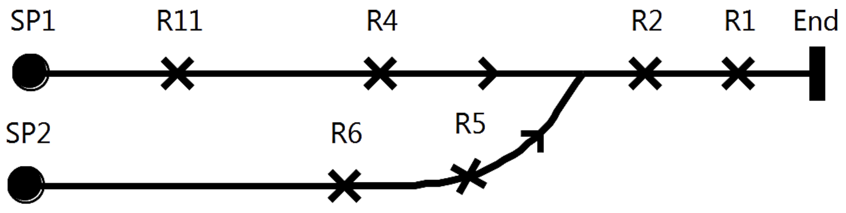
Figure 3.6 describes an example of handling OR-Joins. Responsibilities *R8*, *R10*, and *R12* are irrelevant with respect to the slicing criterion (R2, x).

Algorithm 9: UCM slicing: Handling OR-Joins

```
Input : $OrJoin$
if (visitedjoins Not contain($OrJoin$)) then
    Add($OrJoin$) to $visitedjoins$ ;
    if (visitedNodes not contain ($OrJoin$)) then
        {Add the node to the global list} Add($OrJoin$) to $visitedNodes$ ;
    end
    {get all dependency variables}
    $criVar$= $criterionVariables$;
    $VisJoin$= $visitedjoins$ ;
    $stubs$= $stubStack$ ;
    foreach (link $l \in$ PredeccessorLinks($OrJoin$) do
        $result$= invoke BackwardTraversal($l$,$criVar$, $stubs$, $VisJoin$) ;
        Add($result$) to $criterionVariables$ ;
        return;
    end
else
    {otherwise it's a Loop }
    return;
end
```



(a) A scenario with an OR-Join in UCM

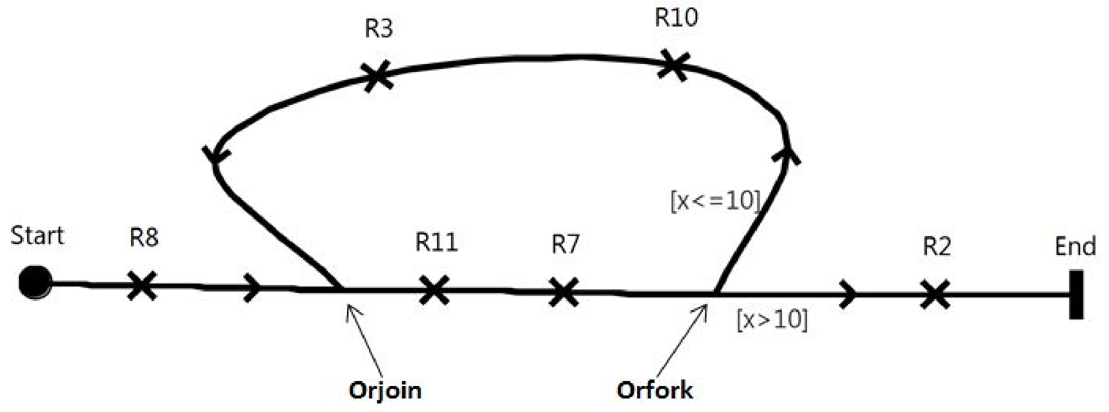


(b) Executable UCM slice with SC=($R2, x$)

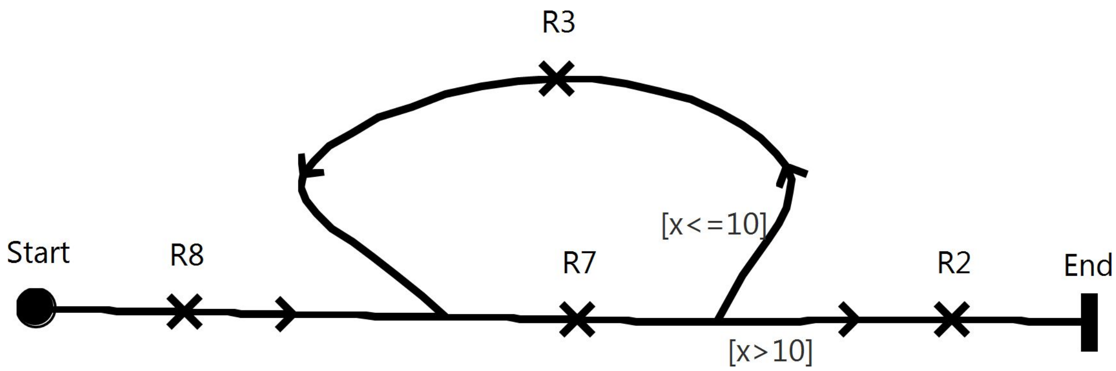
Figure 3.6: Example of handling OR-Joins

3.4.4 Handling loops

Several forms of loops can exist within an UCM such as the loop in the UCM scenario shown in Fig. 3.7 where it is formed by connecting an OR-Join and OR-Fork. The execution of the loop depends on the evaluation of the OR-Fork condition x>10.



(a) A scenario with a loop in UCM



(b) Executable UCM slice with SC=$(R2, x)$

Figure 3.7: UCM Slicing: Handling Loops

The backward traversal algorithm requires detecting these kinds of loops in order to avoid infinite backward traversal of path nodes. The list *visitedjoins*, used as a parameter in main backward traversal algorithm in Fig. 6, is a branch-specific list that is used to keep track of the visited branches. The first time an

OR-Join is encountered, it will be added to the list. The second time the same OR-Join is encountered, the loop will be detected and the algorithm will stop traversing the loop branch again. Checking whether a loop exists or not is shown in the part responsible for handling orjoin in Algorithm 9.

3.4.5 Handling StartPoints

A start point is similar to an OR-Fork, a timer, or a waiting place path nodes with respect to control flow dependency since it may contain a pre-condition that determines the execution of its entire path. Moreover, encountering a startpoint does not mean that we have reached the end of the traversal since the current map might be part of a plug-in map, which requires resuming the traversal of its parent UCM. Algorithm 10 illustrates the handling of start points.

3.4.6 Handling stubs

When a stub is encountered during backward traversal, the *PluginBinding* are used to access the lower-level UCM map(s) connected to this stub. Two types of plugin bindings can be distinguished:(a) *InBinding* which binds the stub's in-path (i.e., a link) with a startPoint on the plug-in UCM map, and (b) *OutBinding* which binds the stub's out-path with an end point on the plug-in UCM map. PluginBinding can bind more than one UCM maps to the same stub (e.g.,, synchronizing stubs [36]), and this requires traversing more than one plug-in UCM map. In addition, the data flow dependency computation is performed on the

Algorithm 10: Handling StartPoint

```
Input   : startPoint within the UCM
condition= getcondition(startPoint);
if condition $\neq$ empty then
    conditionVariables=getVariables(condition) ;
    Add(conditionVariables) to criterionVariables;
end
if (visitedNodes Not contain(startPoint)) then
    Add(startPoint) to visitedNodes ;
end
if (stubStack Not empty) then
    stub=get the top element in stubStack ;
    remove the top element in stubStack ;
    foreach (PluginBinding binding $\in$ getBindings(stub)) do
        {get IN binding}
        foreach (InBinding In $\in$ getIN(binding)) do
            if (startPointOf(In)== startPoint ) then
                {get the link that enters the stub}
                stubEntry=getStubEntryOf(In) ;
                {unrelatedIN stores unrelated branches}
                remove(stubEntry) from unrelatedIN list ;
                criVar= criterionVariables;
                VisJoin= visitedjoins ;
                stubs= stubStack ;
                {resume traversal of parent map} result= invoke
                  BackwardTraversal(stubEntry,criVar,
                stubs, VisJoin) ;
                Add(result) to criterionVariables ;
            end
        end
    end
end
```

plug-in map and it is propagated to the parent map(s). To handle multiple levels of stubs (stubs within plug-ins), *StubStack* is used to store such an hierarchy.

Handling stubs is performed in two main steps: (a) Enter the plug-in maps via *OutBinding* (see Algorithm 11), and (b) Exit the stub when reaching its start point(s) and resume the traversal of stub's parent map using its $InBinding$ (explained in handling start points in Algorithm 10). For both steps, we search for all bindings that match: (a) the reached start points and all stub-entry links bound to it, and (b) the reached out-path links with all stub exit end points bound to it. Two lists, $unrelatedIN$ and $unrelatedOUT$ are used to store those branches that are not traversed. These lists are essential when generating the output slice since they are used to eliminate the unrelated stub branches. In addition, end point path nodes are handled in Algorithm 11 since the only case where end points are encountered is when the traversal reaches a stub and enters the plug-in map via its bound end-point. End points impact control flow, so its post-condition is extracted and the condition variables are added to the $criterionVariables$ list.

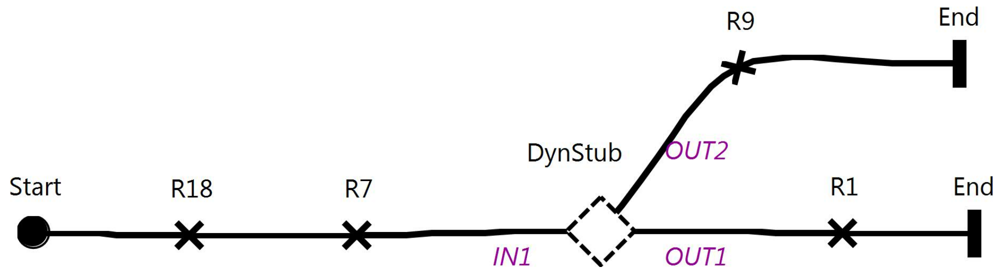
Figure 3.8 illustrates a UCM scenario with stubs. Fig. 3.9 shows the output slice with respect to $SC = (R1, x)$. While computing the slice, the following points are handled during the traversal:

- Traversing more than one plug-in map. The root map (Fig. 3.8(a)) contains a dynamic stub *DynStub* that is bound to two different plug-in maps, Fig. 3.8(b) and Fig. 3.8(c).

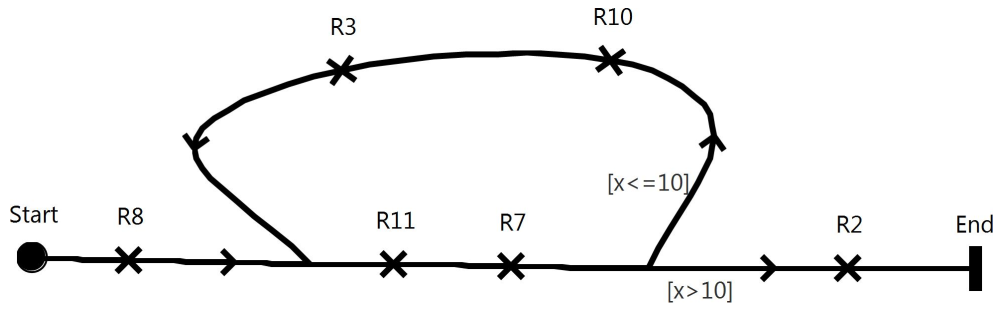
- Handling end-points by extracting their post conditions and add their vari-

Algorithm 11: Handling Stub

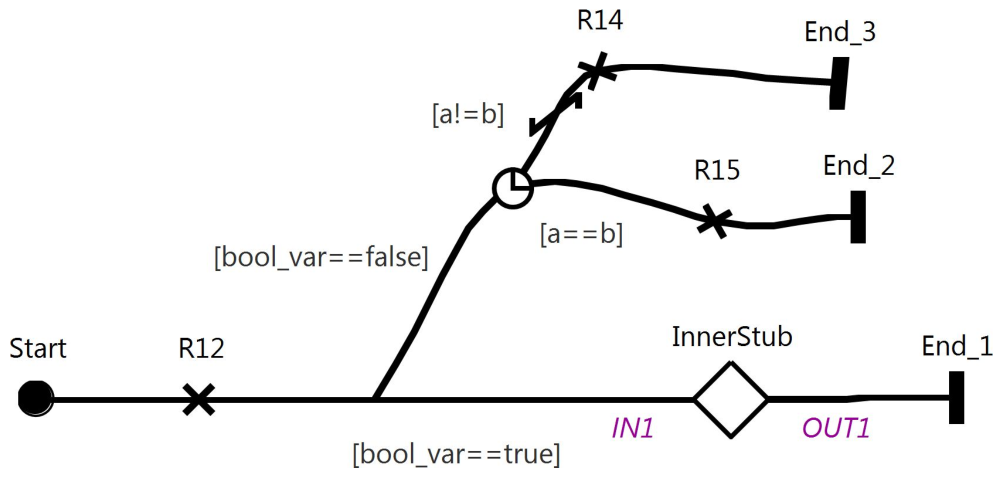
```
Input  : startPoint within UCM
if (visitedNodes Not contain(stub)) then
    Add(stub) to visitedNodes ;
    foreach (link l ∈ successorLinks(stub) do
        Add(l) to unrelatedOUT list;
    end
    foreach (link l ∈ predecessorLinks(stub) do
        Add(l) to unrelatedIN list;
    end
else
    {otherwise it's already been visited} remove(currentLink) from
     unrelatedOUT ;
end
Add(stub) to the top of stubStack ;
{Enter the plug-in map(s)}
foreach (PluginBinding binding ∈ getBindings(stub)) do
    {get OUT bindings}
    foreach (OUTBinding OUT ∈ getOUT(binding)) do
        if (exitLinkOf(OUT)== currentLink) then
            {get the end-Point}
            stubExit=getEndPointOf(OUT) ;
            {Add endPoint to visitedNodes}
            if (visitedNodes not contain stubExit) then
                Add(stubExit) to visitedNodes ;
            end
            {get the condition of the ednPoint}
            condition= getcondition(stubExit);
            if (condition ≠ empty) then
                conditionVariables=getVariables(condition);
                Add(conditionVariables) to criterionVariables;
            end
            link=getPredecessorLinkOf(stubExit);
            criVar= criterionVariables;
            VisJoin= visitedjoins ;
            stubs= stubStack ;
            {start traversal of plug-in map}
            result= invoke BackwardTraversal(link,criVar,stubs, VisJoin) ;
            Add(result) to criterionVariables ;
        end
    end
end
```



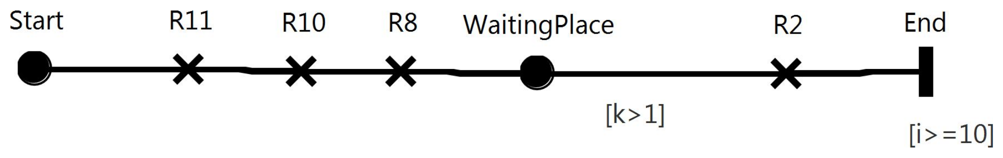
(a) Root map with a dynamic stub in UCM



(b) loop map bound to *DynStub*



(c) Plug-in map bound to *DynStub*

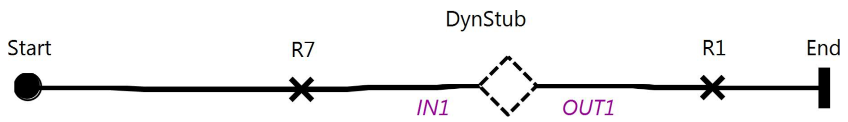


(d) WaitingPlace map bound to *InnerStub*

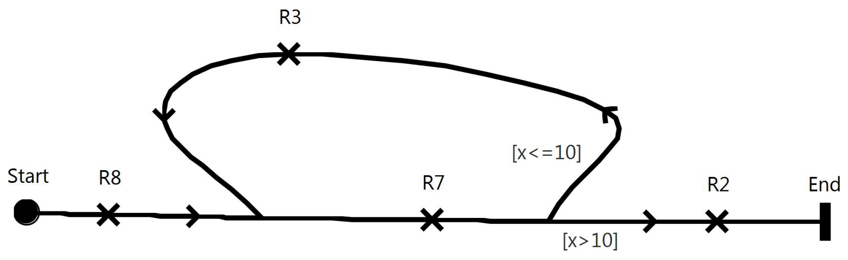
Figure 3.8: A scenario with stubs in UCM

ables as criterion variables (see Fig. 3.9(d) where $R10$ is considered relevant to the slicing criterion since it assigns the variable i, which is used in the condition of the end-point).

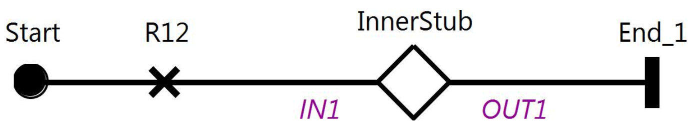
- Handling many levels of abstraction. Fig. 3.8(c) shows a plug-in map, bound to $DynStub$, which contains $InnerStub$ bound to $WaitingPlace$ map (Fig. 3.8(d)). This represents three levels of abstraction, that is handled using $Stubstack$, which is used to stack the stubs during backward traversal and control the levels that are being traversed upwards and downwards.



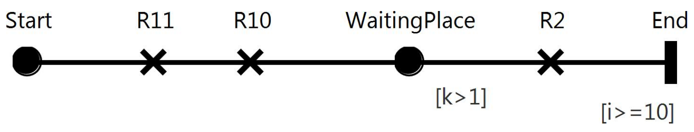
(a) Output slice of root map



(b) Output slice of loop map bound to $DynStub$



(c) Output slice of Plug-in map bound to $DynStub$



(d) Output slice of WaitingPlace map bound to $InnerStub$

Figure 3.9: Output slice of the scenario in Fig. 3.8 with $SC = (R1, x)$ in root map

3.5 Handling Concurrency

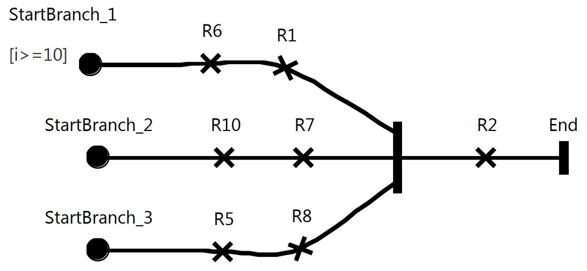
In the previous sections, we have discussed the backward traversal of sequential path nodes. The AND-Fork construct allows many paths to execute concurrently. An AND-Join is a path node that is used to synchronize and merge at least two incoming parallel branches. The execution of concurrent UCM paths may conform to either one of the two following semantic models:

- An interleaving semantics model, i.e., concurrency is reduced to non-determinism, where the behavior of a system that performs two actions a and b concurrently is considered to be the same as the behavior of a system that either does an a followed by b, or a b followed by a. In interleaving semantics model, responsibilities represent atomic actions, not to be decomposable, and their execution is not interruptible. The interleaving variation is based on a single execution thread, where only one single construct can be executing at any given time.

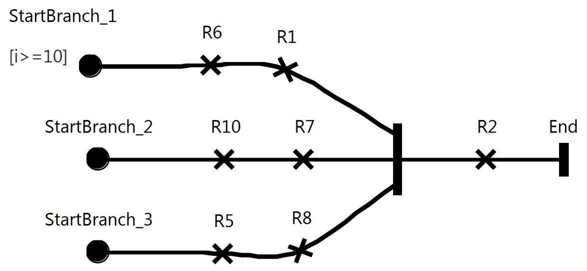
- A true concurrency semantics model, where more than one responsibility can take place at the same time.

It is worth noting that our slicing approach does not dependent on the chosen concurrency semantics.

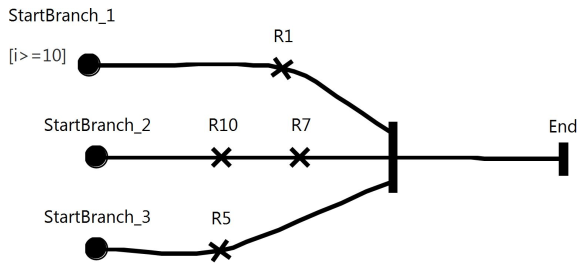
Unlike $OrJoin$ and $OrFork$ branches, concurrent branches are executed independently from each other. However, the order of execution of these branches would have an impact on the UCM global data; hence, it represents a challenge when computing data dependencies. For example, suppose we have the UCM shown in Fig. 3.10(a) and SC=(R2,x). When reaching the AND-Join during backward traversal, the three concurrent incoming branches will be traversed. The order of traversal of these branches can affect the data and control flow dependencies substantially so that each order can have different impact on both, data and control flow. For example, if $StartBranch\_1$ is traversed first , then $StartBranch\_2$, and finally $StartBranch\_3$, *R10* will be relevant to the slice because it assigns variable i, which is used in the condition $(i>=10)$ of $StartBranch\_1$. In addition, *R7* is relevant to the slice because $StartBranch\_1$ is executed before $StartBranch\_2$ and *R1* has the assignment $x = y$; hence y has already been included as a criterion variable before traversing $StartBranch\_2$, and then *R8* is relevant to the slice since $StartBranch\_3$ is traversed after $StartBranch\_2$. However, if the order of traversal is as follows: $StartBranch\_2$, $StartBranch\_1$, $StartBranch\_3$, in this case *R7*, *R8*, and *R10* will not be relevant to the slicing criterion. Since our proposed technique is a static slicing approach, we take a conservative approach by considering all possible executions. Therefore, having SC= (R2,x) will result in a UCM slice with all relevant *resRefs* from all possible orderings such as in Fig. 3.10(b) where all *respRefs* are considered relevant to the SC. SC=(R5,x) in Fig. 3.10(c) shows a different situation, that is, it shows the SC inside an AND-Join concurrency branch. The slicing algorithm moves forward before running the backward traversal, after SC, to check whether SC resides within a concurrency branch and if it does, it catches all the concurrent branches so that backward traversal is applied to them independently. Also, nodes between SC and the AND-Join are removed,e.g., R8, as well as the nodes coming after the AND-Join itself, e.g., R2. This is different from having the SC residing within AND-Fork concurrent branch such as the one in Fig. 3.11 where the algorithm performs backward traversal normally and when the AND-Fork is encountered, it moves forward and catches Branch_2 and Branch_3. Our proposed solution to deal with concurrency is based on three main steps:



(a) Three concurrent paths synchronizing at an AND-Join



(b) UCM slice with respect to $SC = (R2, x)$



(c) SC within AndJoin concurrent branch: UCM slice with respect to $SC = (R5, x)$

Figure 3.10: Example of an AND-Join handling

1. First, construct a tree structure list for each concurrent branch. This tree structure list has three main responsibilities :

    - It stores only the unrelated $respRefs$ corresponding to its path.
    - It stimulates the structure of the branch by having child sub-paths attached to their parent path (i.e., when the branch has an *OR-Join* the incoming paths are children to the path that is past the *OR-Join*).
    - The construction of the tree is performed during backward traversal, when a unrelated $respRef$ is identified, it is added to the tree at the corresponding path/or child path.

2. When a concurrency path node is encountered the concurrent branches are grouped together, and each branch executes the backward traversal algorithm independently using the the set of dependency variables computed before reaching the concurrency node.

3. Once the backward traversal of the concurrent branches is finished, dependencies are computed, and unrelated $respRefs$ are identified and stored in each branch's tree list. Then, a final re-computation of dependencies takes place. This algorithm performs the following:

    - Generates all possible sequences of the concurrent branches.
    - Executes traversal for each sequence using its corresponding tree. No traversal is performed on the UCM itself.

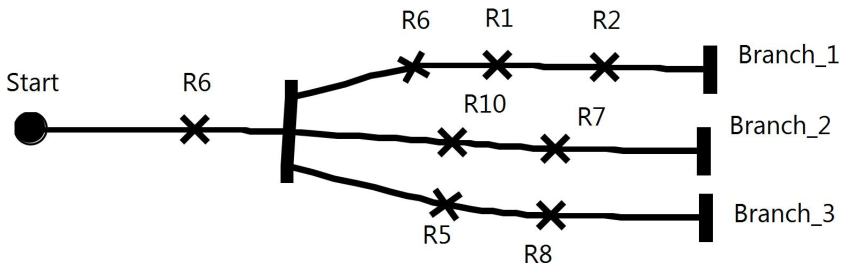
- Dependency re-computation is performed by considering the dependency list (already computed as part of the backward traversal algorithm) of the first branch in the sequence as the initial dependency list. This list is applied on the rest of the branches based on their order in the sequence.
- It only analyzes the unrelated $respRefs$ for each branch's tree list.
- Once a $respRef$ is considered relevant, it is removed from the unrelated list, and it is not re-analyzed any more.
- The process is performed for each possible execution order covering all possible sequences.

The algorithm shown in Algorithm 12 is invoked when an *AND-Fork* is encountered. *GetEndLink(l)* is used to move forward starting from the l as a parameter. This is needed since all branches of *AND-Fork* must be included in the backward traversal.

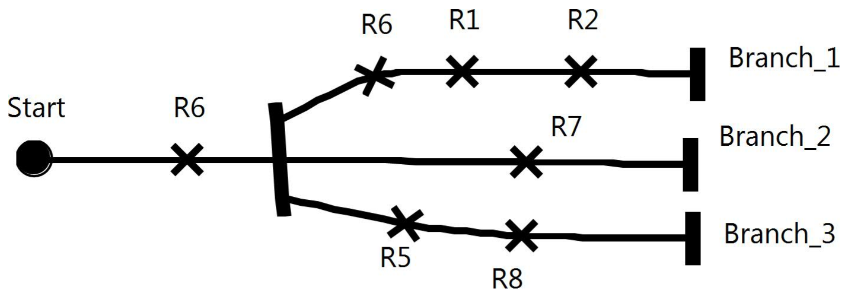
Suppose that the slicing criterion is $SC = (R2, x)$, which resides within the first *Branch_1*, as shown in Fig. 3.11. When the *AND-Fork* is encountered, *getEndLink(l)* is invoked to traverse *Branch_2*, and *Branch_3* in a forward manner to reach the links of their end-points. Once these links are caught, the slicing algorithm is executed for *Branch_2*, and *Branch_3* independently with the caught links as *startingLink* parameters, and initial *criterionVariables*={x}. The $criterionVariables$ of *Branch_1* when *AND-Fork* is encountered={x,y}. However, since each branch is independent, the slicing algorithm, executed for

Algorithm 12: Handling AND-fork

```
Input   : AND-Fork within UCM
if (visitedNodes not contain(AndFork)) then
    Add(AndFork) to visitedNodes list ;
    foreach (Link(l) ∈ SuccessorLinksOf(AndFork)) do
        if (l ≠ currentLink ) then
            endLink=TraverseForward and getEndLink(l) ;
            Add(endLink) to forwardLinks list ;
        end
    end
    criVar= criterionVariables;
    VisJoin= visitedjoins ;
    stubs= stubStack ;
    foreach (Link(l) ∈ forwardLinks) do
        slice=create new instance of slicingAlgorithm ;
        slice(invoke BackwardTraversal(l,criVar, stubs,VisJoin) ;
        Add(slice) to group list ;
    end
    Add(group) to Groups;
end
```



(a) A scenario with *AndFork* in UCM



(b) Output slice with $SC = (R2, x)$

Figure 3.11: UCM Slicing: Handling $AND - Fork$

Branch_2 and *Branch_3*, will start with the initial *criterionVariables*={x}. The variable *slice* is an instance that contains all data including the unrelated $repRefs$ tree, and the *criterionVariables* which consists of the initial variable x and all dependency variables computed when backward traversal is performed on the corresponding branch. For example, unrelated $respRefs$ of $Branch_1$= $(R6, R6)$, and its $criterionVariables = \{x, y\}$. Unrelated $respRefs$ of $Branch_2$= $(R7, R10, R6)$, and its $criterionVariables = \{x\}$. Finally, Unrelated $respRefs$ of $Branch_3$= $(R8)$, and its $criterionVariables = \{x, j\}$. All the concurrent branches grouped in one list in order to execute final re-computation of dependencies. This part is discussed in Sect. 3.6 where it explains how dependency re-computation of all possible sequences is performed.

Similarly, handling *AND-Joins* is shown in Fig. 3.10 with $SC = (R5, x)$. In this example, the selected slicing criterion $R5$ resides within a concurrent branch_3. Therefore, the two branches are also included in the slicing process. The proposed technique always moves forward starting from the selected $respRef$ before executing the backward traversal. If an $AndJoin$ is encountered, the other branches are also included in the slicing process.

3.6 Final Dependency Re-computation

In order to solve the dependency issue, explained in the previous section, caused by the concurrency, we add a dependency re-computation step.

Algorithms 14 and 15 are used to recompute all possible sequences. The idea

Algorithm 13: Handling AND-Join

```
Input   : AND-Join within UCM
if (visitedNodes not contain(AndJoin)) then
    Add(AndJoin) to visitedNodes list ;
    criVar= criterionVariables;
    VisJoin= visitedjoins ;
    stubs= stubStack ;
    foreach (Link(l) ∈ PredecessorLinksOf(AndJoin)) do
        slice=create new instance of slicingAlgorithm ;
        slice(invoke BackwardTraversal(l,criVar, stubs,VisJoin) ;
        Add(slice) to group list ;
    end
    Add(group) to Groups;
else
    {Otherwise it's a loop}
    return;
end
```

Algorithm 14: Final Dependency Computation-Get Sequences

```
Input   : Groups of concurrent branches
for (List (gr) ∈ Groups list ) do
    foreach (combination (sequence) ∈ getAllCombination(gr)) do
        branchSlice= get First branch in sequence(sequence) ;
        sequenceVariables= get criterion variablesOf(branchSlice) ;
        foreach (Element (branch)∈ sequence) do
            branchTree= get the unrelatedRespRef of(branch);
            {call Algorithm 15}
            sequenceVariables= invoke
              recompute(branchTree,sequenceVariables);
        end
    end
end
```

Algorithm 15: Final Dependency Computation-Recompute Branch

```
Input  : $branchTree$, $criterionVariables$
Output: $criterionVariables$
$root$=getRootOf($branchTree$) ;
$unrelatedResp$= getUnrelatedListOf($root$);
foreach ($Element$ ($respref$) $\in$ $unrelatedResp$ ) do
    $respDef$= getDefinition($respRef$);
    $expression$=getExpression($respDef$) ;
    $isRelevant$= dependencyAlg(expression,$criterionVariables$) Fig. 4;
    if ($isRelevant == true$) then
        Add($respRef$) to $relevantResp$ list ;
        Remove($respRef$) from $irrelevantResp$ list;
        Add the new dependency variables to $criterionVariables$ list;
    end
end
foreach $Element$ $childBranch$ $\in$ $getChilrenOf(root)$ do
    $criterionVariables$= invoke recompute($childBranch$,$criterionVariables$);
end
return $criterionVariables$;
```

is to take one sequence at a time, fetch the *criterionVariables* list of the first branch in the sequence, and apply it as initial dependency variables against all other branches sequentially.

The tree of each branch contains a list of all irrelevant *respRef* related to this particular branch. Dependency Algorithm 4 is invoked to calculate relevance against the variables passed to it. If the *respRef* found to be relevant, it is added to the *relatedResp* list and removed from *unrelatedResp* list. The process continues until all possible sequences are computed. The presented solution of concurrency is effective as it covers all possible situations. However, it suffers from overhead computation. We tried to alleviate this limitation by (a) constructing a tree structure in which irrelevant *respRef* are stored instead of re-traversing the UCM concurrent branches, (b) Only the unrelated *respRefs* are re-computed (the related ones are excluded), and (c) Once a *respRef* is found relevant, it is removed from the tree list of the branch to which it belongs so that it will not be re-computed again, and this can reduce the number of re-computations.

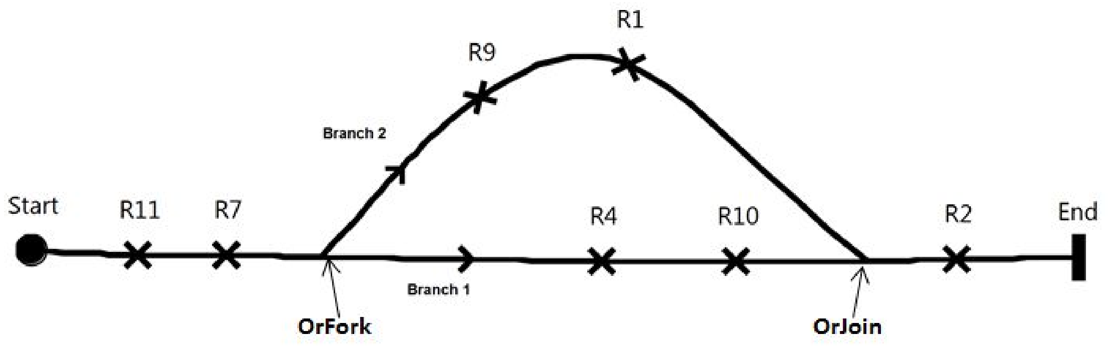
Furthermore, the dependency re-computation helps solve inconsistencies in scenarios where branches of an *OR-Fork* are merged to an *OR-Join* forming a circle as shown in Fig. 3.12(a). The inconsistency issue happens when executing UCM slicing with $SC = (R2, x)$. The set of dependency variables of $branch\_1$, $branch\_2$ are $\{x, k\}$, $\{x, y, z\}$ respectively. $R7$ is considered irrelevant to $branch\_1$, while it is relevant to $branch\_2$. Similarly, $R11$ is relevant to $branch\_1$, but it is irrelevant to $branch\_2$. This makes the *respRefs* $R7$ and $R11$ stored in both lists, the relevant *repsRefs* as well as the irrelevant *respRefs* list making them inconsistent. Before generating the output slice, this inconsistency is solved by comparing the two lists and removing any *respRef* that is stored as relevant from the *irrelevant* list.

3.7 Slice generation

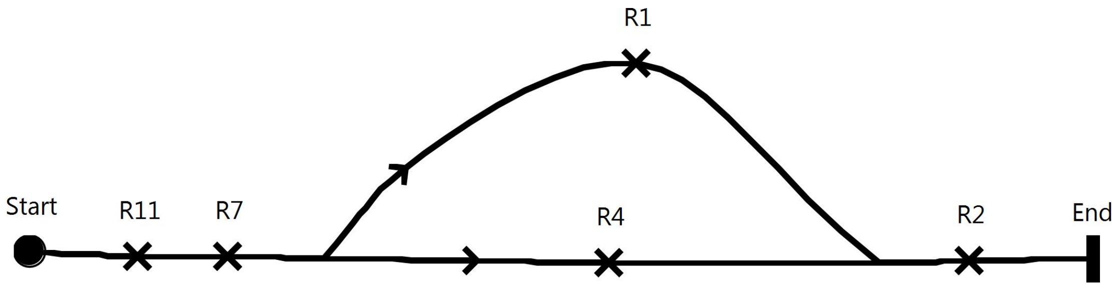
Our proposed slicing technique produces either a reduced slice or a closure slice. In this section, we describe how the final slice is generated.

3.7.1 Executable Slice

The UCMs used in previous sections are reduced slices, where all SC unrelated parts are removed and the output is shown a new valid UCM specification. Pro-



(a) A scenario with $ORFork - ORJoin$ circle in UCM



(b) Output slice with $SC = (R2, x)$

Figure 3.12: Handling Inconsistent $respRefs$ in UCM slicing

ducing an executable slice is more complex that producing closure slice since it is subject to the following challenges:

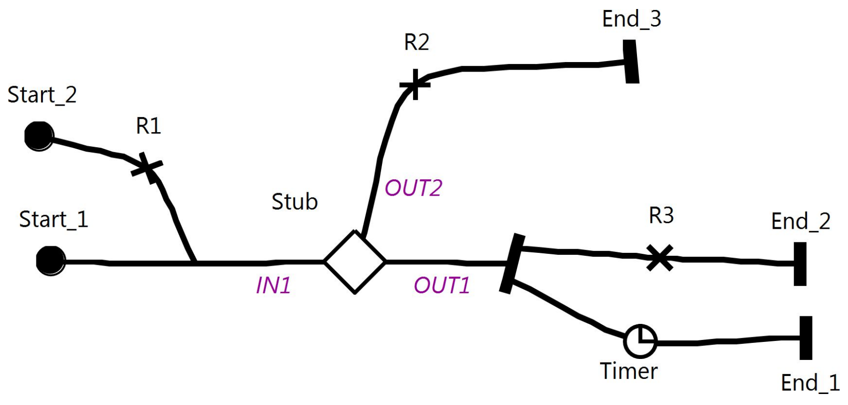
- Preserving graph connectivity and generating a valid UCM specification. The removal of unrelated $repRefs$ as well as un-traversed path branches must not cause the map elements to be disconnected.

- The jUCMNav [38] framework does not provide complete path removal when removing path nodes having subsequent child nodes/paths, e.g., removing *OR-Forks*, *OR-Joins*, and *stubs*. Instead, model transformations are automatically created. This is a challenging task since in most cases some transformations involve the creation of new separated paths that are difficult to access. Figure 3.13 illustrates few examples.

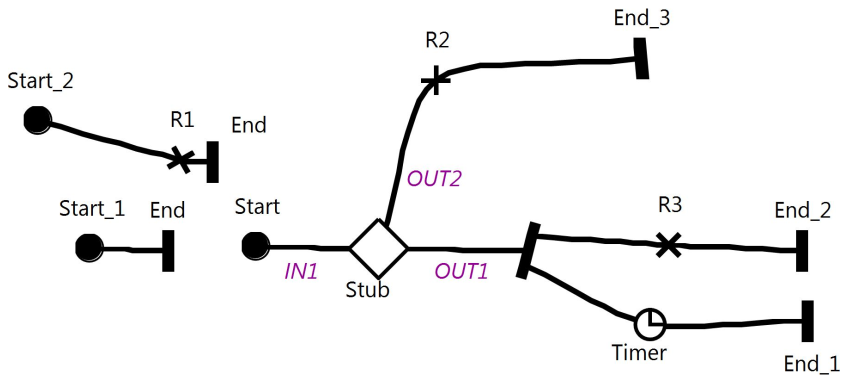
- Backward slicing considers what comes after the slicing criterion as irrelevant. Therefore, all model elements past the target, i.e., $respref$, must be removed when generating the output slice. However, in some scenarios, the slicing criterion resides within a loop, which means that the path nodes past the slicing criterion might be relevant to the output slice. Moreover, the loop structure must be preserved and not disconnected such as the one in Fig. 3.7, where model elements are removed and the loop structure is not disconnected. Similarly, when the slicing criterion resides within a concurrent branch that is merged to an $AND - Join$, the elements between SC and the $AND - Join$ path nodes must be removed while preserving the concurrent branch connected to its $AND - Join$ and then remove all model elements past the $AND - Join$ node such as the one in Fig. 3.10.

Algorithm 16 handles the removal of unrelated elements while keeping map structure and graph connectivity. The lists of all path modes that should be removed are passed as input to the algorithm. These lists are collected during backward traversal, as shown in the aforementioned algorithms handling different kinds of path nodes, e.g., $visitedNodes$ and $unrelatedOrForkBranches$ lists.

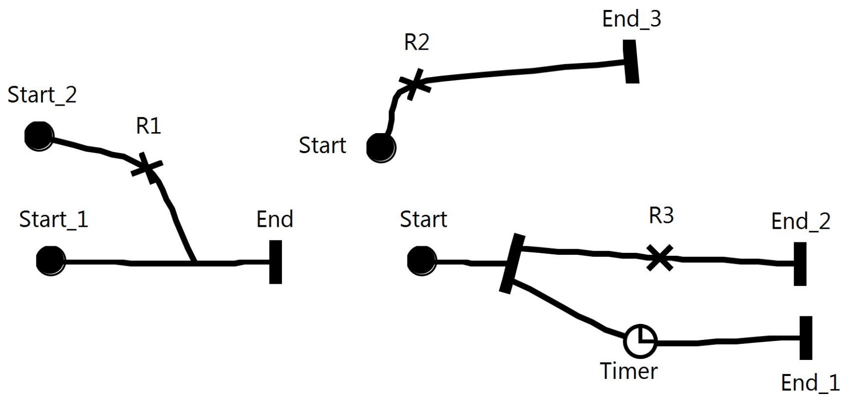
The list $visitedNodes$ is used to exit a loop, if any, and then remove all path nodes past the slicing criterion. To solve the issue of removing path nodes having subsequent child paths/nodes, shown in Fig. 3.13, we traverse the unrelated branches forward and backward to collect each path node and then remove it from UCM. The removal algorithm performs the following actions when generating the



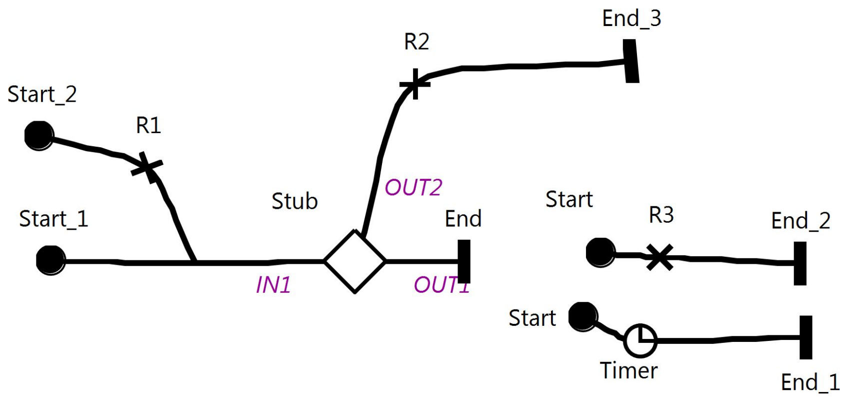
(a) A scenario in UCM



(b) Impact of removing an *OrJoin*



(c) Impact of removing a *Stub*



(d) Impact of removing an *AndFork*

Figure 3.13: Impact of removing path nodes that have consequent child nodes/branches

output slice:

- Removes the unrelated respRefs stored in *unrelatedRespRef* list from UCM model.
- Removes unrelated *stubs'IN* branches stored in *unrelatedStubINs* list along with their contained path nodes.
- Removes unrelated *stubs'OUT* branches stored in *unrelatedStubOUTs* list along with their contained path nodes.
- Removes unrelated *OR-Fork* branches stored in *unrelatedOrforkBranches* list along with their contained path nodes.
- Removes unrelated *Timer* branches stored in *unrelatedTimerBranches* list along with their contained path nodes.
- Removes un-traversed maps from *URN*.
- Removes unrelated scenario start-Points from UCM scenario definitions.
- Removes unrelated Component references.

3.7.2 Closure Slice

Closure slice is an easier alternative to the reduced slice, since it only marks the related as well as the unrelated parts with respect to SC within the original UCM without having to remove or make changes on the model. The related $respRefs$

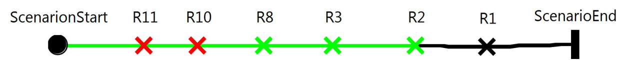
Algorithm 16: Remove unrelated model elements

```
Input  : $visitedNodes$,$unrelatedRespRefs$ $unrelatedOrforkBranches$,$unrelatedStubINs$,
         $unrelatedStubOUTs$,$UnrelatedTimerBranches$
Output: $UCMslice$
foreach (Element ($respRef$) ∈ $unrelatedRespRefs$) do
    remove($respRef$) from UCM ;
end
if $SC$ resides within a concurrent $OrJoin$ branch then
    remove all path nodes between $SC$ and $AndJoin$ node;
    remove all path nodes past the $AndJoin$ node;
end
{check if criterion resides within a loop}
$currentLink$=getSuccessorLinkOf($SC$);
$currentNode$=getTargetOf($currentLink$);
{Move Forward}
while ($currentNode$ is included in $visitedNodes$) do
    $currentLink$=getSuccessorLinkOf($currentNode$);
    $currentNode$=getTargetOf($currentLink$);
end
{reached the first link out of loop}
$unrelatedNodes$=move forward and collect all nodes starting from link($currentLink$);
remove all path nodes in $unrelatedNodes$ list from UCM ;
{remove unrelated $OrFork$ branches}
foreach (Element ($branchLink$) ∈ $unrelatedOrforkBranches$) do
    $unrelatedNodes$=move forward and collect all nodes starting from Link($branchLink$);
    remove all path nodes in($unrelatedNodes$) list from UCM model ;
end
{remove unrelated Stub IN branches}
foreach (Element ($StubINLink$) ∈ $unrelatedStubINs$) do
    $unrelatedNodes$=move forward and collect all nodes starting from Link($StubINLink$);
    remove all path nodes in($unrelatedNodes$) list from UCM model ;
end
{remove unrelated Stub OUT branches}
foreach (Element ($StubOUTLink$) ∈ $unrelatedStubOUTs$) do
    $unrelatedNodes$=move forward and collect all nodes starting from Link($StubOUTLink$);
    remove all path nodes in($unrelatedNodes$) list from UCM model ;
end
{remove unrelated Timer branches}
foreach (Element ($TimerLink$) ∈ $UnrelatedTimerBranches$) do
    $unrelatedNodes$=move forward and collect all nodes starting from Link($TimerLink$);
    remove all path nodes in($unrelatedNodes$) list from UCM model ;
end
{remove unrelated maps}
foreach (Element (map) ∈ getAllMaps($URN - Def$)) do
    if (map is not in $traversedMaps$ list) then
        remove($map$) from $URN - Def$;
    end
end
{remove unrelated scenario start-points}
foreach (Element ($scenarioStartPoint$) ∈ $UnrelatedScenarioStartPoint$) do
    remove($scenarioStartPoint$) from its UCM scenario definition;
end
remove unrelated component-references ;
```

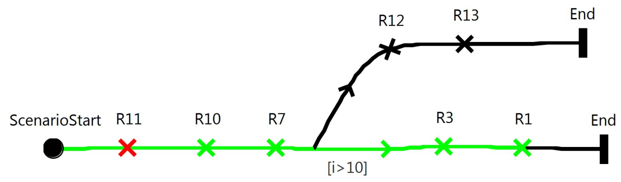
are marked with green color, while the unrelated ones are marked with red color. The constructs that do not contain any code are marked with gray color. The entire paths leading to the SC are marked with green color to show the output slice entirely. Algorithm 17 shows how a closure slice is generated, and examples of marked slices generated from the previous scenarios are illustrated in figures 3.14 and 3.15. As shown in Fig. 3.14(d), the marking of the concurrent branch to which SC belongs starts from SC since the path nodes past SC are not part of the output slice.

Algorithm 17: Slice Marking

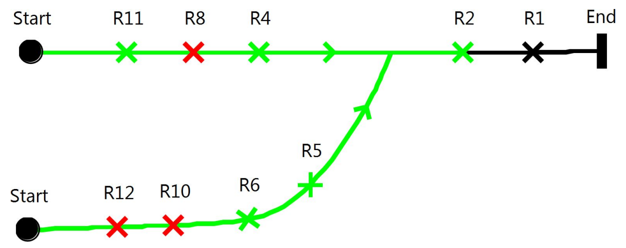
```
Input  : relevantResp:List(respRef),irrelevantResp:List(respRef),
         emptyResp:List(respRef),visitedNodes:List(PathNode)
Output: A UCM marked slice
predLinks:List(link);
currentNode:PathNode;
foreach (PathNode node ∈ visitedNodes) do
    if (relevantResp contains(node)) then
        mark node(green);
    else
        if (irrelevantResp contains(node)) then
            mark node(red);
        else
            if (emptyResp contains(node)) then
                mark node(grey);
            else
                {Otherwise it is not a repRef node}
                mark node(green);
            end
        end
    end
    {mark predecessor links with green color}
    predLinks= get predecessor links of node;
    foreach (link l ∈ predLinks) do
        mark l(green);
    end
end
```



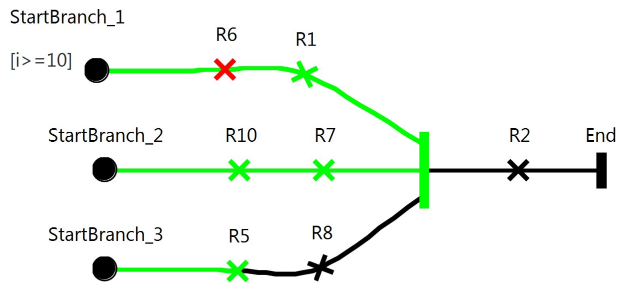
(a) Closure slice of Fig. 3.2 with $SC = (R2, x)$



(b) Closure slice of Fig. 3.3with $SC = (R1, (x, y))$

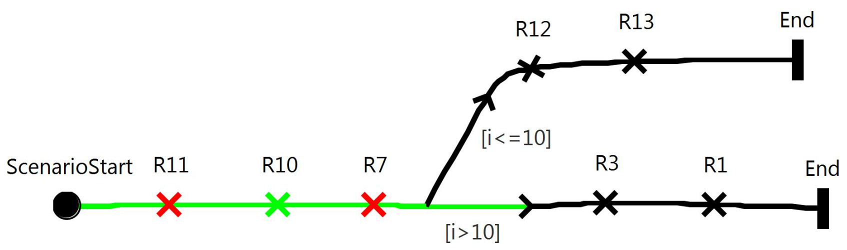


(c) Closure slice of Fig. 3.6 with $SC = (R2, x)$

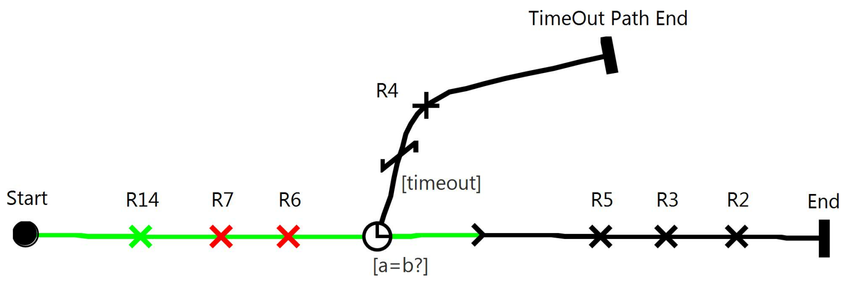


(d) Closure slice of Fig. 3.10(a) with $SC = (R5, x)$

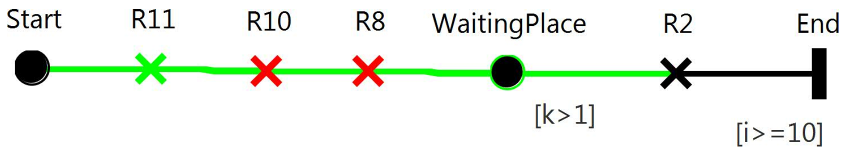
Figure 3.14: UCM slicing using *closure slice* approach



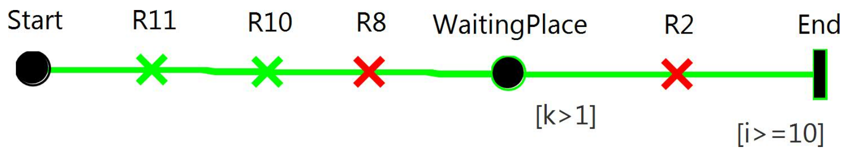
(a) closure slice of scenario Fig. 3.3 (a) with SC=(Orfork branch(i>10),i).



(b) closure slice of scenario Fig. 3.4 (a) with SC=(Branch of a timer $(a = b)$,a).



(c) closure slice of scenario Fig. 3.5 (a) with SC=(Branch of a waiting-place (k>1),k).



(d) closure slice of scenario Fig, 3.5 (a) with SC=(End ,i).

Figure 3.15: *Closure Slice* approach with different UCM constructs as SC

3.8 Tool Support

Our proposed UCM slicing approach is implemented within the jUCMNav framework [38]. To exercise this feature, the user starts by selecting a UCM construct within the set of supported constructs (see Definition 3.1), then right-clicks and chooses the *Static Slicing* command from the menu, as shown in Fig. 3.16. The slicing command is only available for the supported constructs. The expressions enclosed within the construct are then read and all its variables are extracted. Figure 3.17 illustrates the slicing criterion selection GUI. The user may choose zero or many variables to constitute the slicing criterion. When the user does not select any variable the slicing algorithm will generate a slice without computing data flow dependencies. The resulting slice is produced by eliminating all nodes past the chosen slicing criterion, unrelated respRefs within the slice path, and the un-traversed branches.

In addition to the choice of variables, the user can choose the type of the computed slice. The tool offers two output options, namely, *Marked Slice* and *Reduced Slice* (the user is asked to chose the new UCM file name). One advantage of producing a reduced UCM slice is that the slice can be saved in a different file, allowing for further reduction and producing smaller slices. Applying repetitive UCM slicing on already stored slices, known as *incremental model slicing*, may be used to determine the potential required regression tests [58, 55]. The closure slice approach marks the SC-related UCM constructs within the original model, which is a temporary coloring and cannot be saved as a separate UCM specification

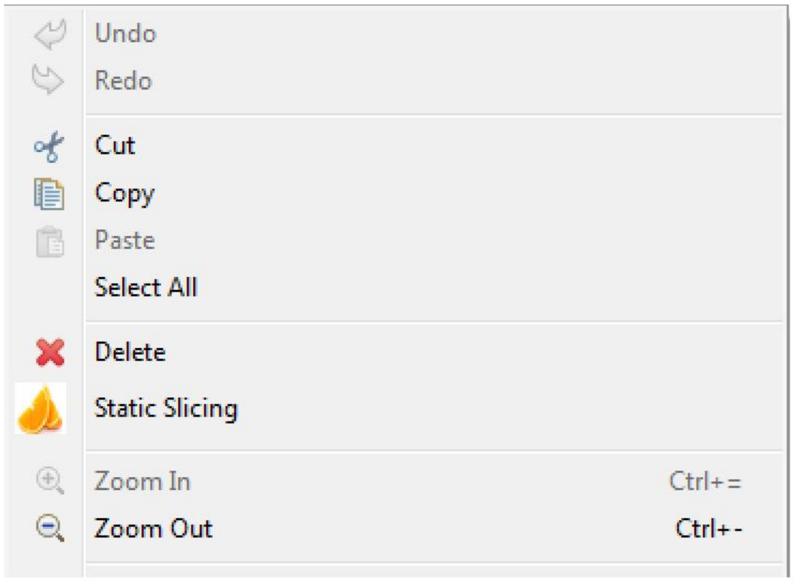
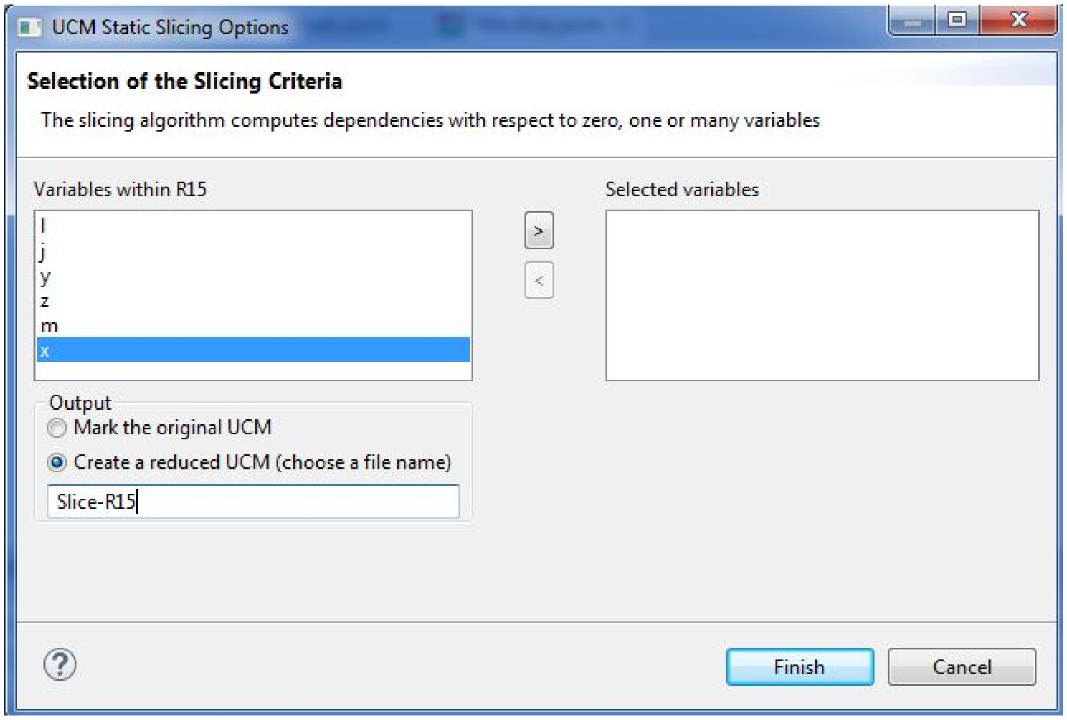


Figure 3.16: UCM slicing command included in command menu of jUCMNav framework


UCM Static Slicing Options

Selection of the Slicing Criteria

The slicing algorithm computes dependencies with respect to zero, one or many variables

Variables within R15: I, j, y, z, m, x

Selected variables:

Output
- [ ] Mark the original UCM
- [✓] Create a reduced UCM (choose a file name)

Slice-R15

Figure 3.17: Slicing Options window, jUCMNav framework

file. However, this approach allows the maintainer to visualize the affected paths and constructs within the original specification. Hence, help him choose different slicing criteria and observe what parts get colored in green color (retained constructs), gray (constructs having no code), and red color (discarded constructs). Using the reduced slice option, the maintainer cannot identify which parts are removed unless he matches the slice with the original specification, which can be a time consuming and error prone activity in large and complex UCM specifications.

CHAPTER 4

EVALUATION AND VALIDATION OF THE SLICING APPROACH

4.1 Empirical Evaluation

In this section, we evaluate our proposed UCM slicing approach using a mock example and three publicly available UCM case studies of different sizes and complexity. Table 4.1 provides some informations about the used case studies in terms of:

- The number of UCM maps, representing root maps and plugins.
- The total number of nodes.

Model	**Number of maps**	**Number of nodes**
Mock	5	86
Discharging Management system	43	257
Ordering System	4	82
Adverse Event Management System	2	43

Table 4.1: Case studies characteristics

4.1.1 Mock System

In this section a mock example is presented to evaluate our proposed static slicing approach. The need to create this mock system was prompted by the necessity of slicing a UCM that contains the entire UCM notational constructs. Such a requirement was not satisfied by any system available online or in the literature. The designed UCM specification has the following features:

- It contains all UCM constructs.
- It contains scenarios with loops and concurrency. Such scenarios require special care when computing dependencies.
- It contains an *OR-Fork OR-Join* circles that might cause the aforementioned inconsistency case.
- It contains hierarchical map structures with the presence of many levels of stubs and plugins.
- The mock example is designed to show the accuracy of the proposed slicing

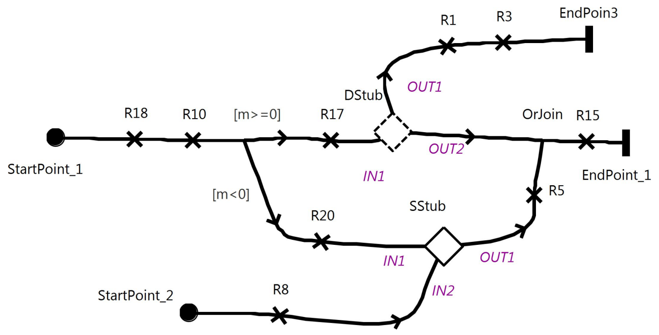
algorithm with respect to dependency computation when handling hierarchy as well as concurrency.

It is worth noting that from a slicing perspective, synchronizing and blocking stub constructs are similar to the dynamic stub construct with respect to the number of plug-in maps allowed. In this mock example, we use dynamic stubs only. All complexity facets that have been discussed are put together within the mock UCM specification in order to check the validity of the proposed slicing algorithm.

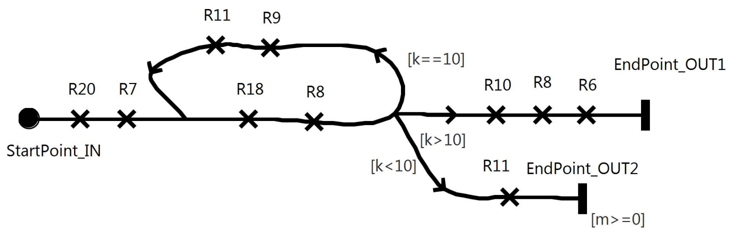
Stub	**Plug-in Map**	**IN Bindings**	**OUT Bindings**
DStub	Fig. 4.1(b)	IN1<—>StartPoint_IN	OUT1<—>EndPoint_OUT1 OUT2<—>EndPoint_OUT2
DStub	Fig. 4.1(c)	IN1<—>StartPoint_IN	OUT1<—>EndPoint_OUT1 OUT2<—>EndPoint_OUT2
SStub	Fig. 4.1(d)	IN1<—>StartPoint_IN1 IN2<—>StartPoint_IN2	OUT1<—>EndPoint_OUT1
SStub2	Fig. 4.1(e)	IN1<—>StartPoint_IN1	OUT1<—>EndPoint_OUT

Table 4.2: Plug-in bindings of stubs in Fig. 4.1

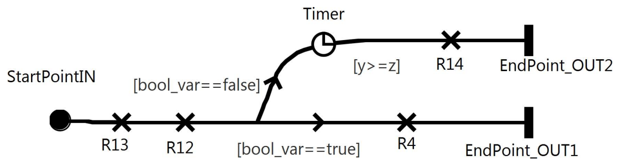
The mock system has five maps shown in Fig. 4.1. The main map is shown in Fig. 4.1(a), and it contains a static stub *(SStub)* and a dynamic stub *(DStub)*. The plug-in map bound to *(SStub)*, shown in Fig. 4.1(d), also contains a static stub, *(SStub2)* that is bound to the map in Fig. 4.1(e). This forms three levels of hierarchy where the top level is in Fig. 4.1(a) and the lowest level is the map in Fig.4.1(e). *(DStub)* in Fig. 4.1(a) is bound to two different maps Fig. 4.1(b) and 4.1(c). The plug-in bindings of these stubs are shown in Table. 4.2. Every stub has one or more stub entries (*INs*) and one or more stub exit node connections (*OUTs*). Plug-in bindings connect *INs* node connections in the parent



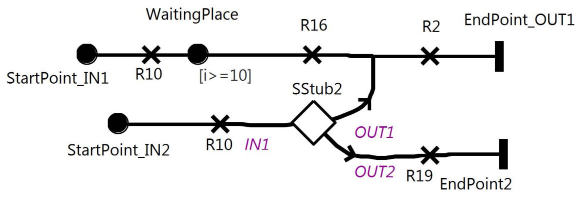
(a) Mock system: Main Map



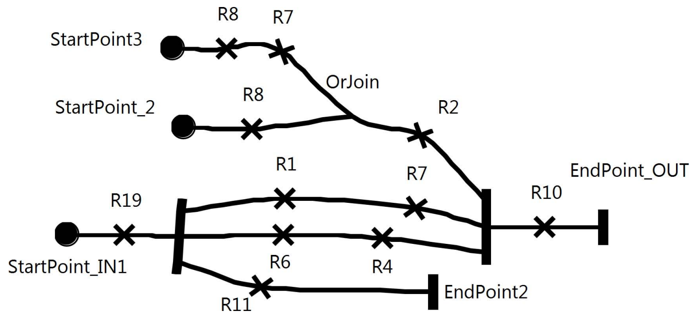
(b) Mock system: Dynamic Stub plug-in map 1



(c) Mock system: Dynamic Stub plug-in map 2



(d) Mock system: Static Stub plug-in map 1



(e) Mock system: Static Stub plug-in map 2

Figure 4.1: UCM Mock model

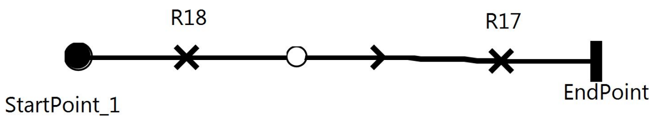
map to start points in the plug-in map whereas end points within the plug-in map are connected to ($OUTs$) node connections in the parent map. For example, DStub in Table. 4.2 has two plug-in maps since it is a dynamic stub. The first binding is in Fig. 4.1(b) where IN1 is bound to a start point called StartPoint_IN. DStub has two *OUTs*, *OUT1* and *OUT2*, where they are bound to end points in map Fig. 4.1(b), EndPoint_OUT1 and EndPoint_OUT2 respectively. We used the *respRefs*, described in Fig. 3.1, to evaluate the accuracy of the proposed slicing approach with intensive use of data and control flow and different slicing criteria. The Output slices are generated based on different slicing criteria within different maps. For each example, we specify the map in which the selected *respRef* resides. Evaluation examples with several slicing criteria are shown in figures 4.2(a), 4.2(b), 4.2(c), 4.3, 4.4, 4.5, 4.6 using executable slice approach. The same output slices using closure slice approach are shown in figures 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13.

4.1.2 Case Studies

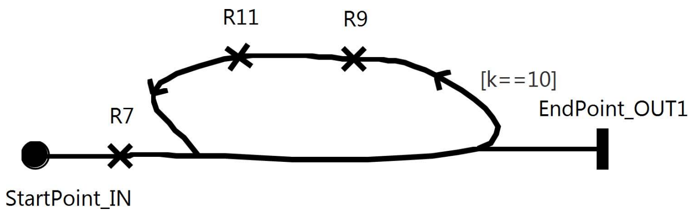
In addition of the Mock model, we also implemented the slicing approach on three publicly available case studies that vary in size and complexity, as shown in Table 4.1.

Case Study 1: Discharge Process Management System

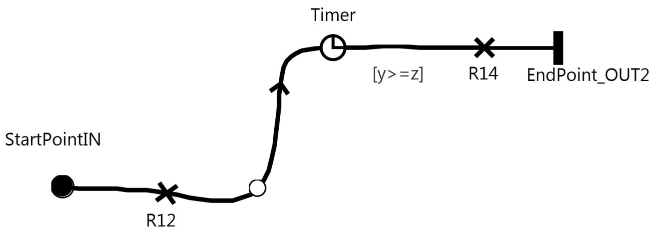
This case study describes a discharge process management at the Ottawa hospital, designed by Pourshahid et al. [59]. Due to the large size of the UCM specifica-



(a) Executable slice when *SC=(R17 in rootmap Fig. 4.1(a), (x, j, k))*

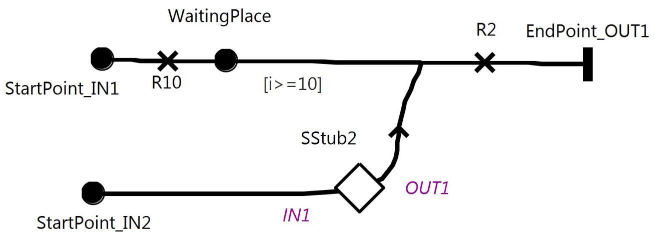


(b) Executable slice when *SC=(R9 in map Fig. 4.1(b), y)*

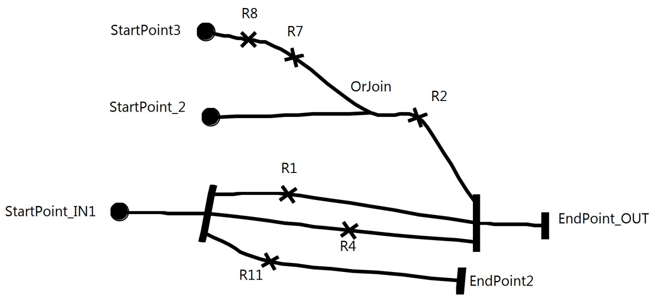


(c) Executable slice when *SC=(R14 in map Fig. 4.1(c), x)*

Figure 4.2: Executable slices for many slicing criteria

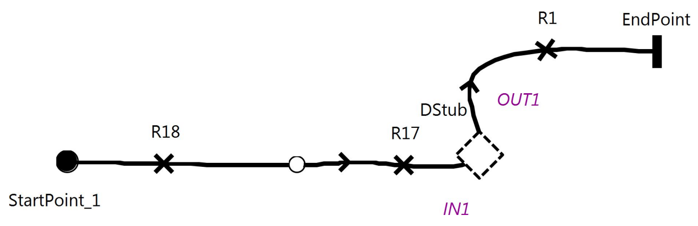


(a) Part of output slice: map Fig. 4.1(d)

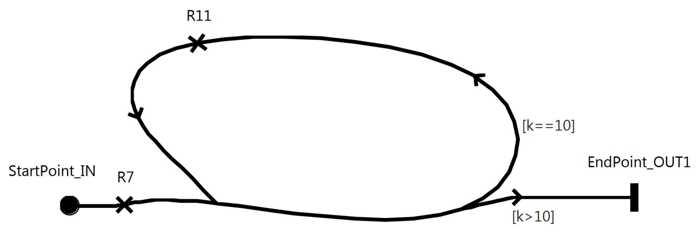


(b) Part of output slice: map Fig. 4.1(e)

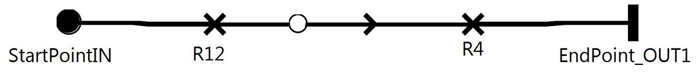
Figure 4.3: Executable slice when *SC=(R2 in map Fig. 4.1(d), x)*



(a) Part of output slice: map Fig. 4.1(a)

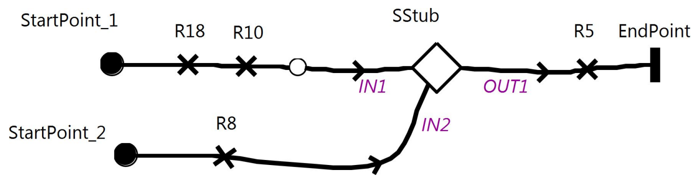


(b) Part of output slice: map Fig. 4.1(b)

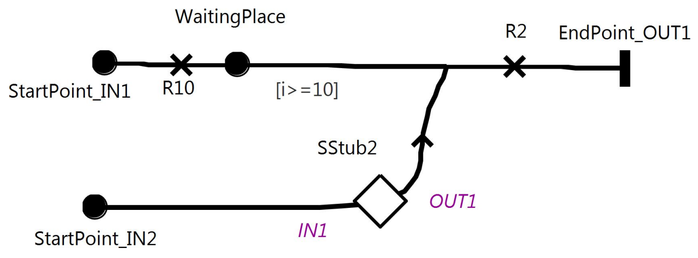


(c) Part of output slice: map Fig. 4.1(c)

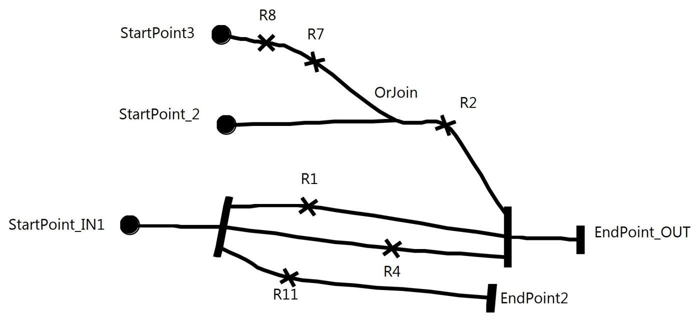
Figure 4.4: Executable slice when *SC=(R1 in map Fig. 4.1(a), (x,y)*



(a) Part of output slice: map Fig. 4.1(a)

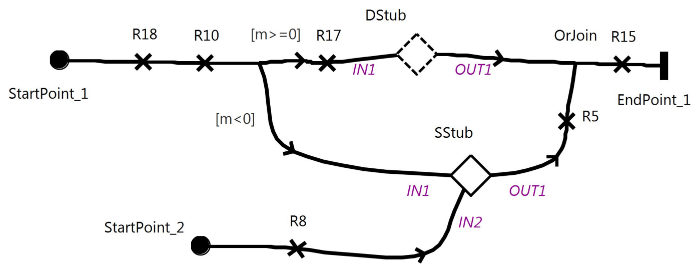


(b) Part of output slice: map Fig. 4.1(d)

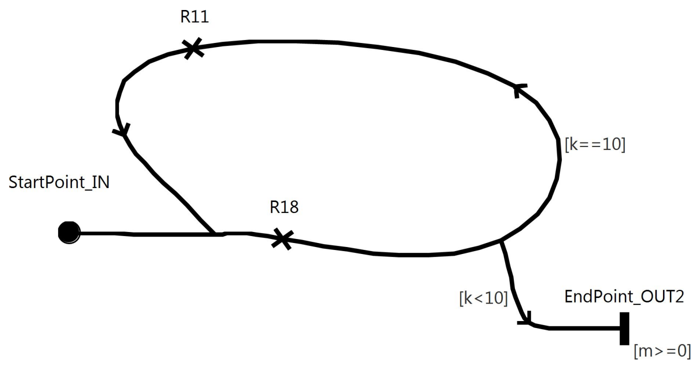


(c) Part of output slice: map Fig. 4.1(e)

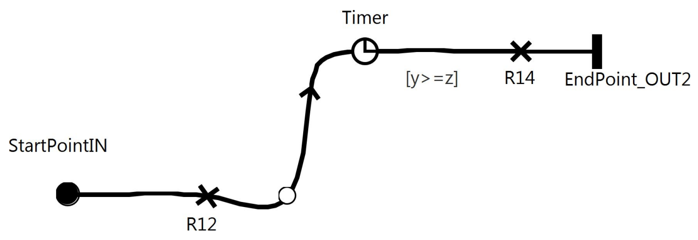
Figure 4.5: Executable slice when *SC=(R5 in map Fig. 4.1(a), x)*



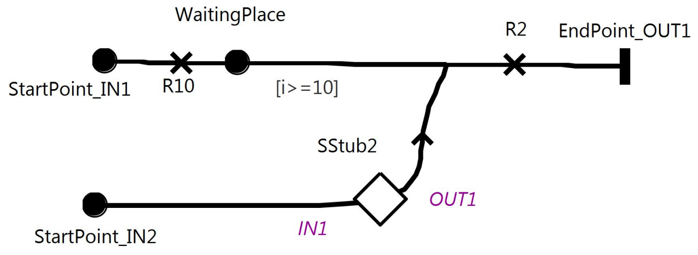
(a) Part of output slice: map Fig. 4.1(a)



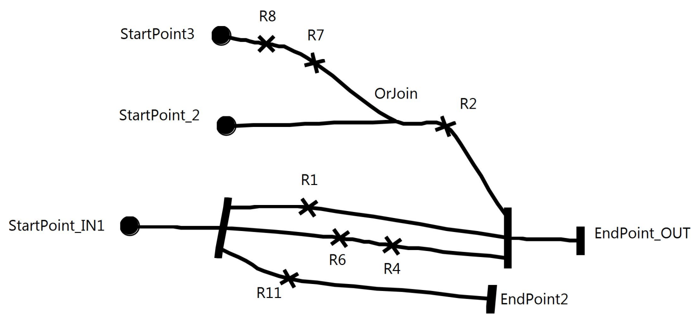
(b) Part of output slice: map Fig. 4.1(b)



(c) Part of output slice: map Fig. 4.1(c)



(d) Part of output slice: map Fig. 4.1(d)



(e) Part of output slice: map Fig. 4.1(e)

Figure 4.6: Executable slice when *SC=(R15 in map Fig. 4.1(a), x)*

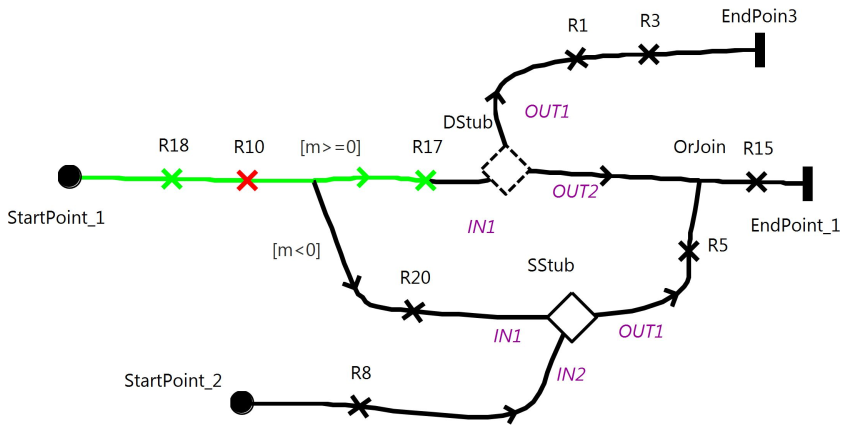


Figure 4.7: Closure slice when *SC=(R17 in map Fig. 4.1(a), (x,j,k))*

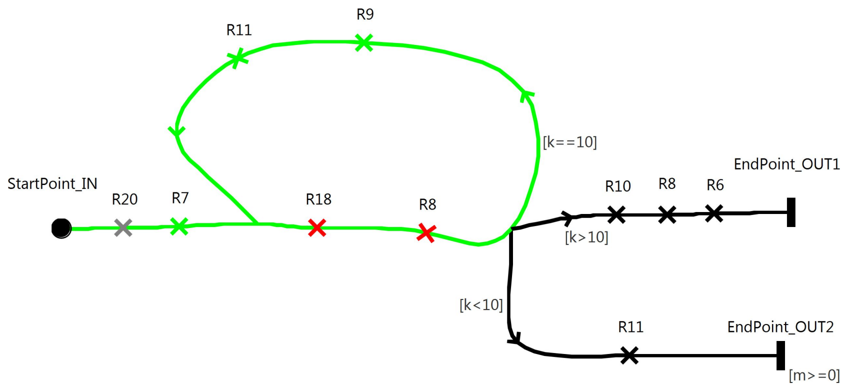


Figure 4.8: Closure slice when *SC=(R9 in map Fig. 4.1(b), y)*

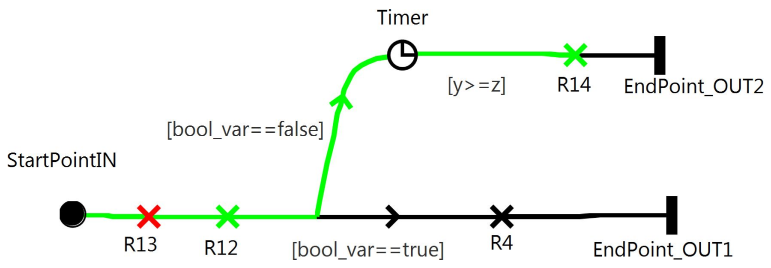
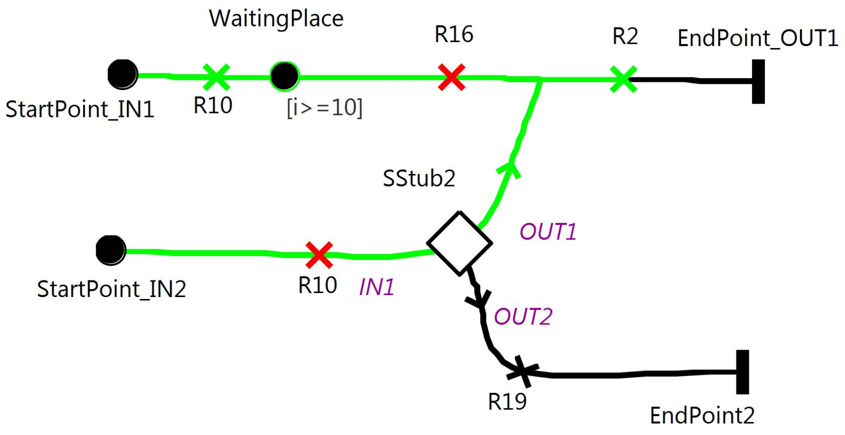
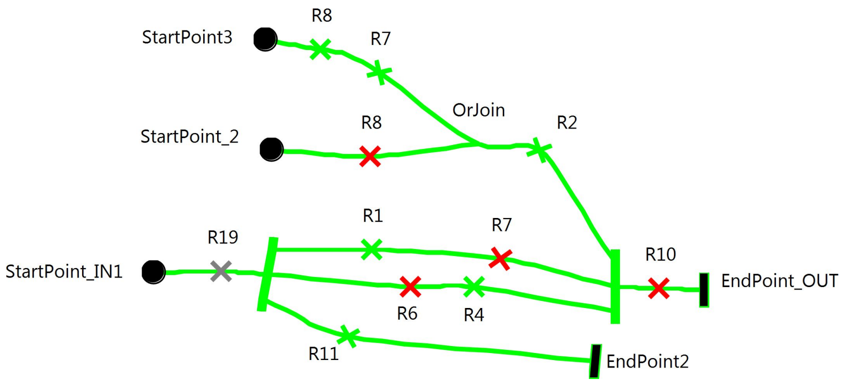


Figure 4.9: Closure slice when *SC=(R14 in map Fig. 4.1(c), x)*

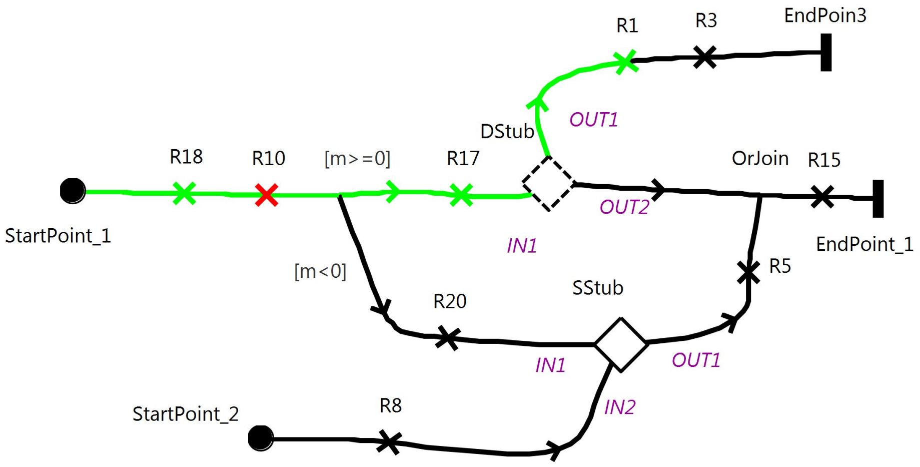


(a) Part of output slice: map Fig. 4.1(d)

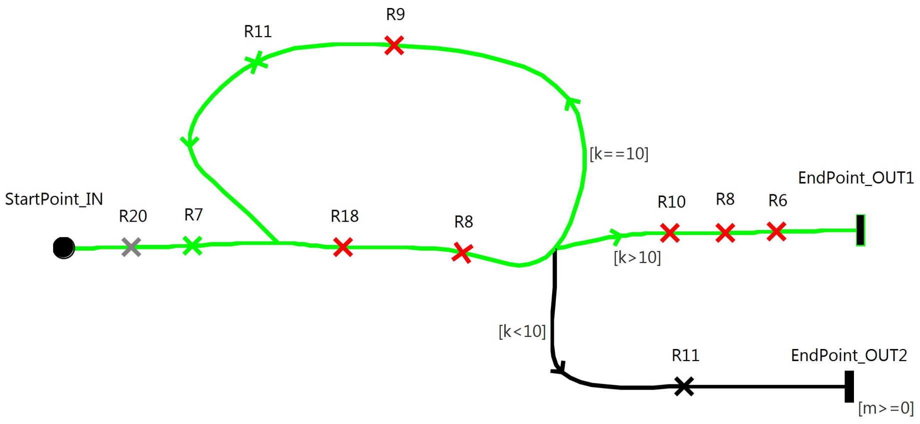


(b) Part of output slice: map Fig. 4.1(e)

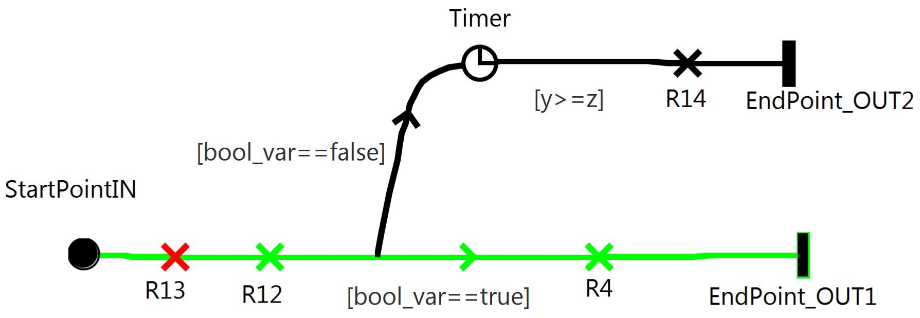
Figure 4.10: Executable slice when *SC=(R2 in map Fig. 4.1(d), x)*



(a) Part of output slice: map Fig. 4.1(a)

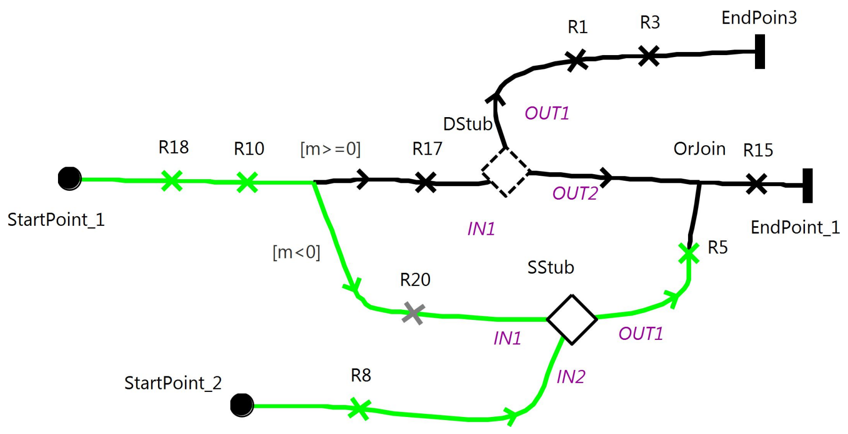


(b) Part of output slice: map Fig. 4.1(b)

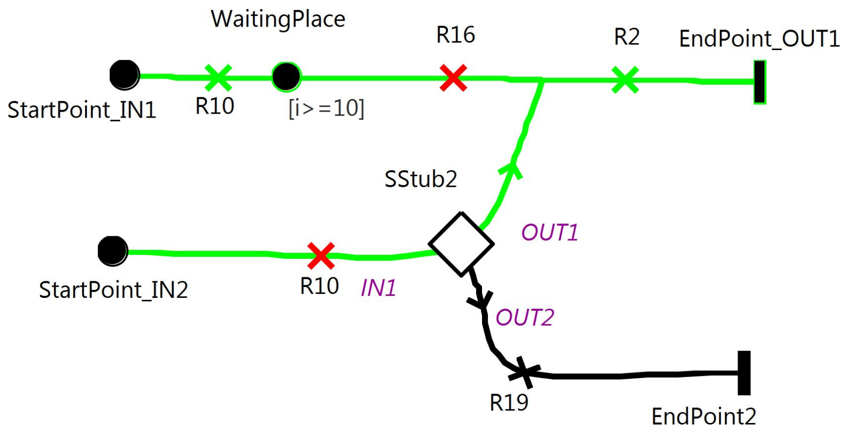


(c) Part of output slice: map Fig. 4.1(c)

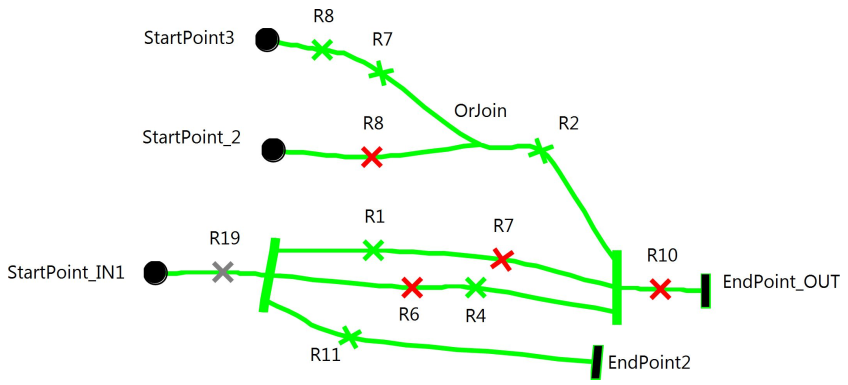
Figure 4.11: Closure slice when *SC=(R1 in map Fig. 4.1(a), (x,y)*



(a) Part of output slice: map Fig. 4.1(a)

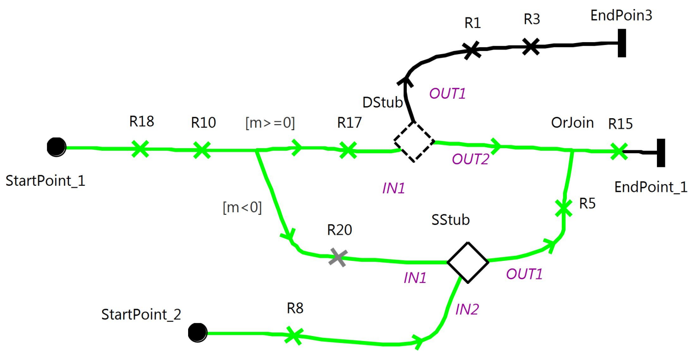


(b) Part of output slice: map Fig. 4.1(d)

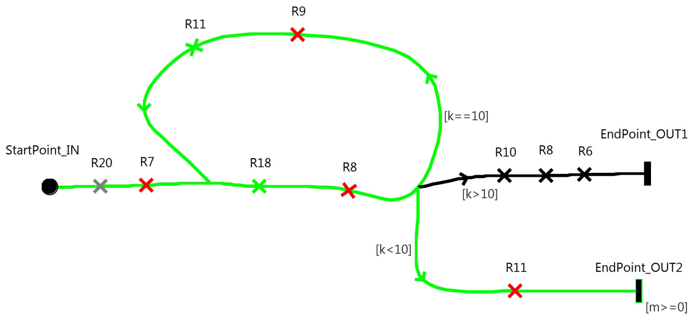


(c) Part of output slice: map Fig. 4.1(e)

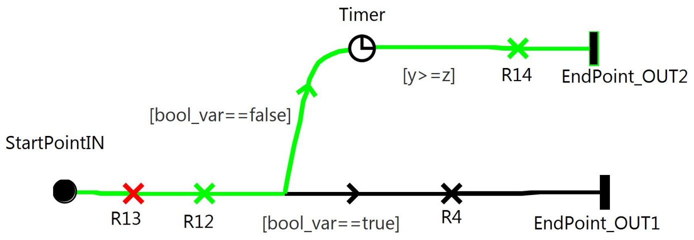
Figure 4.12: Closure slice when *SC=(R5 in map Fig. 4.1(a), x)*



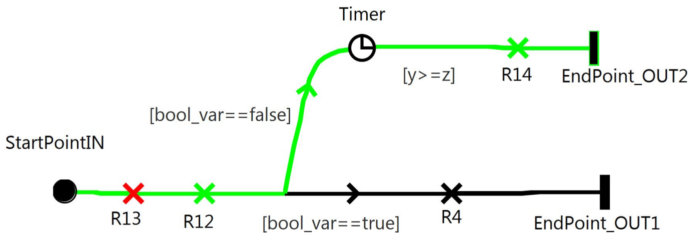
(a) Part of output slice: map Fig. 4.1(a)



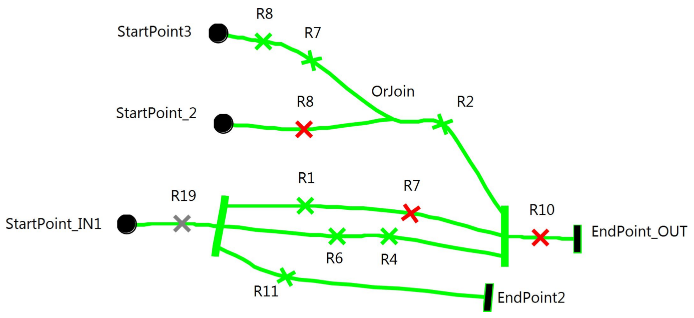
(b) Part of output slice: map Fig. 4.1(b)



(c) Part of output slice: map Fig. 4.1(c)



(d) Part of output slice: map Fig. 4.1(d)



(e) Part of output slice: map Fig. 4.1(e)

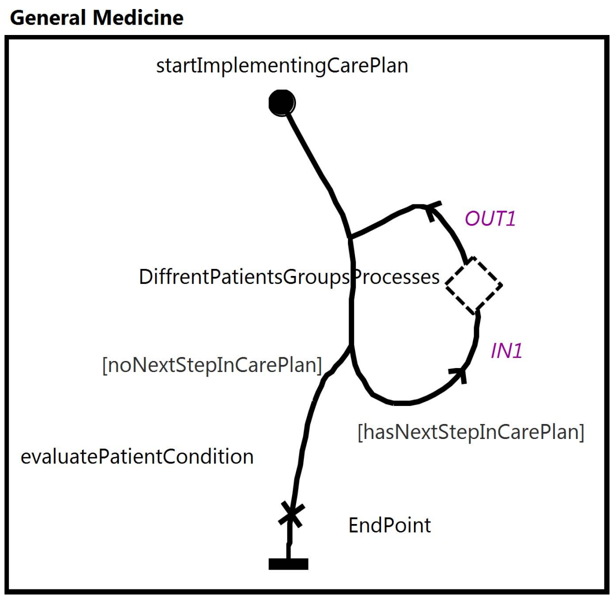
Figure 4.13: Closure slice when *SC=(R15 in map Fig. 4.1(a), x)*

tion (contains 43 maps) and because of the lack of space, we only illustrate the generated slices. The reader is referred to [59] to consult the original UCM maps.

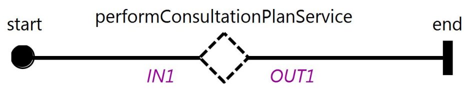
Two executable slices were producd. The first executable slice is shown in Fig. 4.14, where SC = (evaluatePatientCondition, _evaluationOfImplementationPassed). The selected *respref*, *evaluatePatient-Condition*, resides within *CarePlanImplementation* map while the selected variable, *_evaluationOfImplementationPassed*, is one of the variables used within the expression of *evaluatePatientCondition*. The resulting slice is an executable UCM model that contains only 10 out of 43 maps in the original discharge model. The reason behind having 10 maps is that the dynamic stub *DifferentPatients-GroupsProcesses*, in Fig. 4.14(a), has plug-in bindings to 7 different maps, while *PerformConsultationPlanService* stub in Fig. 4.14(b) is bound to 2 different maps.The second evaluation example in the Discharge process model is illustrated in Fig. 4.15, where SC=(recievedByCommunityProviders, admittedByHospital). The selected *respref* resides within *DictateProcess* map, and it contains the variable *addmittedByHospital* within its expression.

Case Study 2: On-line Ordering System

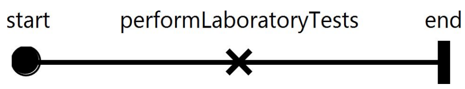
The second case study is an on-line ordering system which consists of 4 maps illustrated in Fig. 4.16. The plug-in bindings of the stubs and the list of model elements that have code expressions are shown in Tables 4.4 and 4.3 respectively. Three executable slices are shown in figures 4.17, 4.18, and 4.19 associated with the slicing criteria *ShipOrder*, *SubmitFinalOrder*,and *ProcessOrder* respectively.



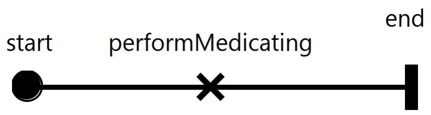
(a) Part of executable slice, map: *CarePlanImplementation*



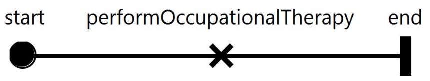
(b) Part of executable slice, map: *ConsultationPlanService*



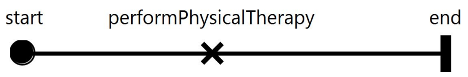
(c) Part of executable slice, map: *LaboratoryTests*



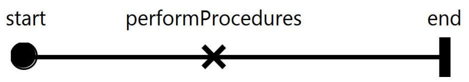
(d) Part of executable slice, map: *Medicating*



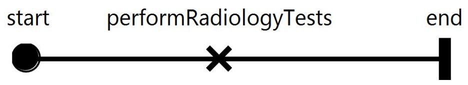
(e) Part of executable slice, map: *OccupationalTherapy*



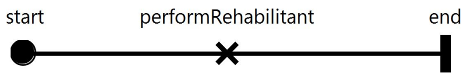
(f) Part of executable slice, map: *PhysicalTherapy*



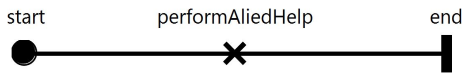
(g) Part of executable slice, map: *Procedures*



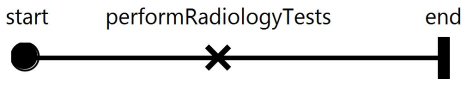
(h) Part of executable slice, map: *RadiologyTests*



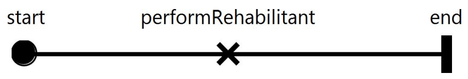
(i) Part of executable slice, map: *Rehabilitant*



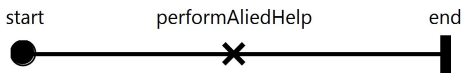
(j) Part of executable slice, map: *AlliedHelp*



(k) Part of executable slice, map: *RadiologyTests*

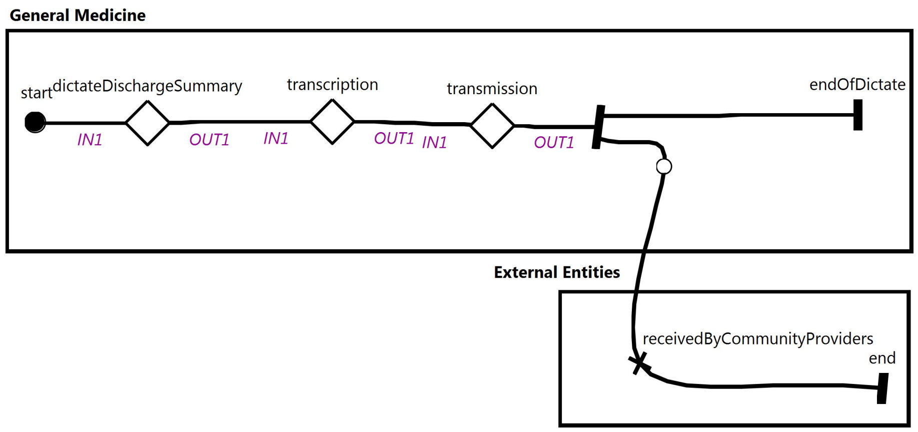


(l) Part of executable slice, map: *Rehabilitant*



(m) Part of executable slice, map: *AlliedHelp*

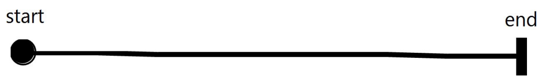
Figure 4.14: Discharge Model: an executable slice, SC=(evaluatePatientCondition,_evaluationOfImplemntationPassed)



(a) Part of executable slice, map: *DictateProcess*

(b) Part of executable slice, map: *Transmission*

(c) Part of executable slice, map: *Transcription*



(d) Part of executable slice, map: *Transcription*

Figure 4.15: Discharge Model: executable slice, SC=(RecievedByCommunityProviders,admittedByHospital)

In Fig. 4.18, no variables are selected along with *SubmitFinalOrder* since it contains no code within its expression, and the result is a reduced model with no data dependency computation. This means that only un-traversed branches and model elements are removed from the executable slice, and if there were any respRef(s) that resides within the slice path, it would have been kept since there is no computation of data dependencies.

Case Study 3: Adverse Event Management System

The third case study describes an adverse event management system (AEMS) that consists of two UCM maps illustrated in Fig. 4.20. The model elements that contains code expressions are shown in Table 4.5. Two executable slices generated with different slicing criteria are illustrated in Figures 4.21 and 4.22.

4.1.3 Characterization of the reduction rates

Table 4.6 shows the reduction rates for the mock system and the three case studies. The reduction rate of each executable slice, generated by the slicing approach, is calculated with respect to the original model size and considering different slicing criteria. The sizes of UCM specification are computed in terms of number of nodes.

Depending on the location of the slicing criterion within the specification, different reduction rates may be obtained. The average model reduction of each model is as follows: Mock (*73%*), Discharge system (*89.5%*), Ordering system (*59%*) and AEMS (*60.5%*). The overall average reduction of the slicing approach

Element Name	**Type**	**Map**	Expression

Process Order	RespRef	Order	OrderToProcess= true;
			ProductShipped= false;

Ship Order	RespRef	Order	ProductShipped=true;
Gather Products	RespRef	Order	IsProductAvailable=!IsWarehouseOutOfStock;

Increment Inventory	RespRef	Order	IsWarehouseOutOfStock =!ReceivedMissingProduct;
			IsProductAvailable=ReceivedMissingProduct;

IsProductAvailable	Orfork Node Connection	Order	IsProductAvailable
item(s) missing	Orfork Node Connection	Order	! IsProductAvailable

cancel missing items or order	RespRef	BackOrdered	if (UserDecision==CancelMissing)
			IsProductAvailable=true;
			else // if (UserDecision == Cancel)
			IsProductAvailable = false;

-	Timer Node Connection	BackOrdered	IsProductAvailable
Cancelled by user	Orfork Node Connection	BackOrdered	UserDecision==Cancel
additional delays	Orfork Node Connection	BackOrdered	UserDecision != Cancel
cancel missing	Orfork Node Connection	WaitForOrder	UserDecision == CancelMissing
cancel order	Orfork Node Connection	WaitForOrder	UserDecision==Cancel
wait more	Orfork Node Connection	WaitForOrder	UserDecision==Wait
Received	Orfork Node Connection	WaitForOrder	ProductShipped
else	Orfork Node Connection	WaitForOrder	! ProductShipped
product available	Orfork Node Connection	shop	IsProductAvailable
product not available	Orfork Node Connection	shop	! IsProductAvailable

Table 4.3: Ordering System: model elements that have code expressions

Stub name	**plug-in Map**	**IN-bindings**	**OUT-bindings**
Wait for supplier	Backordered	IN1<–>Back Brdered	OUT1<–>Ship Order OUT2<–>notify user delays OUT3<–>Cancelled
Wait for Order	WaitForOrder	IN2<–>Wait	OUT1<–>Recieved OUT2<–>Shop Elsewhere
shop	Shop	IN1<–>Shop	OUT1<–>Product Available OUT2<–>Product not available

Table 4.4: Plug-in bindings of stubs in Ordering System

Element Name	**Type**	**Map**	**Expression**
EditEventForVisit	RespRef	Process	ExistingEvent = false;

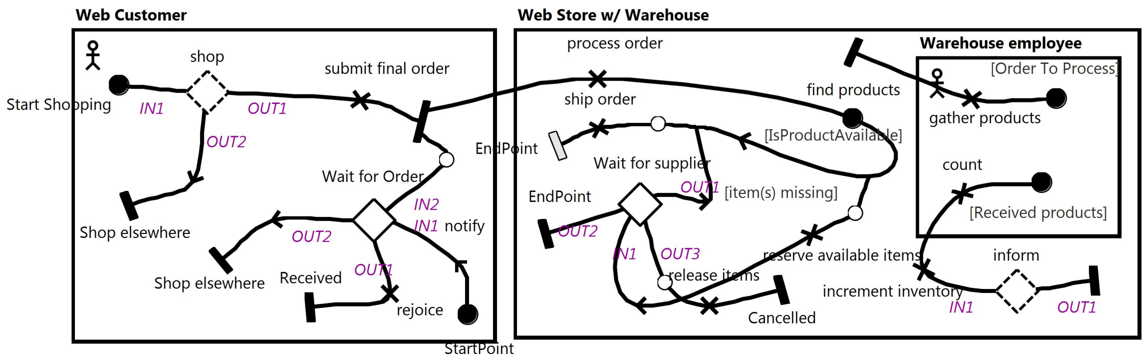
WarnObserver	RespRef	Process	ExistingEvent= true;
			EventComplete = true;
			EventReady = true;

Event Not Complete	Orfork Node Connection	Process	! EventComplete
Event Complete	Orfork Node Connection	Process	EventComplete
Event Not Ready	Orfork Node Connection	Process	! EventReady
Event Ready for Review	Orfork Node Connection	Process	EventReady
Patient Gone	Orfork Node Connection	Process	Discharged
Patient Not Discharged	Orfork Node Connection	Process	! Discharged
EventComplete	Orfork Node Connection	Process	EventComplete
Event Not Complete	Orfork Node Connection	Process	! EventComplete
AEMS-CreateEvent	RespRef	PrepareEvent	EventsCreated = EventsCreated + 1;
DischargePatient	RespRef	PrepareEvent	Discharged = true;
New Event	Orfork Node Connection	PrepareEvent	(EventsCreated <NumEvents) && !Ex-istingEvent
Existing Event	Orfork Node Connection	PrepareEvent	else
Patient Present	Orfork Node Connection	PrepareEvent	(NumEvents >EventsCreated) || ExistingEvent
Patient Gone	Orfork Node Connection	PrepareEvent	else

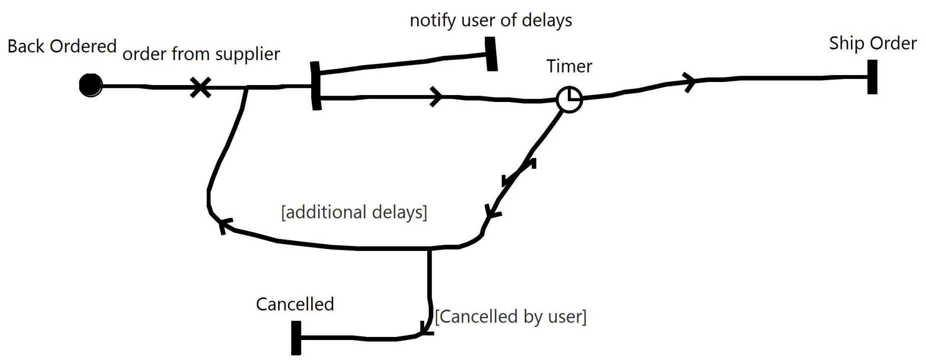
Table 4.5: Adverse Event Management System: model elements that have code expressions

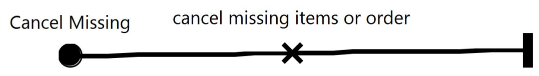
Model	**Executable Slice**	**Slice Size**	**Original Spec Size**	**Reduction Rate**
Mock	Fig. 4.2(a)	6	86	93%
Mock	Fig. 4.2(b)	9	86	89.5%
Mock	Fig. 4.2(c)	7	86	92%
Mock	Fig. 4.3	23	86	73%
Mock	Fig. 4.4	24	86	72%
Mock	Fig. 4.5	35	86	59%
Mock	Fig. 4.6	56	86	35%
Discharge	Fig. 4.14	37	257	85%
Discharge	Fig. 4.15	15	257	94%
Ordering	Fig. 4.17	57	82	30%
Oredring	Fig. 4.18	6	82	93%
Ordering	Fig. 4.19	38	82	54%
AEMS	Fig. 4.21	30	43	30%
AEMS	Fig. 4.22	4	43	91%

Table 4.6: Reduction rates of original models after generating executable slices

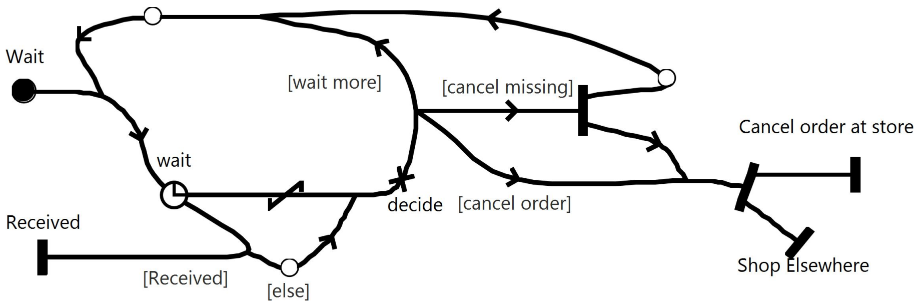


(a) Order map

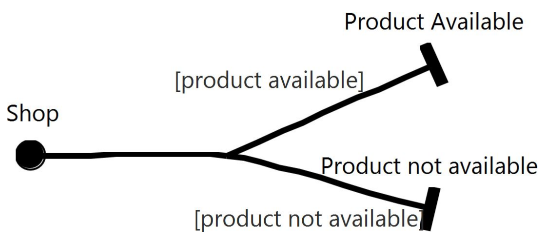




(b) Backordered map

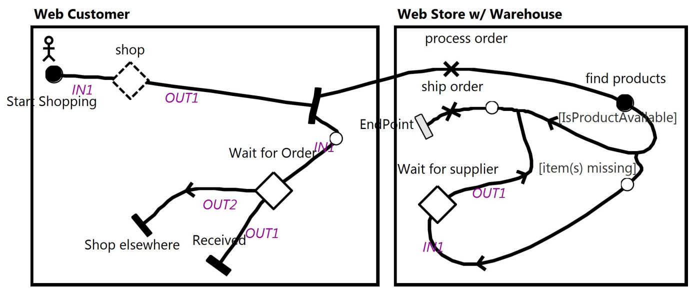


(c) WaitForOrder map

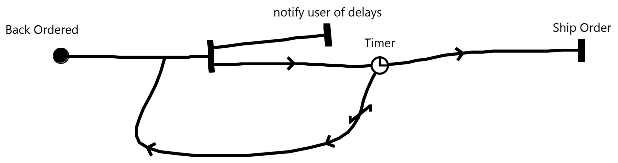


(d) Shop map

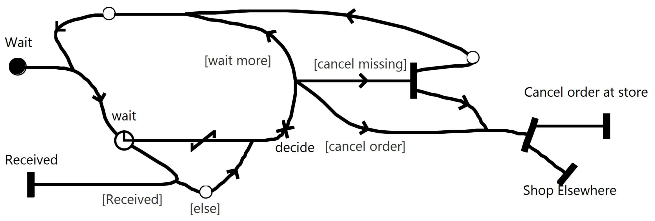
Figure 4.16: Ordering Model



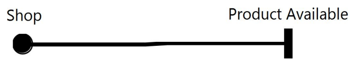
(a) Part of executable slice, map: Order



(b) Part of executable slice, map: Backordered

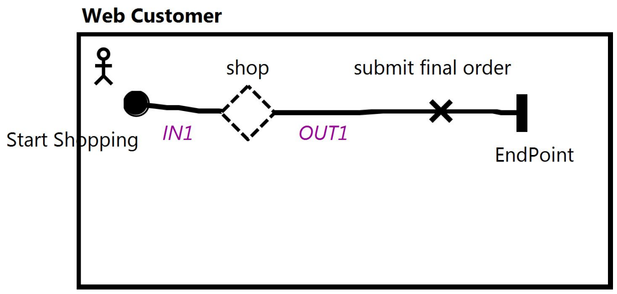


(c) Part of executable slice, map: WaitForOrder

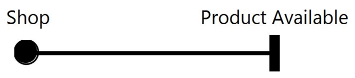


(d) Part of executable slice, map: Shop

Figure 4.17: Ordering Model: executable slice, SC=(ShipOrder,ProductShipped)

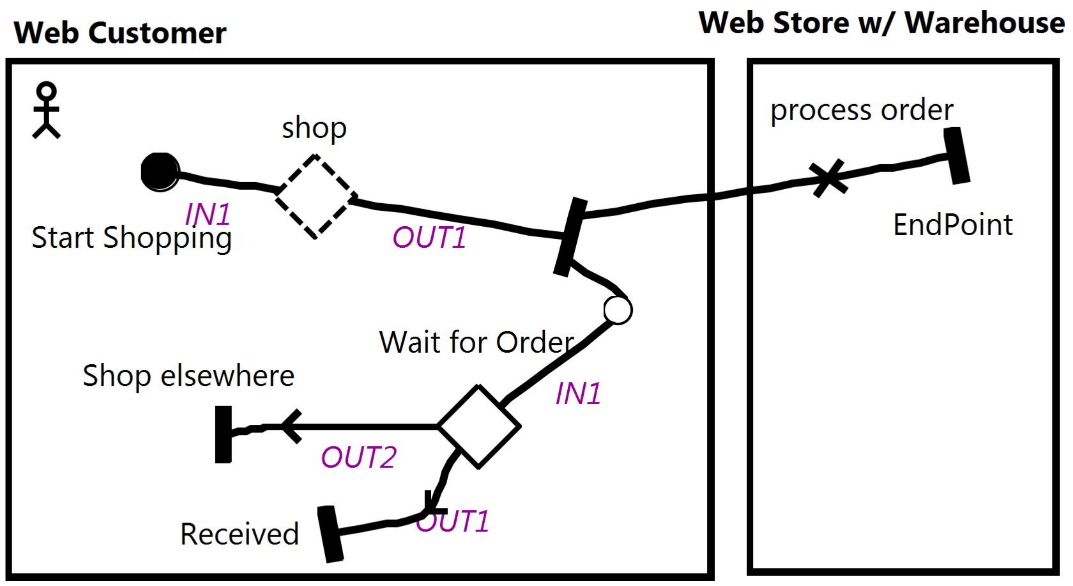


(a) Part of executable slice, map: Order

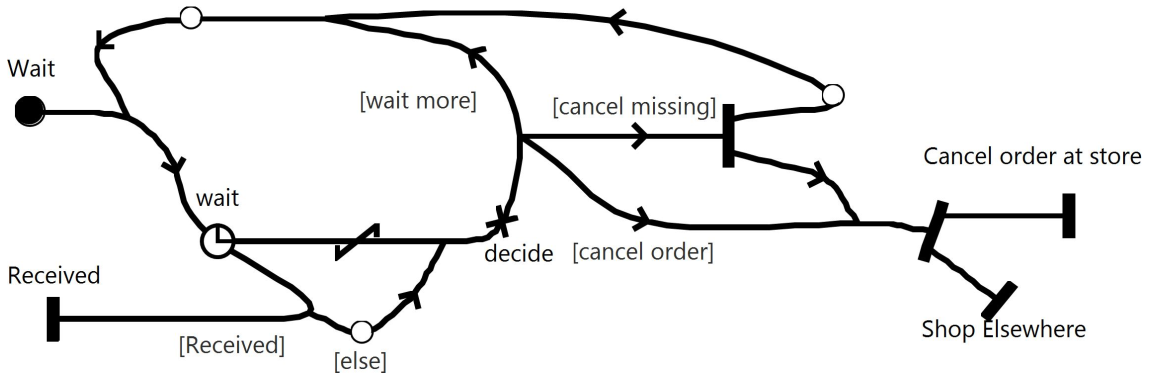


(b) Part of executable slice, map: WaitForOrder

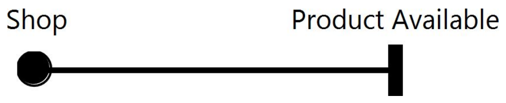
Figure 4.18: Ordering Model:executable slice, SC=(SubmitFinalOrder,-)



(a) Part of executable slice, map:Order

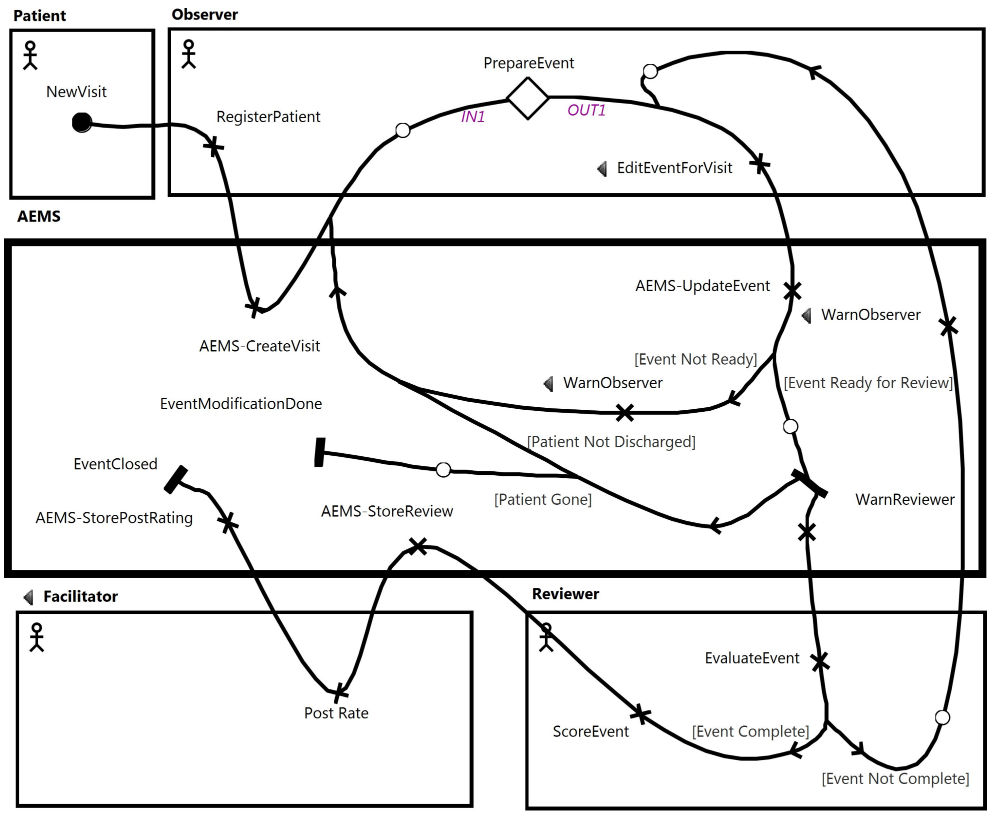


(b) Part of executable slice, map: Backordered

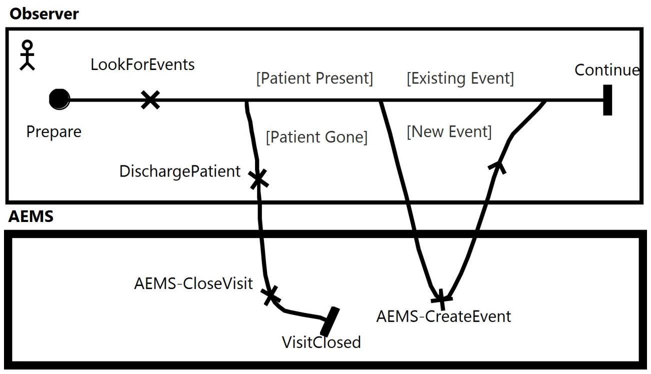


(c) Part of executable slice, map: WaitForOrder

Figure 4.19: Ordering Model:executable slice, SC=(ProcessOrder,OrderToProcess)

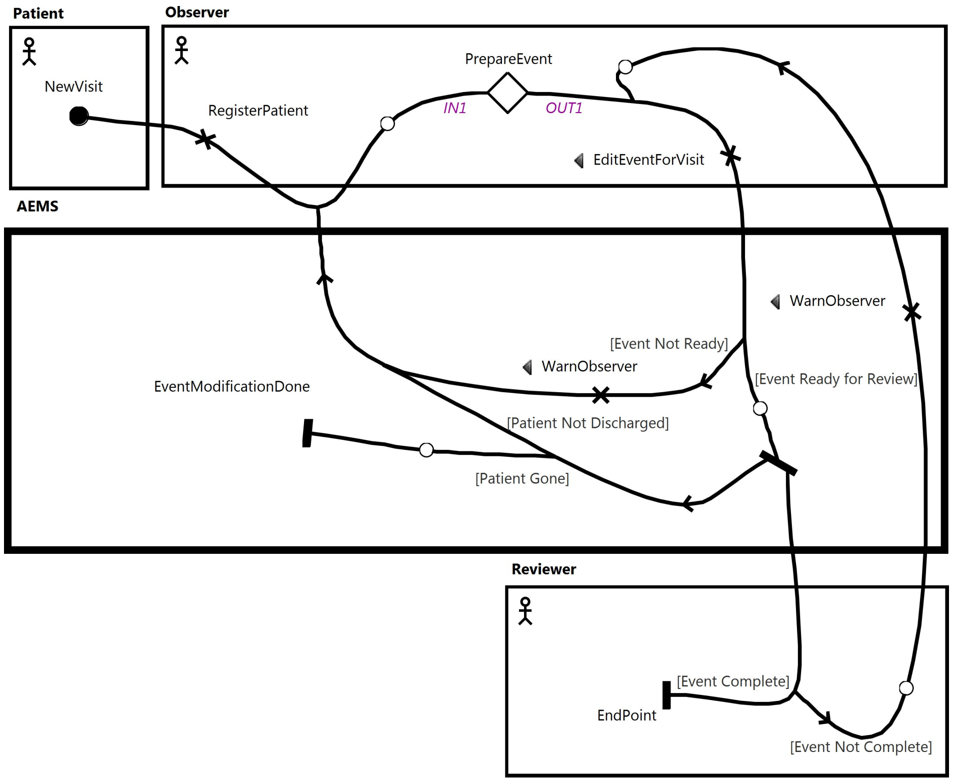


(a) Process map

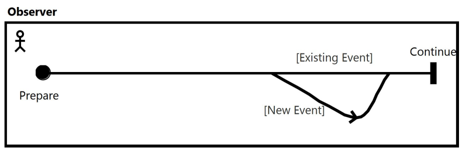


(b) PrepareEvent map

Figure 4.20: Adverse Event Management System



(a) Part of executable slice, map: Process



(b) Part of executable slice, map: PrepareEvent

Figure 4.21: Adverse Event Management System: executable slice, SC=(WarnObserver, EventReady)

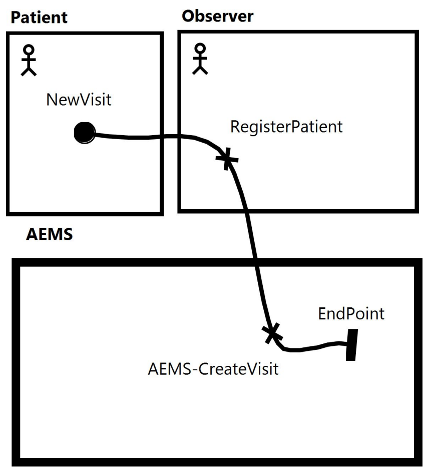


Figure 4.22: Adverse Event Management System: executable slice, SC=(AEMS-CreateVisit, -)

on the four models is *70.5%*. In summary, the slicing approach can reduce UCM specification from *30%* to *94%*. An average reduction rate of *70%* is obtained for the four case studies.

4.2 Empirical Validation

This section describes the experiment carried out to provide empirical evidence with regard to the potential benefits of the UCM-based static slicing feature in facilitating the understandability of UCM models.

Part of the UCM understandability process is to be able to identify which UCM paths are executed to reach a specific construct in the specification. Such paths may visit many plugin-maps, follow a branch based on some conditions, enter loops, execute concurrent paths, etc. Furthermore, in presence of variables within a UCM model, some responsibilities (having embedded code) may not contribute to the reachability of some parts of the UCM specification. Discarding such unrelated responsibilities would help understand which responsibilities really influence a certain scenario execution.

The main goal of this experiment is to ascertain whether the use of slicing improves the understandability of UCM models. The research question can be stated as follows: **"Does the use of UCM-based static slicing facilitate the understandability of UCM models?"**

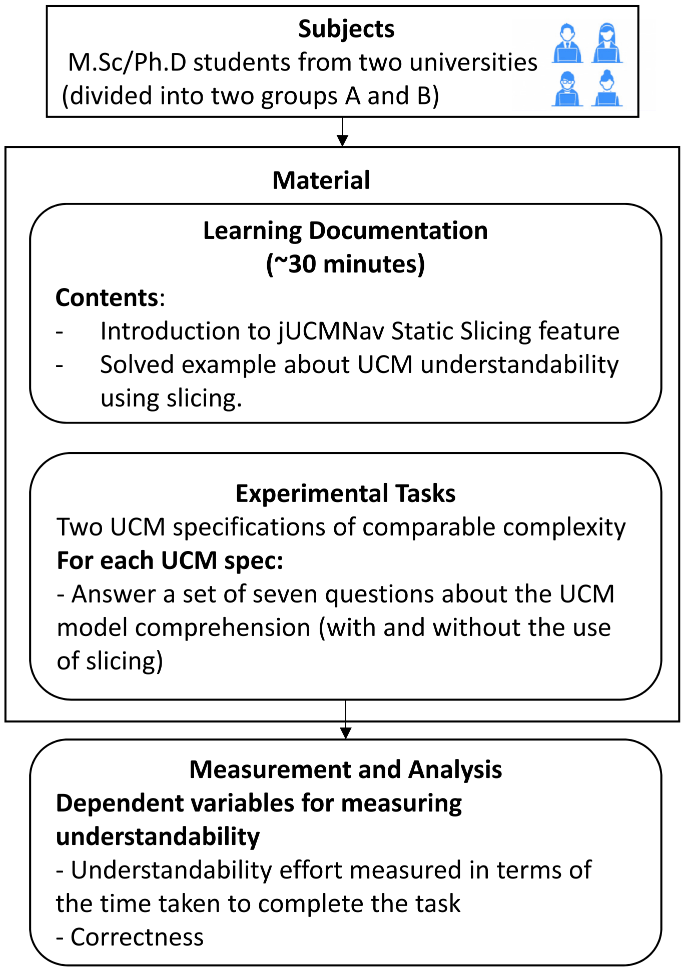

Subjects
M.Sc/Ph.D students from two universities
(divided into two groups A and B)
Material
Learning Documentation
(~30 minutes)
Contents:
- Introduction to jUCMNav Static Slicing feature
- Solved example about UCM understandability using slicing.
Experimental Tasks
Two UCM specifications of comparable complexity
For each UCM spec:
- Answer a set of seven questions about the UCM model comprehension (with and without the use of slicing)
Measurement and Analysis
Dependent variables for measuring understandability
- Understandability effort measured in terms of the time taken to complete the task
- Correctness

Figure 4.23: Overview of the experimental plan

4.2.1 Experiment planning

The general goal of the empirical study is derived as follows: "To analyze the impact of using UCM-based static slicing in improving the understandability of UCM models."

In order to test the proposed hypotheses (see Sect. 4.2.1), an experiment was designed and conducted. This is achieved by following the templates and recommendations presented in Wohlin et al. [60], Juristo and Moreno [61], Kitchenham et al. [62], and Jedlitschka and Ciolkowski [63]. Figure 4.23 shows an overview of the experimental plan. Each of the steps of this experimental plan is explained in greater detail in the following sections.

Subjects

The subjects of the experiment were 8 members of two universities from different countries. The group of subjects was chosen for their strict compliance with the following characteristics:

- M.Sc/Ph.D students in computer science or software engineering.
- Their experience with modeling and more specifically their experience with the UCM language and their familiarity with the jUCMNav tool.
- Their lack of knowledge of the new UCM-based slicing feature.

Material

The material given to the subjects was composed of two parts:

Learning Documentation: this part of the material was prepared to provide subjects with the information needed to carry out the experimental tasks. This documentation was planned to be read in about 30 minutes on average and it was composed of the following parts:

- An introduction to the UCM slicing feature.
- An introduction to the jUCMNav slicing feature.
- Instructions that subjects should follow to carry out the experimental tasks.
- A solved example of the understandability tasks using the jUCMNav slicing feature.

Experimental Tasks (Understandability Tasks): The material given to each subject is summarized in Table 4.2.1. Case studies 1 and 2 have a total of 7 questions of very similar complexity, as assured by three UCM experts.

Group A
Case study 1: Ordering system. **Three questions** about understandability of the model to be answered **without using the slicing feature**.
Case study 2: Adverse event management system. **Four questions** about understandability of the model to be answered **using the slicing feature**.
Group B
Case study 1: Adverse event management system. **Four questions** about understandability of the model to be answered **without using the slicing feature**.
Case study 2: Ordering system. **Three questions** about understandability of the model to be answered **using the slicing feature**.

Table 4.7: Experiment Material

Variables

We measure understandability by means of the following dependent variables: (1) the time spent by the subjects in answering the seven questions, (2) the correctness of the obtained answers. The independent variable is the performed understandability tasks.

Hypotheses

The experiment is planned with the purpose of testing the hypotheses stated in Table 4.8 which, for each set of hypotheses, details the null and alternative hypothesis formulation and the dependent variables. The first hypothesis tests whether the use of slicing improves the understandability of UCM specifications, assessed by checking the correctness of the comprehension tasks. The second hypothesis tests whether the use of slicing facilitates the understandability of UCM specifications, assessed by measuring and comparing the time that it takes to complete a comprehension task with and without the use of the slicing feature.

4.2.2 Data analysis and interpretation

Once the experiment had been carried out, we collected the forms filled in by the subjects and we have discarded the unanswered questions. We have used SPSS to test our hypotheses. For the correctness variable, correct answers are coded as ”1”, while incorrect ones are coded as ”0”. The use of the slicing feature is coded as ”1”, while manual execution of the task is coded ”0”.

Hypotheses
Hypothesis 1
Null hypothesis–$H_0 - 1$**:** There are no differences in UCM spec understandability with or without using slicing.
Alternative hypothesis–$H_1 - 1$**:** There are differences in UCM spec understandability with or without using slicing.
Dependent variable: Correctness
Hypothesis 2
Null hypothesis–$H_0 - 2$**:** There are no differences in the time taken to understand UCM specs, with or without using slicing.
Alternative hypothesis–$H_1 - 2$**:** There are differences in the time taken to understand UCM specs, with or without using slicing.
Dependent variable: Time taken to perform the task

Table 4.8: Set of hypotheses

			Correctness		
			0	1	Total
Slicing	0	Count	9	11	20
		% within Slicing	45.0%	55.0%	100.0%
		% within Correctness	75.0%	28.2%	39.2%
	1	Count	3	28	31
		% within Slicing	9.7%	90.3%	100.0%
		% within Correctness	25.0%	71.8%	60.8%
Total		Count	12	39	51
		% within Slicing	23.5%	76.5%	100.0%
		% within Correctness	100.0%	100.0%	100.0%

Table 4.9: Slicing * Correctness cross tabulation

To test the first hypothesis $H_0 - 1$, we have computed a cross tabulation analysis on the use of slicing versus the correctness of the obtained answers. As shown in Table 4.9, using the slicing feature, we obtained 28 correct answers (90.3%) versus only 3 incorrect answers (9.66%), while the manual execution of the tasks produced 11 correct answers (55%) versus 9 incorrect answers (45%).

These results show that the understandability of the UCM specs has improved substantially by using the slicing feature. To prove that this improvement is significant, we have conducted Independent samples t-test with the correctness as test variable and the use of slicing as grouping variable. Table 4.10 illustrates the obtained results. The Levenes test shows that the equality of variances is not assumed (Sig. equal to 0.000 which is less than α:0.05). Based on the significance value 0.009 (which is less than 0.05), we can conclude that there is a significant difference between both the correctness of both groups (with and without use of slicing). Thus, we reject the null hypothesis $H_0 - 1$ and accept the alternative hypothesis $H_1 - 1$.

		Levene's Test for Equality of Variances		t-test for Equality of Means				
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference
Correctness	Equal variances assumed	33.706	.000	-3.115	49	.003	-.35323	.11340
	Equal variances not assumed			-2.798	27.576	.009	-.35323	.12625

Table 4.10: Test differences between means with respect to correctness (t-test)

To test the second hypothesis, we have discarded incorrect answers and conducted an independent samples t-test with the time spent to conduct a task as test variable and the use of slicing as grouping variable. The means of the time taken to perform the tasks with/without slicing is shown in Table 4.11. The mean times of completing a task are 102.86sec and 370.91sec, with and without using slicing, respectively.

Time

Slicing	Mean	N	Std. Deviation	Grouped Median
0	370.91	11	193.156	320.00
1	102.86	28	103.534	76.67
Total	178.46	39	179.909	110.53

Table 4.11: Means of the time taken to perform the tasks with/without slicing

The t-test results are illustrated in Table 4.12. The Levenes test shows that the equality of variances is not assumed (Sig. equal to 0.006 which is less than α:0.05). Based on the significance value 0.001 (which is less than 0.05), we can conclude that there is a significant differences between the time taken to perform the understandability tasks with and without the use of slicing. Hence, we reject the null hypothesis $H_0 - 2$ and accept the alternative hypothesis $H_1 - 2$.

		Levene's Test for Equality of Variances		t-test for Equality of Means				
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference
Time	Equal variances assumed	8.670	.006	5.629	37	.000	268.052	47.616
	Equal variances not assumed			4.363	12.327	.001	268.052	61.437

Table 4.12: Test differences between means with respect to the time spent to perform a task (t-test)

CHAPTER 5

DISCUSSION

In what follows, we discuss the benefits and shortcoming of our proposed slicing approach. We also compare it with related work and present potential threat of validity.

5.1 General benefits of the approach

The presented slicing approach have the following advantages:

- **Model reduction**: The first advantage of the presented slicing approach is model reduction. Reducing the size of a given UCM specification by eliminating unrelated model elements will enhance manual review of large UCM specifications, and will save the time that is usually spent on reviewing unrelated model elements. Consequently, large UCM specifications may be easier to comprehend and maintain using our slicing approach.

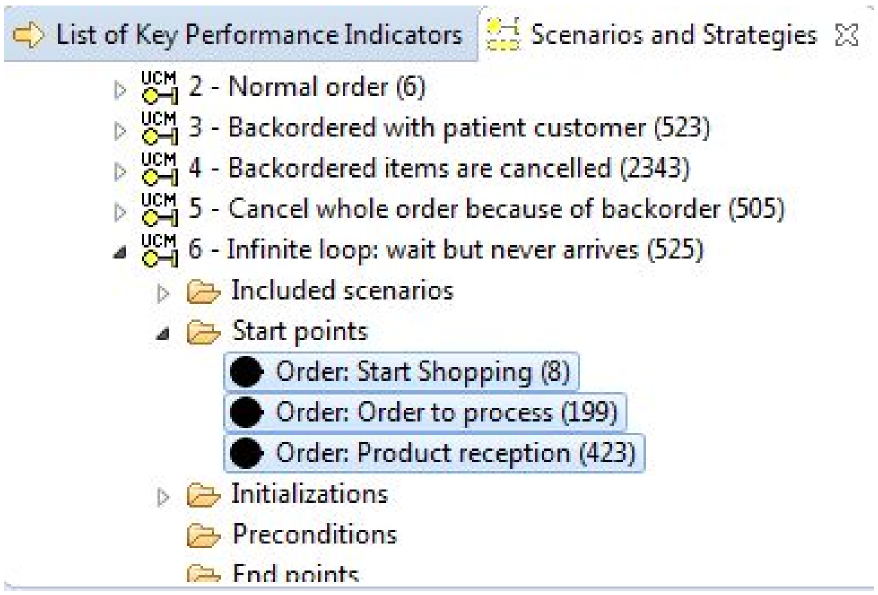
- **Supports UCM abstraction mechanism**: UCM specifications can be composed of hierarchical structures expressed by UCM stubs, defining multiple levels of abstractions. Our UCM slicing approach can handle all levels of abstraction starting from the root map, where the slicing criterion resides, traverse the lower level maps, compute dependencies, and resume traversing the upper-level maps (*parent maps*).

- **Completeness**: Our slicing approach covers all UCM language constructs.
- **Generated slices**: Two types of slices are generated, namely, a closure slice and an executable slice. The characteristics of each type of slice is presented in Sect. 3.7.
- **Loop recognition**: Traversing loops represent one important challenge in our slicing approach, since a loop should be detected in order to prevent infinite traversal.
- **Preserve semantics**: Graph connectivity of the executable slice is a known issue in State Based Model (SBM) slicing since removal of unrelated model elements can cause the rest of graph elements to be unreachable or disconnected [20]. Our presented slicing approach preserves the original structure of the executable slice. The produced slice is a valid and fully executable UCM specification, where all constructs are reachable. This was achieved by handling all effects of removing unrelated elements and un-traversed branches, e.g., preserving the loop structure after removing an OR-Fork. The later example is tricky because the removal of an OR-Fork will break

the loop structure (by definition an OR-Fork must have at least two outgoing branches otherwise [36]). In order to avoid such scenario, the algorithm first checks whether the slicing criterion resides within a loop and whether removing the unrelated branches will cause the loop to be disconnected. If so, the algorithm will create an empty OR-Fork branch, then eliminate the irrelevant OR-Fork branch. Another example of preserving connectivity is when we have to remove model elements past the slicing criterion. The algorithm first cuts the slicing criterion's successor node connection, and then removes all nodes. However, this will produce an invalid UCM, since the map must have an end point attached to the target side of slicing criterion successor link. Therefore, a new end point node is created and linked to the map in order to have a valid UCM slice. Another more complex example of preserving connectivity is when we have to remove model elements between the slicing criterion and an *AND-Join* node, where the slicing criterion is on a concurrent branch. We have to delete path nodes between SC and the *AND-Join* without deleting the concurrent node. This is achieved by cutting the links (successor link of SC and predecessor link of the AND-Join), remove the path nodes in the middle,and finally, reconnect the two links again, such as in Fig. 3.10.

- **Handling concurrency**: Handling concurrency is required since UCM share a global data model among all elements and different execution orders can influence the data and control flow. Slicing concurrent programs or state-based models (SBM) requires the computation of special kind of dependency called interfere dependence [15, 20]. Handling such dependencies is complex and require the order of execution to be taken into account to guarantee precise slices. Our adopted solution consists of examining all possible orders of execution and compute dependencies for each possible sequence, as explained in Sect.3.5. In order to obtain precise slices, the re-computation step is preformed on tree structures of each concurrent branch instead of re-traversing the UCM map.

- **Removal of infeasible scenarios**: The removal of irrelevant UCM parts (with respect to the slicing criterion) may remove start points. The irrelevant start points may be part of predefined scenario definitions (*ScenarioDef*). Once the slice is produced and these start points are deleted, the scenario definitions become infeasible; hence, they are removed.



2 - Normal order (6)
3 - Backordered with patient customer (523)
4 - Backordered items are cancelled (2343)
5 - Cancel whole order because of backorder (505)
6 - Infinite loop: wait but never arrives (525)
Included scenarios
Start points
Order: Start Shopping (8)
Order: Order to process (199)
Order: Product reception (423)
Initializations
Preconditions
End points

Figure 5.1: Ordering model: scenario start points within *InfinitLoop* scenarioDef

For example, the ordering model case study contains six scenario definitions. The sixth scenario definition,*Infinite loop: wait but never arrives* (see Fig. 5.1), contains three scenario start points which reside within Order map in Fig. 4.16: *Start Shopping*, *Order to process*, and *Product to reception.* When generating the executable slices in figures 4.17, 4.19, and 4.18, the scenario start points *Order to process* and *Product to reception* become infeasible and, consequently, they are removed from the slice.

5.2 Limitations

The UCM slicing approach is subject to the following limitations:

- **Irrelevant code statements within RespRefs are not removed**: Our slicing approach considers a respRef relevant to the slicing criterion when there is one or more code statements defining a variable in the *Dependent-Variables* list. However, the slicing approach does not remove the unrelated code statements within related respRefs. The main reason for not removing code statements in respRef's expression is to avoid syntactic errors resulting from the removal of statements contained within if-else blocks. For example, suppose R18 is to be examined during backward traversal, while the selected variable of an SC is *x*. The statements "$x=z$" and "$y=y+1$" are relevant while the rest of code statements is not. However, statements within if-else blocks cannot be removed since it will cause the if-condition to be empty, and the code checker of the framework will trigger a syntax error even when writing symbol ";" instead of the code statements, so R18 is considered relevant to the slice without cleaning the unrelated code statements within its expression. Nevertheless, this limitation can be considered minor since at the requirements level the focus is more on finding relevant/irrelevant model elements, rather than refined code statements.

- **Complexity of concurrency solution**: In order to solve the concurrency challenge where the order of executing concurrent paths can have an impact on dependency, we had to consider all possible sequences and re-calculate dependencies accordingly. This results in precise slices, but it is computationally expensive. We have tried to alleviate this limitation and minimize the cost of re-calculating concurrent branches, as explained in Sect. 3.5.

- **Choice of the slicing criterion:** The slicing criterion (SC) can be chosen from any UCM map and would produce a valid slice. However, if the SC is part of a plugin map, its parent map cannot be traversed because (1) a plug-in may be bound to more than one stub (2) such information can only be known at run time in case we have dynamic or synchronizing stubs.

5.3 Comparison with related work

In what follows, we survey and compare existing model based slicing approaches with respect to the following criteria:

- *Model* refers to the target state-based model.

- *Type* is the type of slicing approach: A-Amorphous, C-Conditioned, D-Dynamic, E-Environment-based, P-Proposition-based, R-Reactive, and S-Static. Definitions of these slicing approaches are explained in [20].
- *Dir* is the direction of traversal where B is forward and F is forward.
- *E/C*: is the type of output slice where E is executable and C is closure slice.
- *Dep*: is the type of computed dependency where D is data flow, and C is control flow. Some approaches support either data flow or control flow, while most of them support both types. There are other types of dependencies, but in this comparison, we focus only on whether or not these two dependencies are supported. In some approaches such as the slicing approach by Ganapathy and Ramesh [64], dependencies are not defined explicitly, they are denoted by "-".
- *Purpose* refers to the objective/use of the approach.

As shown in Table 5.1, our slicing approach (last row in the table), has the advantage of providing both output approaches, executable and closure slice. Moreover, not all approaches provide both data and control flow dependency computations. Another advantage of UCM slicing approach, not mentioned in the table, is producing precise slices while preserving graph connectivity when using executable slice approach. Executable slice involves removal of unrelated parts from the target model. Most approaches in model based slicing only remove the unrelated parts (e.g., states or transitions) that do not impact the reachability of

Approach	**Model**	**Type**	**Dir**	**E/C**	**Dep**	**Purpose**
Korel et al. [24]	EFSMs	S	B	E	D, C	Comprehension
Fox and Luangsodsai [65]	Statecharts	S	B	E	D	Comprehension
Ojala[66]	State machines	S	B	E	D, C	Model Checking
Labbe and Gallois [67]	IOSTs	S	B	C	D, C	Comprehension
Androutsopoulos et al. [68]	EFSMs	S	B	C	D, C	Comprehension
Lano and Rahimi [69]	restricted UML state machines	S	B	E	D	Comprehension
Chan et al.[70]	RSML	P	B	E	D	Model Checking
Wang et al. [71]	EHAs	P	B	E	D, C	Model Checking
Langenhove [72]	EHAs	P	B	E	D, C	Model Checking
Janowska and Janowski [73]	Timed automata	P	B	E	D, C	Model Checking
Colangelo et al. [74]	State machines	P	B	E	-	Model Checking
Ganapathy and Ramesh [64]	Argos, Lustre	R	B	E	-	Comprehension
Guo and Roychoudhury [75]	Java (map tostatecharts)	D	B	E	D, C	Debugging
Korel et al. [24]	EFSMs	A	B	E	D, C	Comprehension
Androutsopoulos et al. [76]	EFSMs	E	F	E	-	Reuse
Lano and Rahimi [69]	restricted UMLstate machines	E	F	E	-	Reuse
Heimdahl and Whalen [23]	RSML	C	B	E	D, C	Comprehension
Bozga et al. [77]	Extended automata	C	B	E	D	Testing
Zhao [28]	ADL(WRIGHT)	S	F,B	E	D,C	Comprehension,Reuse
Stafford et al. [50]	ADL(Rapide,Acme)	S	F,B	E	D,C	Comprehension,Testing
Kim et al. [51]	ADL(Rapide)	D	F	E	D,C	Comprehension
Samuel and Mall [52]	Activity diagrams	D	B	E	D,C	Testing
Ray et al. [53]	Activity diagrams	C	B	E	D,C	Testing
Bouras et al. [54]	Sequence diagrams	S	B	E	-	Merge Diagrams
Lity et al. [55]	Delta-Oriented SPLs	S	B	E	C	Testing
Best and Rakow. [56]	BPM (Petri nets)	S	F	E	-	Testing
Hassine et al. [25]	Use Case Maps (UCM)	S	B	E	C	Comprehension
Our Slicing approach	Use Case Maps (UCM)	S	B	E, C	D, C	Comprehension

Table 5.1: Comparison of model based slicing approaches

other model elements. In other words, the unrelated states/transitions that cause unreachability of other parts are kept in order to preserve graph connectivity. However, this will result in bigger and less precise slices. Only the amorphous slicing approach presented by Korel et al. [24] showed a mechanism to remove unrelated transitions and reconnect state machines in order to preserve model reachability of the produced slice. This was achieved by combining states based on two merging rules. However, these rules cannot be generalized to handle all possible cases. Our slicing approach generate precise executable slices by removing unrelated parts while preserving reachability of model elements.

With respect to model reduction,various model based slicing techniques have been presented in order to assist overall model comprehension, review, or analysis by reducing its size. Heimdahl et al. [78] assessed the slicing effectiveness on TCAS II models, a group of airborne devices used to avoid collision for commercial aircraft protection. It contains 650 transitions and more than 300 states. The model reduction rates found by Heimdahl et al. [78] range from 68% to 90%.

The slicing techniques explained by Androutsopoulos et al. [68] and Korel et al.[24] are developed to improve model comprehension of EFSM specifications by reducing the size of these models via slicing. The empirical results of Androutsopoulos et al. [79], reported 38.42% as the smallest average size of backward slice. Their empirical study covered more than 10 EFSM models and all possible transitions. It should be noted that the slice based on Androutsopoulos et al. [68] approach contains unmarked and marked transitions and the size of the slice, with respect to total the number of states and number of transitions, is not reduced. Korel et al. [24] did not present explicitly the set of used examples with their produced slices, but they stated that using their slicing tool they were able to achieve reduction rates between 55% to 80% when implementing amorphous slicing technique on a number of EFSM specifications. Fox and Luangsodsai [65] and Labbe and Gallois [67] described other slicing techniques to enhance model comprehension. The produced slices were sub-models of the original. However, no data about the size of the slices were provided neither in Fox and Luangsodsai [65] nor in Labbe and Gallois [67].

In Sect. 4.1.3, we calculated the reduction rate of our slicing technique based on 14 slices from three case studies and a Mock system and we found that the average reduction rate is *70%* and UCM slicing algorithm can reduce the model size from *30%* to *94%*. However, compared to the model size used by Heimdahl et al. [78], we still need to test our approach on larger models.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis, we have presented a UCM static slicing technique that would help requirements engineers reduce UCM specifications according to a slicing criteria of interest in order to improve their comprehension of the UCM model. Our approach is implemented within jUCMNav framework, and it produces both closure and executable slices. The proposed approach has been tested and evaluated using three case studies and one mock model. Results showed that UCM slicing approach can reduce UCM specifications up to *93%* and the average reduction rate is *70%*. We showed that UCM slicing approach also has the ability to keep graph connectivity and produces precise slices.

In future work, we will test the effectiveness of our slicing algorithms on larger models. We plan also to investigate the implementation of other UCM-based slicing techniques such as conditioned slicing, and dynamic slicing.

REFERENCES

[1] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, no. 3, pp. 121–189, 1995.

[2] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557

[3] J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering," in *Proceedings of the 15th international conference on Software Engineering.* IEEE Computer Society Press, 1993, pp. 509–518.

[4] Y. Deng, S. Kothari, and Y. Namara, "Program slice browser," in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on.* IEEE, 2001, pp. 50–59.

[5] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 1993, pp. 384–396.

[6] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Software Maintenance, 1992. Proceerdings., Conference on.* IEEE, 1992, pp. 299–308.

[7] S. Horwitz, J. Prins, and T. Reps, "Integrating noninterfering versions of programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 3, pp. 345–387, 1989.

[8] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE transactions on software engineering*, vol. 17, no. 8, pp. 751–761, 1991.

[9] R. Bodik and R. Gupta, "Partial dead code elimination using slicing transformations," in *ACM SIGPLAN Notices*, vol. 32, no. 5. ACM, 1997, pp. 159–170.

[10] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 37–61, 1985.

[11] J. Q. Ning, A. Engberts, and W. Kozaczynski, "Recovering reusable components from legacy systems by program segmentation," in *Reverse Engineering, 1993., Proceedings of Working Conference on.* IEEE, 1993, pp. 64–72.

[12] L. I. Millett and T. Teitelbaum, "Issues in slicing promela and its applications to model checking, protocol understanding, and simulation," *International*

Journal on Software Tools for Technology Transfer, vol. 2, no. 4, pp. 343–349, 2000.

[13] M. Plakal and C. N. Fischer, "Concurrent garbage collection using program slices on multithreaded processors," in *ACM SIGPLAN Notices*, vol. 36, no. 1. ACM, 2000, pp. 94–100.

[14] A. De Lucia *et al.*, "Program slicing: Methods and applications." in *scam*, 2001, pp. 144–151.

[15] J. Silva, "A vocabulary of program slicing-based techniques," *ACM computing surveys (CSUR)*, vol. 44, no. 3, p. 12, 2012.

[16] D. W. Binkley and K. B. Gallagher, "Program slicing," *Advances in Computers*, vol. 43, pp. 1–50, 1996.

[17] M. Harman and R. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.

[18] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[19] R. Singh and V. Arora, "Literature analysis on model based slicing," *International Journal of Computer Applications*, vol. 70, no. 16, 2013.

[20] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt, "State-based model slicing: A survey," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 53, 2013.

[21] B. Vaysburg, L. H. Tahat, and B. Korel, "Dependence analysis in reduction of requirement based test suites," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 107–111, 2002.

[22] A. M. Sloane and J. Holdsworth, "Beyond traditional program slicing," in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3. ACM, 1996, pp. 180–186.

[23] M. P. Heimdahl and M. W. Whalen, "Reduction and slicing of hierarchical state machines," in *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 6. Springer-Verlag New York, Inc., 1997, pp. 450–467.

[24] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, "Slicing of state-based models," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on.* IEEE, 2003, pp. 34–43.

[25] J. Hassine, R. Dssouli, and J. Rilling, "Applying reduction techniques to software functional requirement specifications," in *International Workshop on System Analysis and Modeling.* Springer, 2004, pp. 138–153.

[26] J. Chang and D. J. Richardson, "Static and dynamic specification slicing," in *In Proceedings of the Fourth Irvine Software Symposium.* Citeseer, 1994.

[27] L. I. Millett and T. Teitelbaum, "Slicing promela and its applications to model checking, simulation, and protocol understanding," in *Proceedings of the 4th International SPIN Workshop.* Citeseer, 1998, pp. 75–83.

[28] J. Zhao, "Applying slicing technique to software architectures," in *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on.* IEEE, 1998, pp. 87–98.

[29] B. W. Boehm *et al.*, *Software engineering economics.* Prentice-hall Englewood Cliffs (NJ), 1981, vol. 197.

[30] C. McClure, *The three Rs of software automation: re-engineering, repository, reusability.* Prentice-Hall, Inc., 1992.

[31] B. A. Kitchenham, G. H. Travassos, A. Von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang, "Towards an ontology of software maintenance," *Journal of Software Maintenance*, vol. 11, no. 6, pp. 365–389, 1999.

[32] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[33] R. K. Fjeldstad and W. T. Hamlen, "Application program maintenance study: Report to our respondents," *Proceedings Guide*, vol. 48, 1983.

[34] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[35] Slicing techniques. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/mharman/exe1.html

[36] ITU-T, "Recommendation Z.151 (10/12), User Requirements Notation (URN) language definition, Geneva, Switzerland," Genève,Switzerland, 2012. [Online]. Available: http://www.itu.int/rec/T-REC-Z.151/en

[37] N. Genon, D. Amyot, and P. Heymans, *Analysing the Cognitive Effectiveness of the UCM Visual Notation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 221–240. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21652-7_14

[38] jUCMNav v7.0.0, "jUCMNav Project (tool, documentation, and meta-model)," University of Ottawa, 2016. [Online]. Available: http://softwareengineering.ca/\~jucmnav

[39] D. Amyot and G. Mussbacher, "User requirements notation: The first ten years, the next ten years (invited paper)," *Journal of Software (JSW)*, vol. 6, no. 5, pp. 747–768, 2011.

[40] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *ACM SIGPLAN Notices*, vol. 25, no. 6. ACM, 1990, pp. 246–256.

[41] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(88)90054-3

[42] R. Gupta and M. L. Soffa, "Hybrid slicing: An approach for refining static slices using dynamic information," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, pp. 29–40, 1995.

[43] T. Takada, F. Ohata, and K. Inoue, "Dependence-cache slicing: A program slicing method using lightweight dynamic information," in *Program Comprehension, 2002. Proceedings. 10th International Workshop on.* IEEE, 2002, pp. 169–177.

[44] F. Umemori, K. Konda, R. Yokomori, and K. Inoue, "Design and implementation of bytecode-based java slicing system," in *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on.* IEEE, 2003, pp. 108–117.

[45] G. A. Venkatesh, "The semantic approach to program slicing," in *ACM SIGPLAN Notices*, vol. 26, no. 6. ACM, 1991, pp. 107–119.

[46] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned program slicing," *Information and Software Technology*, vol. 40, no. 11, pp. 595–607, 1998.

[47] D. Jackson and E. J. Rollins, "Chopping: A generalization of slicing," DTIC Document, Tech. Rep., 1994.

[48] T. Reps and G. Rosay, "Precise interprocedural chopping," in *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4. ACM, 1995, pp. 41–52.

[49] A. Rakow, “Safety slicing petri nets,” in *International Conference on Application and Theory of Petri Nets and Concurrency.* Springer, 2012, pp. 268–287.

[50] J. A. Stafford and A. L. Wolf, “Architecture-level dependence analysis in support of software maintenance,” in *Proceedings of the third international workshop on Software architecture.* ACM, 1998, pp. 129–132.

[51] T. Kim, Y.-T. Song, L. Chung, and D. T. Huynh, “Software architecture analysis: a dynamic slicing approach,” *ACIS International Journal of Computer & Information Science*, vol. 1, no. 2, pp. 91–103, 2000.

[52] P. Samuel and R. Mall, “Slicing-based test case generation from uml activity diagrams,” *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 6, pp. 1–14, 2009.

[53] M. Ray, S. S. Barpanda, and D. P. Mohapatra, “Test case design using conditioned slicing of activity diagram,” *International Journal of Recent Trends in Engineering (IJRTE)*, vol. 1, pp. 117–120, 2009.

[54] Z.-E. Bouras and A. Talai, “Software evolution based sequence diagrams merging,” *complexity*, vol. 1, p. 2, 2015.

[55] S. Lity, T. Morbach, T. Thüm, and I. Schaefer, “Applying incremental model slicing to product-line regression testing,” in *International Conference on Software Reuse.* Springer, 2016, pp. 3–19.

[56] E. Best and A. Rakow, "A slicing technique for business processes," in *International United Information Systems Conference.* Springer, 2008, pp. 45–51.

[57] J. Hassine, J. Rilling, J. Hewitt, and R. Dssouli, "Change impact analysis for requirement evolution using use case maps," in *Eighth International Workshop on Principles of Software Evolution (IWPSE'05).* IEEE, 2005, pp. 81–90.

[58] S. Lity, H. Baller, and I. Schaefer, "Towards incremental model slicing for delta-oriented software product lines," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* IEEE, 2015, pp. 530–534.

[59] A. Pourshahid, D. Amyot, L. Peyton, S. Ghanavati, P. Chen, M. Weiss, and A. J. Forster, "Business process management with the user requirements notation," *Electronic Commerce Research*, vol. 9, no. 4, pp. 269–316, 2009.

[60] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction.* Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[61] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[62] B. Kitchenham, S. L. Pfleeger, L. Pickard, P. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in

software engineering," *IEEE Trans. Software Eng.*, vol. 28, no. 8, pp. 721–734, 2002. [Online]. Available: http://dx.doi.org/10.1109/TSE.2002.1027796

[63] A. Jedlitschka and D. Pfahl, "Reporting guidelines for controlled experiments in software engineering," in *2005 International Symposium on Empirical Software Engineering (ISESE 2005), 17-18 November 2005, Noosa Heads, Australia*, 2005, pp. 95–104. [Online]. Available: http://dx.doi.org/10.1109/ISESE.2005.1541818

[64] V. Ganapathy and S. Ramesh, "Slicing synchronous reactive programs," *Electronic Notes in Theoretical Computer Science*, vol. 65, no. 5, pp. 50–64, 2002.

[65] C. Fox and A. Luangsodsai, "And-or dependence graphs for slicing statecharts," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[66] V. Ojala, *A slicer for UML state machines*. Citeseer, 2007.

[67] S. Labbé and J.-P. Gallois, "Slicing communicating automata specifications: polynomial algorithms for model reduction," *Formal Aspects of Computing*, vol. 20, no. 6, pp. 563–595, 2008.

[68] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, "Control dependence for extended finite state machines," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 216–230.

[69] K. Lano and S. K. Rahimi, “Slicing techniques for uml models,” *Journal of Object Technology*, vol. 10, no. 11, pp. 1–49, 2011.

[70] W. Chan, R. J. Anderson, P. Beame, and D. Notkin, “Improving efficiency of symbolic model checking for state-based system requirements,” in *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 2. ACM, 1998, pp. 102–112.

[71] W. Ji, D. Wei, and Q. Zhi-Chang, “Slicing hierarchical automata for model checking uml statecharts,” in *International Conference on Formal Engineering Methods*. Springer, 2002, pp. 435–446.

[72] S. Van Langenhove, “Towards the correctness of software behavior in uml: A model checking approach based on slicing,” Ph.D. dissertation, Ghent University, 2006.

[73] A. Janowska and P. Janowski, “Slicing of timed automata with discrete data,” *Fundamenta Informaticae*, vol. 72, no. 1-3, pp. 181–195, 2006.

[74] D. Colangelo, D. Compare, P. Inverardi, and P. Pelliccione, “Reducing software architecture models complexity: A slicing and abstraction approach,” in *International Conference on Formal Techniques for Networked and Distributed Systems*. Springer, 2006, pp. 243–258.

[75] L. Guo and A. Roychoudhury, “Debugging statecharts via model-code traceability,” in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2008, pp. 292–306.

[76] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li, “Model projection: Simplifying models in response to restricting the environment,” in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 291–300.

[77] M. Bozga, J.-C. Fernandez, and L. Ghirvu, “Using static analysis to improve automatic test generation,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2000, pp. 235–250.

[78] M. P. E. Heimdahl, J. M. Thompson, and M. W. Whalen, “On the effectiveness of slicing hierarchical state machines: A case study,” in *Euromicro Conference, 1998. Proceedings. 24th*, vol. 1. IEEE, 1998, pp. 435–444.

[79] K. Androutsopoulos, N. Gold, M. Harman, Z. Li, and L. Tratt, “A theoretical and empirical study of efsm dependence.” in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference*. IEEE Computer society, 2009, pp. 287–296.

Vitae

- Name: Taha Hussein Naji Binalialag
- Nationality: Yemeni
- Date of Birth: 11/10/1986
- Email: *tahahussein515@gmail.com*
- Permenant Address: Al Khobar, Eastern Region, Saudi Arabia
- Academic Background:
  - M.S in Software Engineering, May 2017

    King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.
  - B.S in Computer Information Systems, July 2010

    Al-ahgaff University, Hadramout, Yemen.
- Publications: Taha Binalialhag, Jameleddin Hassine, Daniel Amyot, Applying Static Slicing to UCM Requirements specifications (In preparation).
- Research Interests: Software Engineering, Requirements Engineering.