FormlSlicer: A Model Slicing Tool for Feature-rich State-machine Models

by

Xiaoni Lai

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

© Xiaoni Lai 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A model of the feature-oriented requirements of a software system usually contains a large number of non-trivial features; each feature may have unintended interactions with other features. It may be difficult to comprehend or verify such a model. Model slicing is a useful approach to overcome such a challenge by enabling views of models of individual features that preserve feature interactions. Model slicing evolves from traditional program slicing; it is a technique to extract a sub-model from the original model with respect to a slicing criterion. In this thesis we focus on one type of model: state-based models (SBMs). Because of the difference in granularity between programs and SBMs, as well as the difficulty of maintaining well-formedness of a sliced SBM, SBM slicing is much more challenging than program slicing. Among a diverse range of slicing approaches, dependence-based slicing is the most popular; it relies on the computation of dependence relations among states and transitions in order to determine which model elements of the original model must be in the slice and which can be omitted.

We present a workflow and tool for automatically constructing a feature-based slice from a feature-oriented state-machine model of the requirements of a software system. Each feature in the model is modeled as a complete state-transition machine called a feature-oriented state machine (FOSM). The workflow consists of two tasks—a preprocessing task and a slicing task. The preprocessing task mainly computes three types of dependences: hierarchy dependence (HD), which represents the state hierarchy relation among states in the original model; data dependence (DD), which captures the define-use relationship among transitions with respect to a variable; and control dependence (CD), which captures the notion of whether one state can affect the execution of another state or transition. The slicing task forks off multiple slicing processes; each process considers one of the FOSMs as the feature of interest (FOI)—which is the slicing criterion—and the rest of FOSMs as the rest of the system (ROS)—which is to be sliced. Each slicing process constructs a sliced model to preserve the portion of the ROS that interacts with the FOI. The construction process is multi-staged; it firstly identifies an initial set of transitions in the ROS that directly affect the FOI; it then finds more states and transitions in the transitive closure of dependences; and it eventually restructures the model to further reduce the model size and maintain its well-formedness property.

We provide a correctness proof that shows that the resulting sliced models simulate the original model, by proving that an execution step of a given execution trace in the original model can always be projected to an execution step of at least one execution trace in the sliced model.

Our proposed slicing workflow has been implemented in a tool called FormlSlicer. We conducted an empirical evaluation that demonstrates that, on average, the ROS of a sliced model has 23.0% of states, 15.7% of transitions, 32.8% of regions and 19.3% of variables of the ROS of the original model.

Acknowledgments

I would like to thank my supervisor, Joanne Atlee, who has given me the opportunity to work on this interesting thesis project and has provided me a lot of guidance along the way. I would also like to thank my colleagues, Sandy Beidu and Hadi Zibaeenejad, for giving me enlightenment over some challenges in the project. Lastly I would like to thank my husband, Qiang, for his encouragement and support.

Table of Contents

List of Tables x

List of Figures xi

List of Algorithms xiv

Glossary xv

1 Introduction 1

1.1 Model Slicing . . . 1

1.1.1 What Is Model Slicing . . . 1

1.1.2 Properties of a Useful Sliced Model . . . 2

1.2 Feature-oriented Model Slicing . . . 3

1.2.1 Feature Interactions in Feature-rich System Requirements . . . 4

1.2.2 Model Slicing used in Feature-oriented Requirements in Software Systems . . . 5

1.3 Thesis Overview . . . 5

1.3.1 Thesis Statement . . . 7

1.4 Chapter Summary . . . 8

2 Related Work **9**
2.1 Program Slicing 9
2.1.1 Dependence-based Slicing 10
2.2 Slicing on State-based Models (SBMs) 11
2.2.1 What Is an SBM 11
2.2.2 Challenges of SBM Slicing 13
2.2.3 Relevant SBM Slicing Techniques 14
2.2.4 Dependences in SBM Slicing 14
2.3 Correctness of Slices 19
2.3.1 Simulation in Programs 19
2.3.2 Simulation in SBMs 20
2.4 Cone of Influence Reduction 21

3 Preliminaries **23**
3.1 Terminology 23
3.2 What Is FORML 24
3.3 Scope of FormlSlicer 26

4 FormlSlicer **30**
4.1 Overview of FormlSlicer's Workflow 30
4.2 Preprocessing: Model Parsing and Conversion from FORML to CFGs . . . 31
4.2.1 Transformation of Transition Labels 32
4.2.2 Control Flow Graph (CFG) 34
4.3 Preprocessing: Dependence Analyses 35
4.3.1 Hierarchy Dependence 35
4.3.2 Data Dependence 36
4.3.3 Control Dependence 39
4.4 Multi-Stage Model Slicing Process 49

4.4.1 Initiation Stage 51
4.4.1.1 Variable Extraction Step 51
4.4.1.2 Initial Transition Selection Step 51
4.4.2 General Iterative Slicing Stage 52
4.4.2.1 DD Step 53
4.4.2.2 Cross-Hierarchy Transition Step 55
4.4.2.3 Transition-to-State Step 57
4.4.2.4 CD-HD Step 57
4.4.3 Model Restructuring Stage 59
4.4.3.1 State Merging Step 60
4.4.3.2 State Connecting Step 61
4.4.4 More Examples 64
4.4.5 Summary 66

5 Correctness of FormlSlicer 68

5.1 Overview 68
5.2 Terminology 69
5.2.1 Variable, State, Region, Transition and Model 69
5.2.2 State Configuration and Interpretation 71
5.2.3 Dependences 72
5.2.4 Execution Step 72
5.3 State Transition Rule 74
5.4 FormlSlicer's Multi-Stage Model Slicing Process 75
5.4.1 Definitions 76
5.4.2 Multi-Stage Model Slicing Process 76
5.5 Proof 78
5.5.1 Projection of Snapshot in the Original Model to Snapshot in the Sliced Model 79

5.5.2 Projection of One Transition in the Original Model to Epsilon or One Transition in the Sliced Model 80

5.5.3 Projection of One Execution Step in the Original Model to One Execution Step in the Sliced Model 88

5.5.4 Simulation 89

6 Empirical Evaluations of FormlSlicer 92

6.1 Choosing a Model for Empirical Evaluation 92

6.2 Reduction of Model Size 94

6.3 Properties of a Useful Sliced Model 96

7 Conclusion 97

7.1 Summary of Thesis and Contributions 97

7.1.1 Contributions 98

7.2 Future Work 100

7.2.1 Extending to Slicing on Software-Product-Line Model 100

7.2.2 Bridging the Gap between FormlSlicer's Input Model and FORML model 100

7.2.3 Improving the Workflow 101

7.2.4 Customization of Slicing in FormlSlicer 102

7.2.5 Making the Correctness Proof More Rigorous 102

Appendix A Automotive: A Slicing Example 104

Appendix B Supporting Functions for Control Dependency Algorithm 119

References 122

List of Tables

4.1 Format of an Input/Output File to Specify one FOSM 31

4.2 How FormlSlicer extracts Monitored Variables from WCEs 33

4.3 How FormlSlicer extracts Monitored/Controlled Variables from WCAs . . . 33

4.4 How FormlSlicer extracts Monitored Variables from Transition Guard Conditions 34

4.5 List of Supporting Functions for Main Algorithm in Algorithm 4.2 47

4.6 The Monitored and Controlled Variables of Transitions in the Running Example 50

4.7 The Monitored and Controlled Variables of More Transitions in the Example 65

6.1 Empirical Results of FormlSlicer on the Automotive Case Study 95

List of Figures

1.1 A comparison of the model before and after slicing by FormlSlicer 6

2.1 An example program 10

2.2 A simple CFG with a single end-node e 16

3.1 The structure of a state machine in FORML's behavior model 25

3.2 An example model 27

3.3 Two types of transitions (shown in blue) not allowed in FormlSlicer 28

3.4 Restrictions and solution on the arrow from a pseudo state to default initial state 28

4.1 Overview of FormlSlicer's workflow: the preprocessing task and the slicing task 31

4.2 A simple input example 32

4.3 The class hierarchy of Node, TNode and SNode in CFG 34

4.4 Several CFGs for the FORML model in Figure 4.2b. Yellow nodes are SNodes. 35

4.5 Hierarchy dependence 35

4.6 Data dependence between t_k and t_1 w.r.t. v 36

4.7 The "inState()" expression as a variation of data dependence 37

4.8 An illustration of non-termination sensitive control dependence: n_j is control dependent on n_i 40

4.9 A CFG example to illustrate the paths from Node 1 to Node 5: Node 8 has two outgoing nodes (9 and 10), highlighted in orange color; Node 1 has three outgoing nodes (2, 7 and 11), highlighted in green color. 41

4.10 The paths representation from Node 1 to Node 5 in the CFG in Figure 4.9 42
4.11 Two cases in propagating the paths representation from `currNode` to its neighbor in Algorithm 4.2 46
4.12 A comparison between the original model and the sliced model 49
4.13 The model used as the running example 50
4.14 The control flow graph of the model used as the running example 51
4.15 The sliced ROS after performing the initial transition selection step 52
4.16 Four steps in general iterative slicing stage 53
4.17 The sliced ROS after performing the DD step 55
4.18 Examples of cross-hierarchy transitions 56
4.19 The Example FOSM in the ROS after Transition-to-State Step 58
4.20 The sliced ROS after performing the CD-HD step 59
4.21 Illustration of state merging rules 60
4.22 The sliced ROS after performing the stage merging step 62
4.23 The sliced ROS after the state connecting step 63
4.24 The example model with more contents 64
4.25 The sliced example model w.r.t. *E1* 65

5.1 The FORML example from Figure 3.1 with its current states highlighted in pink 71
5.2 Concurrency in orthogonal regions as an execution step (only blue colored components are relevant in the execution) 73
5.3 Projection of a transition from the original model to the sliced model . . . 80
5.4 The decision tree for case-based analysis in projection of transitions 82
5.5 The simulation relation between the execution traces of M and $M_{\mathcal{L}}$ 91

6.1 A feature model that constraints the relationships between features in *Autosoft* 93

A.1 The original automotive model 105
A.2 Original adaptive cruise control (ACC) feature of the automotive model . . 106

A.3 Original forward collision alert (FCA) feature of the automotive model . . 107
A.4 Original lane change alert (LCA) feature of the automotive model 108
A.5 Original lane centring control (LCC) feature of the automotive model . . . 109
A.6 Original speed limit control (SLC) feature of the automotive model 110
A.7 Original air quality system (AQS) feature of the automotive model 111
A.8 Original air conditioning (AC) feature of the automotive model 112
A.9 The sliced model w.r.t. LCA 113
A.10 Sliced ACC feature w.r.t. LCA 113
A.11 Sliced FCA feature w.r.t. LCA 114
A.12 Sliced LCC feature w.r.t. LCA 114
A.13 Sliced LCC feature w.r.t. LCA 114
A.14 The Sliced Model w.r.t. ACC 115
A.15 Sliced FCA feature w.r.t. ACC 116
A.16 Sliced LCA feature w.r.t. ACC 116
A.17 Sliced LCC feature w.r.t. ACC 117
A.18 Sliced SLC feature w.r.t. ACC 118

List of Algorithms

4.1 Algorithm of Data Dependence Computation used in FormlSlicer 38
4.2 Main Algorithm of Control Dependence Computation 45
4.3 DD step in general iterative slicing stage 54
4.4 CD-HD step in general iterative slicing stage 59

B.1 HasNonEmptyPathsFromNode1 119
B.2 IsControlDependentOn1 119
B.3 ReducePaths 120
B.4 UnionPath 121
B.5 ExtendPath 121

Glossary

AC air conditioning. xiii, 90, 92, 101, 109

ACC adaptive cruise control. xii, xiii, 89, 92, 101, 103, 110, 112–115

AQS air quality system. xiii, 90, 92, 101, 108

BDS basic driving service. 24, 89, 90

CC cruise control. 24, 89, 90

CD control dependence. iii, 10, 13, 14, 36, 71, 98

CFG control flow graph. 9, 14, 16, 17, 31–33, 36–38, 41, 45, 47, 94, 96

DD data dependence. iii, 10, 13, 33, 71, 96

EFSM extended finite-state machine. 2, 10, 13, 16, 18

FCA forward collision alert. xiii, 89, 92, 101, 104, 111, 113

FOI feature of interest. iii, 5, 6, 27, 33, 46–48, 62, 63, 65, 73, 91, 92, 94, 101

FORML Feature-Oriented Requirements Modeling Language. 6, 7, 11, 20–24, 29, 30, 32, 34, 36, 88–90, 94, 96, 97

FOSD feature-oriented software development. 3, 4, 21

FOSM feature-oriented state machine. iii, 5, 6, 24, 28, 31, 33, 46–48, 62, 63, 65, 68, 89, 90, 92, 97, 99, 101

HC headway control. 4, 89

HD hierarchy dependence. iii, 33, 71, 95

LCA lane change alert. xiii, 89, 92, 101, 105, 110, 111, 113

LCC lane centring control. xiii, 89, 92, 101, 106, 111, 114

NTSCD non-termination sensitive control dependence. 16, 36, 37, 40, 41, 96

ROS rest of the system. iii, iv, 5, 6, 27, 46–49, 61–63, 65, 91, 92, 101

SBM state-based model. iii, 2, 3, 5, 6, 10, 11, 13, 23, 29, 36, 65, 94, 97

SLC speed limit control. xiii, 4, 90, 92, 101, 107, 115

SNode A Node in CFG representing a state in FORML model. 31, 46, 54

TNode A Node in CFG representing a transition in FORML model. 31, 34, 46, 54

WCA world change action. 23, 29, 30, 96

WCE world change event. 22, 23, 29, 30, 96

Chapter 1

Introduction

1.1 Model Slicing

1.1.1 What Is Model Slicing

In English, “slicing” usually refers to the act of cutting a portion from something larger using a sharp implement. In the software engineering research community, the word was first adopted to refer to a reduction of a software program to its minimal set of variables and program statements that preserves a subset of the program’s behavior [1]. Since then, program slicing has become a well-investigated topic; multiple notions of program slices have been proposed, as well as different methods to compute them [2]. In general, all of these methods try to extract all parts of an original program that may influence a particular variable of interest. After more than thirty years of development, program slicing has gradually become a mature source-code analysis and manipulation technique and is used in software debugging, software maintenance, optimization, program analysis and information control flow.

As software production nowadays becomes more sophisticated, models used in software specification and design are becoming unwieldy in scale. In recent years, researchers have begun to consider adapting program slicing to apply to models. Similar to program slicing, model slicing extracts a sub-model from an original model while preserving some properties or some behaviors of interest. Because of the graphical nature and other features of models that are different from software programs, model slicing needs to be considered as a distinct research area.

Among a diverse range of software modeling languages, state-based model (SBM) receives an abundant amount of attention from researchers. SBM is an umbrella term for a wide-range of related languages (e.g., extended finite-state machines (EFSMs), statecharts, UML state machines, etc.) [3]. These models are based on finite-state machine formalisms; they depict a set of execution sequences. There are many variants; but in general, an SBM consists of a finite set of states, a set of transitions that move from one state to another state triggered by an event. Many languages have defined additional features, such as global variables, hierarchical constructs, or concurrency constructs.

This thesis concerns model slicing of state-based models. In general, it is a procedure to reduce **an original model** to a smaller one, called **the sliced model**, so that the sliced model contains fewer transitions, states, or other model elements than the original model. At the same time, the sliced model preserves all behaviors of the original model with respect to a **slicing criterion** but may omit other behaviors and details. Usually, the slicing criterion refers to a set of relevant variables. The resulting sliced model, despite being smaller and missing details, produces the same outputs for the relevant variables as the original model does.

1.1.2 Properties of a Useful Sliced Model

We believe that a useful sliced model should possess the following properties[1]:

correctness
: The sliced model simulates the original model with respect to the slicing criteria;

precision
: The sliced model is supposed to retain the information in the original model as much as possible;

reduction
: The sliced model is supposed to be as smaller as possible than the original model.

The property of correctness is non-negotiable. The slicer is incorrect if it produces a sliced model which cannot simulate the original model with respect to the slicing criteria. Simulation is a relation proposed by Milner [26] that relates a structure (e.g., the original program, the original model) to an abstraction of the structure (e.g., the sliced program, the

[1] We summarize these based on information learnt from various literature on model slicing. See Section 2.3.2 for the literature survey.

sliced model). Simulation guarantees that every behavior of a structure is also a behavior of its abstraction; but the abstraction may have behaviors that are not possible in the original structure [26, 4]. In other words, if the original model has a certain execution trace with respect to the slicing criteria, then the sliced model must have a matching execution trace if it is correct[2].

Precision is different from correctness. Consider all the execution traces of an original model as Set A and all the execution traces of its sliced model as Set B. This sliced model is correct when Set A is a subset of Set B. The less difference Set A and Set B have, the more precise the sliced model is.

Sometimes, the properties of precision and reduction oppose each other. If a model slicer reduces the original model aggressively in order to make it as small as possible, it risks sacrificing precision. If a model slicer retains too many details of the original model, it risks sacrificing reduction.

For slicing on an SBM, if we maximize the degree of reduction and minimize the degree of precision, we may obtain a single "super" state in the sliced model such that all relevant transitions become self-looping transitions of the "super" state [6]. If, on the other hand, we minimize the degree of reduction and maximize the degree of precision, we may obtain a sliced model that is exactly the same as the original model. Either of these two sliced models are correct[3], but engineers would gain no useful information from either of these two sliced models.

Therefore, a sliced model is useful when it has appropriate trade-off between the properties of precision and reduction, while it maintains the correctness property. Depending on the situation (that whether one favors reduction over precision in the sliced model or vice versa), a good model slicer is supposed to strike a balance between the two properties. In this thesis, we favors reduction over precision; but we still adopts some existing slicing techniques to preserve a certain degree of precision in the slices.

1.2 Feature-oriented Model Slicing

In this thesis, we apply the SBM slicing techniques on feature-rich state-machine models. These models represent the requirements for feature-rich software systems.

[2]Some may define correctness in a stronger sense such that not only must the sliced model simulate the original model, but also the original model must simulate all observable actions of the sliced model [5]. In this thesis, we will only enforce the basic correctness property.

[3]The sliced model with a "super" state is correct because the set of all its possible execution traces is a superset of the set of all possible execution traces in the original model.

This section will present some background about feature-oriented requirements of software systems, as well as feature interactions that exist in most feature-rich systems. Then it will discuss the importance and challenges of detecting unintended feature interactions and how model slicing is useful in overcoming the challenges.

1.2.1 Feature Interactions in Feature-rich System Requirements

In feature-oriented software development (FOSD), a system's functionality is decomposed into features, where each *feature* is an identifiable unit of functionality or variation. By applying the FOSD paradigm in the requirements of feature-rich software systems, each feature can be represented as a grouping or modularization of individual requirements within the system specification [7]. The requirements model of a feature-rich system can be modeled as a finite set of feature modules, each representing one feature's requirements.

However, features tend not to be completely separate concerns—they operate in a shared context, read and write to the same variables, and affect each other's behavior. One well-known challenge of applying the FOSD paradigm is managing feature interactions. When many features are added to a system, different features can influence one another in determining the overall properties and behaviors of the overall system [8].

Certain feature interactions are *intended*. For example, the call waiting feature of a telephone service is designed to override the basic call service feature's treatment of incoming calls when the subscriber is busy; the engineers who model the call waiting feature understand this intention and handle the feature interaction with care.

On the other hand, certain feature interactions are *unintended*. These unintended feature interactions can cause unexpected behavior of the system and pose potential hazards to the system's users. Consider the headway control (HC) feature and the speed limit control (SLC) feature: if the vehicle approaches an obstacle at a speed faster than the speed limit, both features will simultaneously send messages to the actuators responsible for controlling vehicle acceleration. If their messages are different and not coordinated, the behavior of the vehicle is undefined and the acceleration may be set to an unpredictable value [9].

Features are usually modeled in isolation by different groups of software engineers; as a result, it is likely that there are unintended feature interactions. It is much better to detect and resolve these unintended feature interactions at a requirements stage, rather than at an implementation stage, when the cost for finding the interactions is substantially higher. However, in the industrial setting, there can be hundreds of non-trivial features in a system; thus, complete models of feature-rich software systems can be too large. Engineers

will be overwhelmed if they try to comprehend the whole model, as there is a fundamental limitation of human capacity to deal with such a highly complex model because there are far too many details for a single person to keep track of at once [10].

1.2.2 Model Slicing used in Feature-oriented Requirements in Software Systems

Model slicing is a useful technique to simplify the model to be analyzed for feature interactions. In feature-oriented analysis, we want to examine one feature and its correctness properties with its environment; model slicing can extract the portion of a model representing one feature's environment.

Specifically, slicing can be performed on a model with respect to a single feature, called the feature of interest (FOI), so that the rest of the huge model can be reduced to a feasible size while still preserving the behavior of the FOI. The slicing criterion is the set of variables related to the FOI. All the other features in the model form the rest of the system (ROS) in the model; only a relevant portion of the ROS will be kept after slicing.

Feature-oriented model slicing can improve model comprehension or make model-analyses more feasible. Because it removes irrelevant details of other features in the ROS and only keeps the portions that may (potentially) interact with the FOI, engineers can focus on understanding the FOI together with a smaller ROS, making it easier to identify any unintended feature interactions between the FOI and other features. Similarly, model-analyses can be performed on many smaller model slices in parallel instead of a huge original model; they can produce complete analyses results because feature interactions are taken into account.

Thus, with the help from model slicing, the complex task of detecting unintended feature interactions in a huge model is decomposed into many smaller tasks, where each task focuses on detecting unintended feature interactions between one feature and the others.

1.3 Thesis Overview

We are interested in creating feature-based slices of software models, in particular state-based models (SBMs), to facilitate human understanding of the models and to support feature-based analyses of the models. To be specific, we assume the existence of a feature-oriented state-machine model of a software system, in which each feature is modeled as a

distinct sub-machine (called a feature-oriented state machine (FOSM)) running in parallel with other features' sub-machines. We create *feature-based model slices* (one per feature) that each use a particular feature (called the feature of interest (FOI)) as the slicing criterion, and construct slices of the other features' sub-machines (called the rest of the system (ROS)) omitting details that are irrelevant to the behavior of the FOI. Some features in the ROS are entirely absent in the sliced model, and others are partially absent (see Figure 1.1).

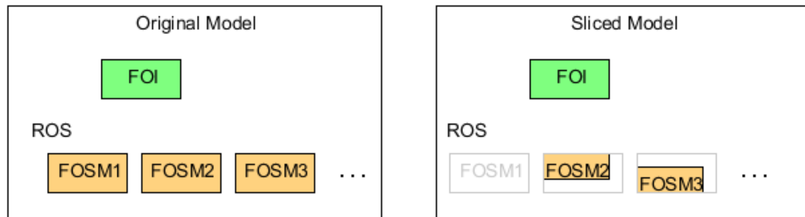


Figure 1.1: A comparison of the model before and after slicing by FormlSlicer

Among different standard software-engineering modeling languages, we choose Feature-Oriented Requirements Modeling Language (FORML) [11] to perform slicing. It is a language that can specify feature-oriented requirements of a software system. Because the syntax of its feature modules is based on UML state machines, and therefore we can adapt many existing slicing techniques on state-based model (SBM) to apply on it. The slicer we have designed and implemented, called FormlSlicer, operates on models written in the FORML, but other than the parser there is nothing that is FORML-specific about our slicing workflow or algorithms.

The workflow consists of two tasks: a preprocessing task and a slicing task. The preprocessing task parses an original model, computes three types of dependences and stores the generated results in different tables. The slicing task forks off multiple slicing processes; each process considers one of the FOSMs as the FOI—which is the slicing criterion—and the rest of FOSMs as the ROS—which is to be sliced. Each slicing process constructs a sliced model to preserve the portion of the ROS that interacts with the FOI. The construction process is multi-staged; it firstly identifies an initial set of transitions in the ROS that directly affect the FOI; it then finds more states and transitions in the transitive closure of dependences; and it eventually restructures the model to further reduce the model size and maintain its well-formedness property.

The main contributions (Chapter 7 will discuss the contributions in details) of this thesis are as follows:

- We adapt traditional definitions and algorithms of program data dependences and state-machine data dependences to apply to SBMs that are hierarchical, have transitions that cross state-hierarchy boundaries, and that have rich transition expressions (including “inState()” expressions).
- We designed and implemented a novel slicing algorithm that employs both data and control dependences, and that *constructs* a model slice from the relevant model elements and then enriches the model slice until its states preserve the reachability properties of the original model.
- Our approach employs a novel decomposition of dependence analyses and slicing tasks that enables parallel construction of multiple model slices.
- Our slicer works on a hierarchical, concurrent and non-terminating state-machine models that have cross-hierarchy transitions, whilst existing slicers only tackle with these modeling constructs partially.
- We prove (by simulation) that all behaviors in the original model are preserved in the sliced model. To our knowledge, this is the first proof of correctness of a construction-based slicer (vs. a slicer that forms a model slice by simply removing model elements from the original model).
- We report the results of a small empirical study in which the slicer was applied to a FORML model comprising seven features.

1.3.1 Thesis Statement

We can slice a feature-oriented requirements model, which consists of many FOSMs, with respect to one FOSM (i.e., the FOI), by performing the following two tasks:

- A preprocessing task, that parses the model and computes dependences,
- A slicing task, that gradually adds model elements from the original model into the sliced model and restructures it in the end;

In order to produce a well-formed sliced model that

- Simulates the original model with respect to the FOI,

- Achieves a satisfactory[4] degree of reduction and precision.

1.4 Chapter Summary

The thesis explains all of the work in creating FormlSlicer. Chapter 2 shows the literature survey on the concepts and techniques related to model slicing. Chapter 3 explains the semantics of FORML, as well as some preliminary knowledge about FormlSlicer. Chapter 4 describes the design and implementation of FormlSlicer. Chapter 5 presents a theoretical evaluation of FormlSlicer, in which we prove the correctness of FormlSlicer by showing how a sliced model produced by FormlSlicer can simulate its original model. Chapter 6 presents an empirical evaluation of FormlSlicer to show that the sliced models are smaller than the original model. Chapter 7 concludes the thesis and shows the contributions, challenges and possible extensions of this research work.

[4] As mentioned in Section 1.1.2, a good model slicer is supposed to strike a balance between the two properties of reduction and precision; in this thesis, we favors reduction over precision but we will use some existing techniques that preserve a certain degree of precision in the slices to prevent producing a over-minimized sliced model.

Chapter 2

Related Work

This chapter presents the related work. Section 2.1 presents a brief account of program slicing, from which model slicing evolves. Section 2.2 presents the related work of model slicing; Subsection 2.2.1 presents the history and different variants of state-based models (SBMs); Subsection 2.2.2 explains the challenges faced by SBM slicing and how the existing techniques tackle them; Subsection 2.2.3 summarizes two general slicing approaches that are in the literature; Subsection 2.2.4 elaborates different types of dependences used in SBM slicing. Section 2.3 discusses how correctness is defined in slicing. Section 2.4 adds some extra discussions about cone of influence reduction, which has a similar idea of dependence-based model slicing.

2.1 Program Slicing

To understand model slicing, it is best to understand first the notion of program slicing from which model slicing evolves.

The notion of a program slice was introduced by Weiser [1]. He defined a program slice S as a reduced, executable program obtained from a program P, such that S replicates part of the behavior of P. Later, many slightly different notions of program slices have been proposed. The general notion of program slice has been summarized in a survey paper by Tip [2]: the program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is called a *slicing criterion*, and is typically specified by a pair (program point, set of variables). The sub-program that has direct or indirect influence on a slicing criterion C is called a

program slice with respect to criterion C. The task of computing program slices is called *program slicing*.

```
1  read(n);
2  i := 1;
3  sum := 0;
4  product := 1;
5  while i <= n do
   begin
6    sum := sum + i;
7    product := product * i;
8    i := i + 1
   end;
9  write(sum);
10 write(product);
```

(a) The original program

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product);
```

(b) Program slice w.r.t. (10,product)

Figure 2.1: An example program

Figure 2.1 is an example adapted from Tip's survey paper on program slicing [2]. The original program (Figure 2.1a computes both the sum and the product of the first n positive numbers. Its program slice with respect to a slicing criterion of (10,product) is shown in Figure 2.1b. We can see that all computations that are irrelevant to the value of variable `product` at Line 10 have been "sliced away".

Program slicing has been further divided into different categories. There is a distinction between static slicing and dynamic slicing. The former is computed without making assumptions regarding a program's input, whereas the latter relies on some selected variables and inputs [12]. Another distinction is made between forward slicing and backward slicing. Informally, a forward slice consists of all statements and control predicates dependent on the slicing criterion, i.e., the program subset that is influenced by the slicing criterion; a backward slice is the program subset that may affect the slicing criterion. In this thesis, we are only interested in static slicing and backward slicing.

2.1.1 Dependence-based Slicing

Slices are computed by finding consecutive sets of transitively relevant statements, according to data flow and control flow dependences. This is generally referred to as dependence-

based slicing [13], because it involves the computation of dependence relations between the program statements. Using dependence relations information, slicing can be reduced to a simple reachability problem.

There are different computation approaches based on different graph representations of a program. Control flow graph (CFG) is a graph representation to capture the reachability of program statements in a program. It is a directed graph, usually seen in compiler analysis to represent all execution paths through a program during its execution [14]. A CFG contains a node for each statement and control predicate in the program; an edge from node i to node j indicates the possible flow of control from the former to the latter. Dependences are defined in terms of the CFG of a program.

Dependences arise as the result of two separate effects [15]:

1. First, a dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed. In the example program in Figure 2.1a, if Line 10 appears before Line 4, the value of `product` written will be incorrect at Line 10. Therefore we know that the statement at Line 10 is dependent on the statement at Line 4. This is data dependence (DD).

2. Second, a dependence exists between a statement and the predicate whose value immediately controls the execution of the statement. In the example program in Figure 2.1a, the statement at Line 7 is dependent on the predicate at Line 5 since the condition in the while loop at Line 5 determines whether the statement at Line 7 is executed. This is control dependence (CD).

2.2 Slicing on State-based Models (SBMs)

As software production nowadays becomes more sophisticated, models used in software specification and design are becoming unwieldy in scale. In recent years, researchers have begun to consider adapting program slicing to apply to models [3]. Similar to program slicing, model slicing extracts a sub-model from an original model while preserving some properties or some behaviors of interest.

2.2.1 What Is an SBM

Among a diverse range of software modeling languages, state-based model (SBM) receives an abundant amount of attention from researchers. SBM is an umbrella term for a wide-

range of related languages (e.g., extended finite-state machines (EFSMs), statecharts, UML state machines, etc.) [3]. These models are based on finite-state machine formalisms; they depict a set of execution sequences. There are many variants; many languages have defined additional features, such as global variables, hierarchical constructs, or concurrency constructs [16].

The basic definition of an SBM comes from Mealy machine [17]. In general, an SBM consists of a finite set of states (including default initial states), a set of events (or "inputs") and a transition function that determines the next state based on the current state and event. Each transition has a source state, a destination state and a label. The label is of the form *e[g]/a*, where each part is optional: e is the event necessary to trigger a possible change of state; g is the guard (i.e., a boolean expression) that further constraints a possible change of state; and a is a sequence of actions (mostly updates to variables in the environment or generations of events) to be executed when the transition occurs.

The notion of an SBM with hierarchy and concurrency constructs has been introduced long time ago [16]. In a hierarchical state machine, a state may be further refined into another sub-machine; this is a composite state. The hierarchy can be arbitrarily deep. If the state machine incorporates concurrency construct into its semantics, a state can consist of multiple orthogonal regions, each containing a sub-machine; the sub-machines are executing concurrently. Researchers have defined that the states in a sub-machine follow an XOR relationship (i.e., it can be in exactly one state at a time) and the orthogonal regions in a state follow an AND relationship (i.e., when the system is in the state, it must be in all of its containing regions) [16]. Communication between concurrent SBMs can be synchronous (i.e., the SBM blocks until the receivers consume the event) or asynchronous (non-blocking).

More advanced constructs have been added to basic SBM languages to augment their expressive power. These include:

- Global variables: a set of variables in the environment that can be read or written by the SBM;
- Parameterized events: triggering events that come with parameters, like functions in a program;
- Event generation: event can be generated by a transition's action to trigger another transition.

2.2.2 Challenges of SBM Slicing

One major challenge of SBM slicing is ensuring that the resulting slice is a well-formed state machine. This is very different from a program slicing. In program slicing, after removing some lines of code from the original program, the resultant program slice is still an executable, standalone program. We say that the program slice is a well-formed program. However, such a well-formedness property is not easy to maintain in SBM slicing, as the omission of transitions may cause the slice-relevant states to become unreachable. Due to this problem, many model slicing approaches are very conservative about removing transitions or states that can break the graph connectivity. Ojala's approach [18] completely avoids removing transitions in an SBM and only replace the triggers, guards and actions with dummy values. Korel et al.'s approach [6] is slightly more aggressive because it merges states; but it deletes only self-looping or unreachable transitions in the end. Only Kamischke et al. [19] mentions a post-processing step after slicing to ensure that all states remaining in the slice are reachable from the initial state. Slices constructed by FormlSlicer are guaranteed to be well-formed through the state connecting step, which ensures that for any two states in a slice, all paths between the corresponding states in the original model have a corresponding path (possibly an abstract "true" transition) in the slice.

Some challenges faced by SBM slicing come from the difficulty of computing dependences among model elements due to many complex modeling constructs: (1) hierarchy (including cross-hierarchy transitions), (2) concurrency, and (3) no final state. Korel et al. [6] present a slicing algorithm that is similar to traditional program slicing; their definitions of data dependence and control dependence are similar to those in a program control graph [15]; this slicing approach only works on a flat state machine that has a final state. Wang et al. [20] present a wide range of dependences for slicing on a hierarchical and concurrent SBM; but because the units of their slicing algorithm are sub-machines (i.e, the whole sub-machine needs to be added to the slice if any transition or state is in the slice), the degree of reduction of model size might be minor. Kamischke et al. [19] use some of Wang et al.'s dependences on a hierarchical and concurrent SBM, but they improve the slicing algorithm so that all irrelevant model elements are sliced away. Neither Wang et al.'s and Kamischke et al.'s slicing approaches consider cross-hierarchy transitions and both use the traditional definition of CD that assumes a final state. Ranganath et al. [21] propose a new definition of CD for reactive software programs—called non-termination sensitive control dependence (NTSCD)—that Ojala [18] and Androutsopoulos et al. [22] adopt to non-terminating SBMs. However, neither slicer considers hierarchy constructs in the model. Unlike the existing techniques that overcome the challenges only partially, our slicing approach works on hierarchical, concurrent and non-terminating state-machine

models that have cross-hierarchy transitions.

There are many other challenges of SBM slicing compared to program slicing: (1) most state-based modeling languages allow non-determinism, which does not exist in programs; (2) if the SBM has hierarchy or concurrency constructs, the SBM can be in multiple states at a time during an execution; (3) the control flow in an SBM is arbitrary compared to that in a program. As what Androutsopoulos et al. [3] describe, by considering all these difficulties together, the SBM slicing task resembles the task of "slicing a non-deterministic set of concurrently executed procedures with arbitrary control flow". SBM slicing is still in the early stages and there are still many challenges yet to be tackled.

2.2.3 Relevant SBM Slicing Techniques

In general, there are two slicing approaches:

1. incrementally removing irrelevant model elements form the original model,

2. constructing a new model by incrementally adding relevant elements.

Almost all existing slicing techniques are a combination of both. One of Korel et al.'s slicing algorithms [6] starts with an initial slice that consists of states from the original model and incrementally adds transitions based on pre-computed dependencies. But the algorithm then repeatedly merges states and deletes self-looping transitions. Kamischke et al.'s slicing algorithm [19] starts with the model elements of the slicing criterion and adds more model elements incrementally based on pre-computed dependencies. FormlSlicer adopts the incremental addition approach as it is more aggressive in reducing the model size.

A technique to ease the slicing process is to transform a complex state machine to a simple graph structure. Ojala's a slicer of UML State Machine [18] constructs a CFG to capture all the possible executions of the UML state machines. In this CFG, there are different types of CFG nodes: BRANCH node, each represents the a state and the triggers and guards of the state's outgoing transitions: and SIMPLE/SEND node, each represents an action effect of a transition.

2.2.4 Dependences in SBM Slicing

As described in Subsection 2.1.1, program slices are computed according to data flow and control flow dependences. Similarly, most model slicing approaches use different depen-

dences to compute slices.

DD in an EFSM (one type of SBM languages) has been defined by Korel et al. [6] as a notion that one transition defines a value to a variable and another transition may potentially use this value. More formally, transition T_k is dependent on T_i with respect to variable v if (1) v is defined by T_i, (2) v is used by T_k, and (3) there exists a path (transition sequence) in the EFSM model from T_i to T_k along which v is not re-defined; such a path is referred to as the *definition-clear path* [6]. We will use this definition in this thesis.

CD in an SBM is a lot more complicated. Subsection 2.2.4.1 elaborates it separately.

Besides DD and CD, researchers have used different types of dependences in their model slicers [20, 18, 19]. Most of these dependences arise due to more advanced constructs added onto a basic SBM. We present some of them here:

- Parallel dependence: two states have parallel dependence if they are concurrent elements;
- Synchronization dependence: two transitions have synchronization dependence when one generates an event that the other consumes;
- Refinement control dependence: this dependence exists between a state and the initial states of all its descendants;
- Decisive order dependence: two nodes m and p are decisively order dependent on n when all maximal control flow paths from n contain m and p, one of them passing m before p and another of them passing p before m;
- Interference dependence: an dependence induced by concurrent reads and writes of shared variables.

Unlike DD and CD which serve for more general purposes in SBM slicing, the above-mentioned dependences are only restricted to a few SBMs with special semantics. In spite of this, they are useful in showing that one of the best ways to cope with an SBM with more complex semantics is to use more types of dependences in the dependence-based slicing technique.

2.2.4.1 Control Dependence

In program slicing, control dependence is traditionally defined in terms of a postdominance relation in a CFG [15, 23]. Intuitively, a node n_i is postdominated by a node n_j in a CFG if all paths to the exit node of the CFG starting at n_i must go through n_j. Having calculated the postdominance relation in a CFG, the control dependence relation can be obtained according to this: a statement n_j is said to be control dependent on a statement n_i if there exists a nontrivial path p from n_i to n_j such that every statement $n_k \neq n_i$ in p is postdominated by n_j and n_i is not postdominated by n_j.

Figure 2.2 shows a simple CFG from Ranganath et al.'s paper [21]. Based on the traditional definition of CD, we know that f is dependent on a and g is dependent on f, but g is not dependent on a because g does not postdominate f.

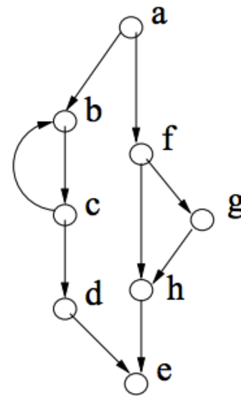


Figure 2.2: A simple CFG with a single end-node e

Although this definition of control dependence is widely used in program slicing, there are some other slightly different definitions as well. Podgurski and Clarke [24] have distinguished two different notions on control dependence—strong control dependence and weak control dependence:

1. n_j is strongly control dependent on n_i if there is a path from n_i to n_j that does not contain the immediate postdominator of n_i[1];

2. n_j is weakly control dependent on n_i if n_j strongly postdominates n_k, a successor of n_i, but does not strongly postdominate n_l, another successor of n_i.

The notion of strong control dependence is similar to the traditional notion, except that it captures both direct and indirect control dependences. In Figure 2.2, g is strongly control dependent on a because the path afg does not contain e, which is the immediate postdominator of a.

Weak control dependence shows a dependence relationship between the predicate condition of a loop (i.e. the branching node in a CFG) and a statement outside the loop that may be executed after the loop is exited; this dependence is not shown in strong control

[1]The immediate postdominator of n_i is the postdominator of n_i that does not postdominate any other postdominators of n_i.

dependence [24]. In Figure 2.2, d is weakly control dependent on c but not strongly control dependent on c.

As we can see, both the traditional control dependence and strong control dependence ignores the possibility of an infinite loop. They determine that a node outside the loop (e.g. node d in Figure 2.2) must always be executed after the predicate condition (e.g. node c), and therefore the latter is postdominated by the former. In other words, they are *non-termination insensitive* [21]. In spite of this, they are still widely used in most program slicing tasks which focus on debugging and program visualization and understanding; these slicing tasks do not consider preserving non-termination properties as an important requirement.

Because the postdominance relation assumes a single end-node in a CFG, all of these definitions of control dependence assume that the CFG has a single end-node.

The traditional definition on control dependence has been applied in model slicing. Korel et al. [6] presents the definition for control dependence between transitions in terms of the concept of postdominance:

Definition (Control dependence [6]). *Transition T_i has control dependence on transition T_k (transition T_k is control dependent on transition T_i) if (1) T_k's source state does not postdominate T_i's source state, and (2) T_k's source state postdominates transition T_i.*

Similar to the traditional control dependence in program slicing, Korel et al.'s definition is limited to SBMs with a single end state.

Later, some researchers in program slicing community have recognized that these traditional notions on control dependence are no longer applicable to most modern programs. Ranganath et al. [21] have identified two trends in modern program structures:

1. Many methods in modern programs raise exceptions or include multiple returns. This means that their corresponding program CFGs have multiple end nodes.

2. There is an increasing number of reactive programs with control loops that are designed to run indefinitely. This means that their corresponding CFGs have no end-node.

Therefore, most modern programs' corresponding CFGs do not satisfy the single end-node property. Ranganath et al. [21] proposed a new definition on control dependence:

Definition (NTSCD [21]). *In a CFG, n_j is (directly) non-termination sensitive control dependent on node n_i if n_i has at least two successors, n_k and n_l, such that*

1. *all maximal paths from n_k eventually reach n_j and do so before (possibly) reaching n_i;*

2. *there exists a maximal path from n_l on which either (1) n_j is not reached, or (2) n_j is reached but only after reaching n_i.*

The key observation on NTSCD is that reaching again a start node in a loop is analogous to reaching an end-node. This intuition is similar to weak control dependence [24] which considers that an infinite loop is important in determining control dependence. This implies that NTSCD is *non-termination sensitive* [21], like weak control dependence. In addition, NTSCD gets rid of the concept of postdominance and uses the concept of a maximal path. A maximal path is any path that terminates in a final transition, or is infinite [25]. The consideration of maximal paths imply that NTSCD is applicable to CFGs that do not satisfy the single end-node property.

There are more definitions that extend the definition of NTSCD. In summary, these definitions replace the "maximal paths" in NTSCD to other types of paths [25].

- By replacing "maximal paths" with "sink-bounded paths", we obtain the definition on non-termination insensitive control dependence (NTICD). This definition does not calculate control dependences within control sinks[2] [21].

- By replacing "maximal paths" with "unfair sink-bounded paths", we obtain the definition on unfair non-termination insensitive control dependence (UNTICD). This definition is in essence a version of NTICD modified to EFSMs rather than CFGs [22].

As a summary, we can see that the definition of control dependence has been improved by researchers in several generations, because of a change in modern program structures and a need to apply control dependence from program slicing to model slicing. Nevertheless, they all aim to capture the dependence that one node determines the execution of another node. Among a diverse range of definitions, the NTSCD has been chosen as a guideline in implementing the control dependence computation in this thesis. We will adapt Ranganath et al.'s algorithm of computing NTSCD [21] to fit our own needs.

[2]A control sink is a set of nodes that form a strongly connected component such that for each node n in the control sink each successor of n is also in the control sink [21].

2.3 Correctness of Slices

As stated in Section 1.1.2, a sliced structure (e.g., a program slice, a model slice) is correct when it simulates the original structure (e.g., the original program, the original model) with respect to the slicing criterion. Simulation has been defined as a relation between the original structure and a sliced structure in various literature.

2.3.1 Simulation in Programs

Simulation is a relation (i.e., the original) that relates a structure to an abstraction of the structure (i.e., the sliced). Simulation guarantees that every behavior of a structure is also a behavior of its abstraction; but the abstraction may have behaviors that are not possible in the original structure [26, 4].

Milner [26] proposed a technique to formally prove simulation between two programs in order to make precise the notion in which two programs are realizations of the same algorithm. The proof requires a precise definition of the program which enables simulation properties to be stated and proved succinctly:

Definition. *A program a is a quadruple $(D_{in}, D_{comp}, D_{out}, F)$ where $D_{in}, D_{comp}, D_{out}$ are disjoint domains and $F : D \rightarrow D$ is a total function ($D = D_{in} \cup D_{comp} \cup D_{out}$) with restrictions:*

- $F(D_{in} \cup D_{comp}) \subseteq D_{comp} \cup D_{out}$;
- *The restriction of F to D_{out} is the identity $I_{D_{out}}$.*

$D_{in}, D_{comp}, D_{out}$ are called the input, computation and output domains of a, respectively.

Milner [26] explained that this definition ensures that starting with a member of D_{in} and applying F repeatedly we get either an infinite sequence in D_{comp}, or a finite sequence in D_{comp} followed by a single member of D_{out}. Formally, he defined a *computation sequence* of program a to be a sequence $\{d_i \mid i \geq 0\}$ where $d_0 \in D_{in}, d_{i+1} = F(d_i), i \geq 0$, and either $d_i \in D_{comp}, i > 0$ or for some k, $d_i \in D_{comp}, 0 < i < k$ and $d_i = d_k \in D_{out}, i \geq k$. In addition, he defines the associated partial function of program a as $\hat{a} : D_{in} \rightarrow D_{out}$.

Based on these explicit definitions, Milner defines simulation as a relation:

Definition. *Let $R \subseteq D \times D'$. Then R is a weak simulation of a by a' if:*

1. $R \subseteq D_{in} \times D'_{in} \cup D_{comp} \times D'_{comp} \cup D_{out} \times D'_{out}$*;*

2. $R\,F' \subseteq F\,R$.

The condition 2 can be restated as $\forall d, d', (d, d') \in R \Rightarrow (F(d), F'(d')) \in R$.

Milner shows a proof example on pairs of simple flowchart programs. Basically, to prove condition 1 he needs to list all the possible input, computation and output domains of all programs. To prove condition 2, he needs to show for all d, d' that $(d, d') \in R \Rightarrow (F(d), F'(d')) \in R$ and this can be done using an inductive proof, in which the inductive step needs to be analyzed by cases. He did not give the details of the case-based analysis for condition 2.

2.3.2 Simulation in SBMs

In general, there are two conditions of the correctness property:

1. The sliced model simulates the original model;
2. The original model simulates the sliced model.

Korel et al. [6] formalizes the first condition by considering an input x to the execution of both the original and the sliced models:

> "Let M be an EFSM model. Let v be a variable at transition T_I in M. An EFSM sub-model M' is a non-deterministic slice of M with respect to variable v at transition T_I if for every input x the value of v at T_I during execution of M is equal to the value of v at T_I during at least one possible execution of M' on x."

Amtoft et al.'s research note [5] phrased the first condition slightly differently:

> "If the original program can do some observable action then also the sliced program can do that action; here an observable action may be defined either as one that is part of the slicing criterion, or as one that is part of the slice."

Milner's definition on weak simulation also has a similar idea to the first condition.

A sliced model that satisfies only the first condition is a slice that satisfies the basic correctness property. In this thesis, we enforce only the basic correctness property. Chapter 5 will prove this basic correctness property on model slices constructed by our model slicer.

Some researchers believe that the basic correctness property is not sufficient and they have defined the second condition of correctness. As an example, Korel et al. [6] present an "over-minimized" sliced model that satisfies the first condition only: the SBM becomes a single state such that all relevant transitions become self-looping transitions on this state. This sliced model is not useful for comprehension purpose. In light of this, Korel et al. impose the second condition of correctness and refer to it as the *traversability property* of a slice. They then proposed two state merging rules that produce slices that satisfy the *traversability property*.

A sliced model that satisfies both conditions of correctness is a slice that satisfies the stronger correctness property. Although we do not enforce the stronger correctness property in this thesis, we still attempt to make the slices constructed by our slicer as precise as possible. For example, we use Korel et al.'s state merging rules so as to avoid producing an "over-minimized" slice.

2.4 Cone of Influence Reduction

We notice that the idea of model slicing is a bit similar to the technique of *cone of influence reduction* in model checking. Cone of influence reduction is one of the abstraction techniques that applies on a high level description of the system, before a model for the system is constructed. It decreases the size of the constructed model by eliminating those variables that do not influence the variables that are referred to in the specification; in this way, the properties to be checked are preserved but the size of the model that needs to be verified is smaller [4].

Biere et al. [27] discussed the classical definition of cone of influence reduction. They defined the model as follows:

Definition. *Let* $X = \{x_1, \ldots, x_n, x_{n+1}, \ldots, x_m\}$ *be a set of* m *boolean variables and let* $F = \{f_1, \ldots, f_n\}$ *be a set of* $n \leq m$ *boolean transition functions where* f_j *is a function for* x_j *for all* $1 \leq j \leq n$*. Finally, let* $R = \{r_1, \ldots, r_n\}$ *be a set of initialization functions where* r_j *is a function for* x_j *for all* $1 \leq j \leq n$*. Then* $M = (X, F, R)$ *is called a model.*

To perform cone of influence reduction on such a model, firstly a dependency graph of the state variables needs to be constructed. The immediate dependency set, $dep(x_j)$, of a state variable x_j is defined as $dep(x_j) = \{x_l \mid x_l \; occurs \, in \; f_j\}$ where f_j is the transition function for x_j. The cone of influence of a state variable x_j, denoted as $coi(x_j)$, is the least set of variables that contains x_j and includes $dep(x_l)$ for all $x_l \in coi(x_j)$. In other words, $coi(x_j)$ is the solution of a least fixed-point equation; its computation is iterative. With respect to an LTL formula f, the reduced model is defined as $coi(M, f) = (coi(x), coi(t), coi(r))$ where $coi(x) = \cup\{coi(x_j) \mid x_j \in var(f)\}$ in which $var(f)$ is the set of variables that occur in f, $coi(t)$ and $coi(r)$ include the corresponding transition and initialization functions over $coi(x)$ [27].

We can see that the cone of influence reduction shares some similarities with dependence-based model slicing. The computation of $coi(x_j)$ resembles the computation of a transitive closure of dependencies in model slicing. The LTL formula f has a similar role as the slicing criterion in model slicing. However, the difference between cone of influence reduction and model slicing is also evident: the model used in cone of influence reduction is a highly abstract system that is quite different from the SBMs on which our model slicing techniques apply.

Chapter 3

Preliminaries

This chapter presents some background and terminology that is related to understanding the design of FormlSlicer. Section 3.2 introduces the semantics of FORML [28]. Section 3.3 discusses the scope of FormlSlicer and explains how FormlSlicer treats feature modules in FORML.

3.1 Terminology

Feature-oriented Software Development (FOSD) This paradigm advocates the use of features as the primary criterion to identify separate concerns. Usually, it is applied to the requirements and development of feature-rich software systems.

System and Environment In this thesis, we use "system" to refer to the collection of features in a software system and "environment" to refer to any external factors that can influence the system. This divides the variables in the requirements model of a software system into two groups. The first group is the set of environment-controlled variables (i.e., their values cannot be directly modified by the system). As an example, consider the actual speed of a car, which value can only be a result of a combination of external and internal factors; no equipped features of the car can directly modify the actual speed; instead, the system can only monitor its value. The second group is the set of system-controlled variables (i.e., their values can be modified by one or more features in the system). For example, the acceleration of a car is a system-controlled variable.

3.2 What Is FORML

FORML is a modeling language that can be used to specify feature-oriented models of requirements of software systems. It is based on the paradigm of feature-oriented software development (FOSD) and accompanied by modeling-language constructs for explicitly expressing intended feature interactions [11]. A model expressed in this language consists of two views [28]:

- A world model is a description of the environment in which the software system will operate. The concepts in the world model are expressed in the UML class-diagram notation.
- A behavior model is a state-machine model that describes the requirements for a software system. The syntax for the behavior model is based on UML state machines. It is structured in terms of many feature modules, each specifying the behavior of one feature. If a feature is independent of existing features, then the module is expressed as a fully executable state machine. If a feature enhances (i.e., extends or modifies) existing features, the enhancements are expressed as a set of state-machine fragments that extend existing feature modules.

A state machine in a FORML behavior model mainly consists of [28]:

- finite sets of states and orthogonal regions, organized into a state hierarchy defined by a containment relation over states and regions;
- a set of transitions between states, each carrying a label that specifies the transition's name, trigger, guard and actions;
- a set of macro definitions which are abbreviations of FORML expressions that are used to simplify transition labels.

A state of a FORML state machine can be either a *composite state*, which contains other states, or a *basic state*, which contains no other states. A state that is contained in a composite state is called the *child state* of the composite state; the composite state is called the *parent state* of the *child state*. A black solid circle, called a *pseudo initial state*, points to the *default initial state* of a sub-machine. The first state immediately after the initial pseudo-state in the region is called the *default initial child state* of its parent state. A composite state contains one or more *orthogonal regions* (*regions* for short), where

each region holds a sub-machine that executes concurrently with sub-machines in sibling regions. The structure of a state machine is shown in Figure 3.1. It contains two states, *S1* and *S2*. *S1* is a basic state. *S2* is a composite state which consists of two orthogonal regions, *C1* and *C2*. *S2* is *S3*'s parent state and *S3* is *S2*'s default initial child state. *S1*, *S3* and *S5* are default initial states in their respective machines.

Throughout the thesis, we use the following definitions adapted from Shaker's PhD thesis [28] to access information about state hierarchy in a state machine:

> "The ancestors of a state x are all of the nodes along the path in the tree of state hierarchy from the root node to x. The descendants of a state x are all of the states in the subtrees of x. The rank of a state x in the state hierarchy is the length of the path from the root to x. The least common ancestor of a state $x1$ and a state $x2$ is the maximum-rank state that has both $x1$ and $x2$ as descendants."

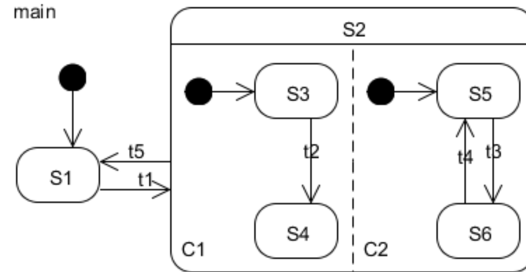


Figure 3.1: The structure of a state machine in FORML's behavior model

A "flat" state machine (i.e., a non-hierarchical and non-concurrent state machine) can be in only one state at a time; the state it is in at any given time is called the *current state*. Because a FORML state machine has hierarchy and concurrency constructs, it can be in multiple states at a time. The set of current states in the FORML state machine is the **state configuration**.

A transition label has the following format:

$id : te[gc]/al_1...al_n$

where *id* is the name of the transition, *te* is an optional trigger expression, *gc* is an optional guard condition, and $al_1...al_n$ are labels that specify a set of concurrent actions.

The trigger expression of a transition is a world change event (WCE): an event generated by the external environment (e.g., user pressing a button to generate a message) or by the system (e.g., a transition's action generating a message) to change the world model. There

are three types of WCEs: (1) a new message object generated in the world model, (2) an existing message object removed from the world model, and (3) a change in value of a message object's attribute.

The guard of a transition is a boolean condition which is evaluated together with the transition's trigger expression. Usually, the guard checks the properties of the object associated with the WCE, using the object variable "o"; for example, if a transition is expressed as "*t1: SetHeadway+(o) [o.to=myproduct] /a1: HC.headway := o.dist*", the guard is checking whether the message object *SetHeadway* is sent to *myproduct*. The guard can also check whether a global variable or a function's output is equal to a certain value.

The action of a transition is a world change action (WCA) that specifies a change to the world model. There are three types of WCAs: (1) generating a new message object; (2) destroying a set of existing objects; and (3) setting an object's attribute or a global variable to a new value. Again, the object variable "o" is used in an WCA to refer to the current object associated with the transition's WCE; for example, if a transition is expressed as "*t1: SetHeadway+(o) [o.to=myproduct] /a1: HC.headway := o.dist*", the value of *dist* of the message object *SetHeadway* is assigned to the global variable *HC.headway*.

There are many reasons why we choose FORML. The most important ones are:

- FORML is a UML-like language. Its feature modules are based on UML state machines, which is one type of state-based model (SBM). This makes the design of our slicing tool much easier, because we can utilize some existing literature on SBM slicing.
- It is good practice for the software engineering research community to adopt and extend state-of-the-art analysis tools [29]. FORML is designed as a precise modeling language and there have been a few tools developed to manipulate the language, including the FORML Feature Composer and a collection of transformation tools from FORML to SMV [29]—a modeling language used by the NuSMV model checker [30]. It is beneficial to the community in further extending this tool set by creating a model slicer on FORML.

3.3 Scope of FormlSlicer

FormlSlicer faces many challenges, because of the difficulty of slicing on SBMs (Section 2.2.2) and the fact that FORML state machine is a highly complex type of SBM

(Section 3.2). We have solved some of the challenges in this thesis project; but there are still a few limitations. In this section, we describe what kind of input model that FormlSlicer can handle.

FormlSlicer focuses on slicing the feature modules in the behavior model of a FORML model. It does not perform slicing on the world model. However, it is very easy to perform slicing on the world model by using the set of relevant variables in FormlSlicer's slicing output. This is left to future work.

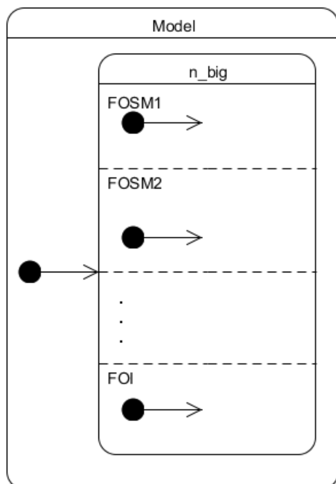


Figure 3.2: An example model

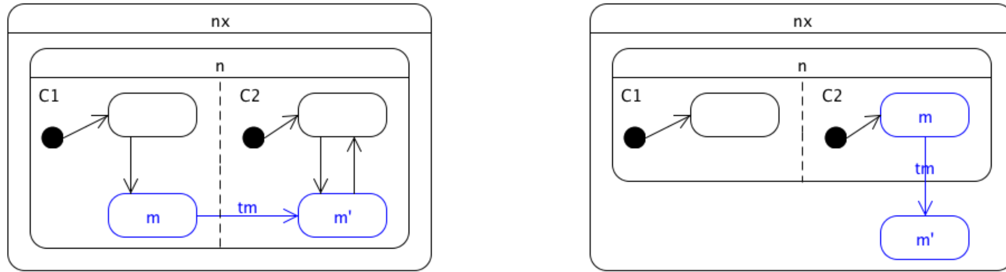
FormlSlicer treats each feature module in FORML as a state machine. If a feature is an independent feature, the feature module is treated as a complete state machine. If a feature is an enhancement or modification of another feature (e.g., the cruise control (CC) feature which extends and overrides the basic driving service (BDS) feature), the feature module is represented as a fragment in FORML. Because it is very difficult to perform slicing on a state-machine fragment, we compose the fragment with its base feature modules to create one complete state machine. We call such a complete state machine a feature-oriented state machine (FOSM) to indicate that it is a composed feature in the form of a state machine.

FormlSlicer treats the whole model as one big state machine. This big state machine comprises a composite state containing many orthogonal regions. Each orthogonal region contains one sub-machine, which is an FOSM. Figure 3.2 shows an example of such a big state machine. We can see that the model is a composition of all features.

The input to FormlSlicer is a model of all features currently making up a software system. If new features are added to the software system, then slicing needs to be performed from scratch. We discuss slicing of an evolving software system in future work.

FormlSlicer imposes a few restrictions on the input model. The first restriction is that a transition cannot cross from one orthogonal region into another orthogonal region belonging to the same parent. An example of such a transition is shown in Figure 3.3a. The ultimate reason of having concurrent regions in a model is to simulate multiple threads in a program. Although different threads can interact with one another by generating and

reacting to shared events or by accessing shared memory, it is illogical that a running thread suddenly reaches into the execution of another thread.

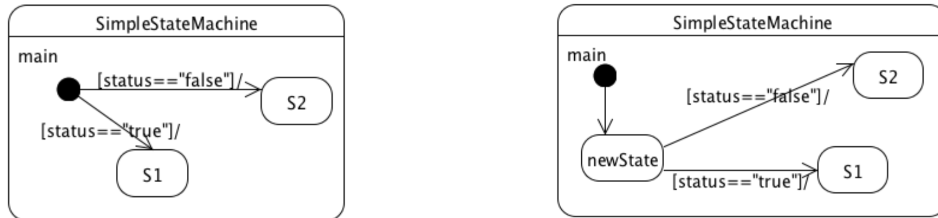


(a) A transition crossing from a region to its sibling region

(b) A transition emanating from a descendant state of a composite state containing multiple regions and exiting the composite state

Figure 3.3: Two types of transitions (shown in blue) not allowed in FormlSlicer

The second restriction is that no transition enters or exits the boundary of the parent state of an orthogonal region that has a sibling orthogonal region. This kind of transition is illustrated in Figure 3.3b. Because FormlSlicer has many other more important challenges to solve, we do not further complicate FormlSlicer by considering this special case which occurs very rarely in real-life models. This will be left for future work.



(a) Labeling on arrow from a pseudo state to a default initial state is not allowed; more than one arrow from a pseudo state is also not allowed.

(b) If user needs to use guard condition on the arrow from a pseudo state, user can redefine the original pseudo state as a new default initial state.

Figure 3.4: Restrictions and solution on the arrow from a pseudo state to default initial state

Another minor restriction is that the arrow from a pseudo state to the default initial state cannot be labeled. Furthermore, a state machine has exactly one default initial state; in other words, we do not allow a pseudo state to have multiple outgoing arrows pointing to different states. Figure 3.4a illustrates an example of multiple labeled arrows that exit from a pseudo state. If user wants to use guard conditions on the arrow from a pseudo

state, in order to specify different default initial states under different conditions, user can redefine the original pseudo state as a new default initial state and create a new pseudo state to point to it (Figure 3.4b).

Chapter 4

FormlSlicer

4.1 Overview of FormlSlicer's Workflow

Given a feature-rich state-machine model that consists of n features, FormlSlicer creates a sliced model with respect to each feature. Figure 4.1 shows the big picture of FormlSlicer's workflow. It consists of two major tasks: a preprocessing task and a slicing task[1].

The preprocessing task parses an original model, computes three types of dependences and stores the generated results in different tables. The preprocessing task is described in detail in Section 4.2 and Section 4.3.

In the slicing task, FormlSlicer forks off multiple slicing processes; each process operates independently but reads the same resources generated from the preprocessing task. Each process considers a different feature as the FOI and treats the rest of FOSMs as the ROS; then it goes through a multi-stage model-slicing procedure to slice the FOSMs in ROS with respect to the selected FOI; eventually, it outputs a sliced model. The use of concurrent threads in FormlSlicer can save the time of the slicing task. The slicing task is described in detail in Section 4.4.

[1] We use five different words—"workflow", "task", "process", "stage" and "step"—to indicate the granularity of a procedure. FormlSlicer's "workflow" consists of two "tasks". One of the "task" has many "processes". The slicing "process" goes through many "stages". Each "stage" is further divided into many "steps".

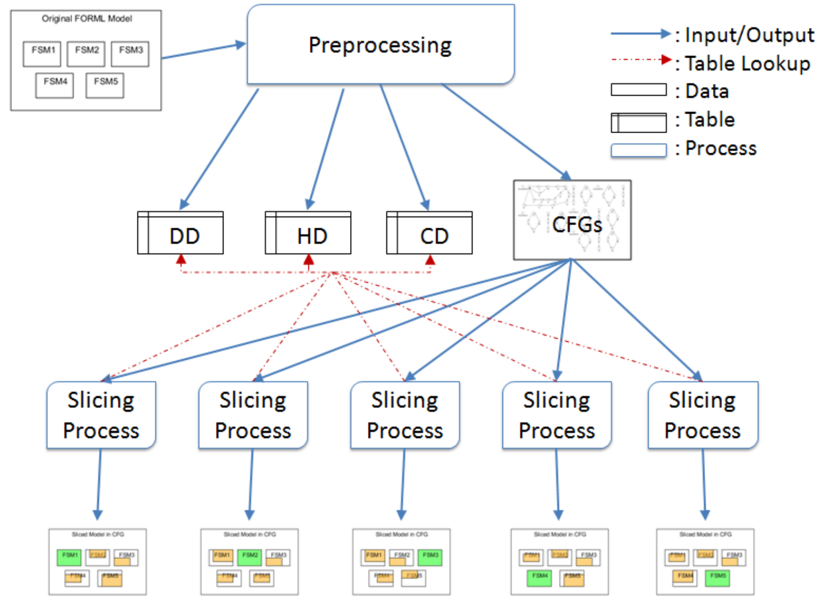


Figure 4.1: Overview of FormlSlicer's workflow: the preprocessing task and the slicing task

4.2 Preprocessing: Model Parsing and Conversion from FORML to CFGs

FormlSlicer includes a simple parser to read a feature-oriented state-machine model in which each feature is modeled as a distinct sub-machine running in parallel with other features' sub-machines. The input model is expressed in textual format. The input format is shown in Table 4.1.

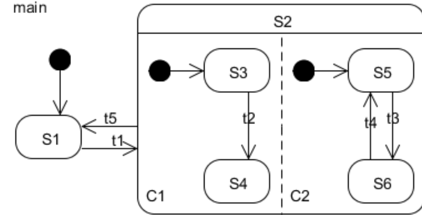
Model Element to be Declared	Declaration Format
Feature	feature ⟨name⟩
Macro	macro ⟨input sequence⟩⟨replacement output sequence⟩
State	state ⟨name⟩⟨parent region's full name⟩⟨default initial state?⟩⟨composite state?⟩
Transition	transition ⟨label⟩⟨source state's full name⟩⟨destination state's full name⟩
Region	region ⟨name⟩⟨parent state's full name⟩

Table 4.1: Format of an Input/Output File to Specify one FOSM

A simple input example that corresponds to the FOSM in Figure 4.2b is shown in Figure 4.2a.

```
feature F
region main F
state S1 F.main true false
state S2 F.main false true
transition t1:E1+(o)/a1:x:=1; F.main.S1 F.main.S2
transition t5:E1-(o)/a1:x:=0; F.main.S2 F.main.S1
region C1 F.main.S2
region C2 F.main.S2
state S3 F.main.S2.C1 true false
state S4 F.main.S2.C1 false false
transition t2:[inState(F.main.S2.C2.S6)]/ ...
state S5 F.main.S2.C2 true false
...
```

(a) Input text



(b) The corresponding FOSM

Figure 4.2: A simple input example

4.2.1 Transformation of Transition Labels

As described in Section 2.2.4, data dependence in an SBM captures the notion that one transition assigns a value to a variable and another transition may potentially use this value. Similarly, between the transitions in FORML state machines, there can be data dependence. However, a FORML transition label can contain a WCE, a guard and some WCAs and it is very inconvenient to compute data dependencies by directly using these complex labels. Instead, FormlSlicer transforms the transition labels into simple expressions, preserving only the information needed to detect data dependence.

In general, a WCA performed by one transition can directly affects the WCE or guard of another transition. For example, a message object that is generated by a transition t_1 may trigger another transition t_2; in this case, the type of message C appears in the WCE of t_2 (i.e., *C+(o)*) and in the WCA of t_1 (i.e., *+C(list(param = e))*). For our purposes, it is sufficient to be able to match actions and trigger events involving the same message type or the same variable. FormlSlicer simplifies a transition label into two groups of variables:

- *Monitored variables*, which represent phenomena that are sensed by or are inputs to the system;
- *Controlled variables*, which represent phenomena that are controlled or affected by system outputs [31].

In the previous example, FormlSlicer will identify "*+C*" as a controlled variable of transition t_1 and as a monitored variable of transition t_2, so that it is easy to detect the data dependency between t_1 and t_2 automatically.

Table 4.2 shows how FORML will transform different types of WCE expressions. The first two columns show the types and formats of WCE: creation of a message object, deletion of a message object, and changing the value of an attribute of a message object. The third column shows the corresponding string value (representing a monitored variable) that FormlSlicer transforms each WCE type into.

WCE Type	WCE Format	Monitored Variable
Message Object C Appears	*C+(o)*	*+C*
Message Object C Disappears	*C-(o)*	*-C*
Attribute in Message Object C Changes Value	*C.a~(o)*	*C.a*

Table 4.2: How FormlSlicer extracts Monitored Variables from WCEs

Table 4.3 shows how FormlSlicer transforms different types of WCAs. The first two columns show the three types and formats of WCA in FORML, including generation of a message object, deletion of a message object[2], and assignment. The fourth column specifies the corresponding string value (representing a controlled variable) that FormlSlicer transforms each WCA type into. Note that in an assignment action, the arguments of the function are monitored variables and this is depicted in the third column.

WCA Type	WCA Format	Monitored Variable	Controlled Variable
Generate Message Object C	*+C(list(param = e))*		*+C*
Destroy Message Object C	*C-(O)*		*-C*
Assignment Action	*v*:=function(*v1,v2,...*)	*v1,v2,...*	*v*

Table 4.3: How FormlSlicer extracts Monitored/Controlled Variables from WCAs

The guard of a transition may refer to variables whose values are defined in an assignment action of another transition. For example, if a transition t_1's WCA is "*a1: status='failed';*" and another transition t_2's guard is "*[status=='failed']*", there exists a data dependency between t_1 and t_2 with respect to the variable *status*. To detect data dependence, FormlSlicer just needs to know which variables are involved in the guard expression (e.g., the variable *status*); it is not concerned with the guard expression itself. Based on this idea, FormlSlicer simplifies a guard expression to its constituent monitored variables.

[2]In FORML, when an action destroys a message object, it needs to specify exactly what set of objects are destroyed; but in FormlSlicer we over-approximate the influence of this action by assuming that all the objects belonging to that category of message are destroyed.

A guard expression can be one of the three different types, as shown in the first two columns in Table 4.4. The third column shows how FormlSlicer transforms each of the different types of guard expression into a set of monitored variables. In particular, the guard condition of *inState(DummyState)* is an interesting construct. It means that this guard condition is true only when the model's current state configuration contains the state *DummyState*.

Guard Type	**Guard Format**	**Monitored Variable**
Comparison	*var1 >var2*	*var1, var2*
InState	*inState(DummyState)*	*DummyState*
Function	*function1(var1) == 5*	*var1*

Table 4.4: How FormlSlicer extracts Monitored Variables from Transition Guard Conditions

4.2.2 Control Flow Graph (CFG)

FormlSlicer converts the input model into a set of CFGs to simplify the representation of the model to ease dependence analysis. A CFG consists of only nodes and edges. As shown in Figure 4.3, there are two types of nodes—TNode and SNode—which are subtypes of the generic type—Node. Each Node has an ID, a set of outgoing nodes, and a set of incoming nodes.

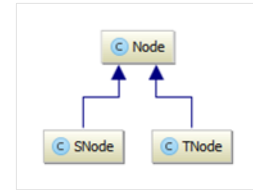


Figure 4.3: The class hierarchy of Node, TNode and SNode in CFG

FormlSlicer creates a TNode for each transition. Each TNode contains the set of monitored variables and controlled variables that appear in the label of the corresponding transition. It also contains a singleton set of incoming nodes containing the ID of the transition's source state and a singleton set of outgoing nodes containing the ID of the transition's destination state.

FormlSlicer creates an SNode for each state in the input model. An SNode is simpler than a TNode. It contains only the generic information (i.e., its own ID and sets of outgoing and incoming nodes' IDs) that are inherited from the generic type Node.

The result is one CFG for each distinct sub-machine in an FOSM. But if there is a transition that crosses a hierarchy border, it will be converted into a TNode that connects two distinct CFGs together. The CFGs created from the FOSM example in Figure 4.2b are shown in Figure 4.4.

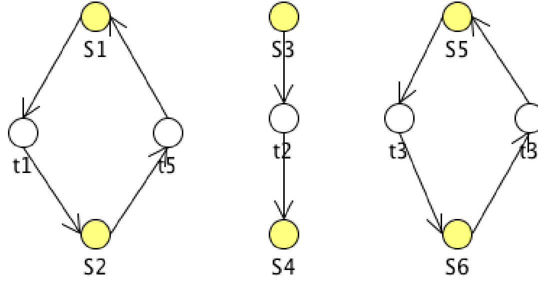


Figure 4.4: Several CFGs for the FORML model in Figure 4.2b. Yellow nodes are SNodes.

4.3 Preprocessing: Dependence Analyses

CFGs are lightweight graph structures suitable for dependence analyses, because the analyses concern only the connectivity relationship among nodes. This section introduces three types of dependence that are computed by FormlSlicer.

4.3.1 Hierarchy Dependence

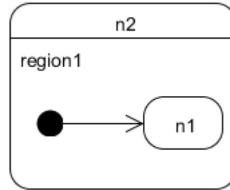


Figure 4.5: Hierarchy dependence

As explained in Section 3.2, the states in a FORML state machine are organized in a hierarchy. Hierarchy dependence (HD) reflects the state hierarchy relationship among nodes. We say that a child state n_1 is **hierarchy dependent** on its parent state n_2, denoted as $n1 \xrightarrow{hd} n2$. Figure 4.5 illustrates such a dependence.

FormlSlicer computes the hierarchy dependencies for all states in the model while creating the CFGs. As it scans through all states once, the computation complexity is $O(|S|)$ where $|S|$ is the number of states. The computation results are kept in the tables:

HDtable1 a 1D-table mapping each SNode to its parent SNode;

HDtable2 a 1D-table mapping each SNode to a set of its default initial child states[3].

Note that we have two tables that have opposite mapping directions in state hierarchy. The HDtable1 keeps records for all states except the root state. The HDtable2 keeps records for the composite states only; it is similar to Wang et al.'s refinement control dependence [20].

4.3.2 Data Dependence

As introduced in Chapter 2, data dependence (DD) usually depicts a "define-use" relationship among different instructions in a program. The relationship is also applicable in models.

We use $mv(t)$ and $cv(t)$ to represent monitored and controlled variables of transition t, respectively. We say that transition t_k is **data dependent** on transition t_1 with respect to a variable v, denoted as $t_k \xrightarrow{dd}_v t_1$, iff t_k monitors (i.e., reads from) v and t_1 controls (i.e., writes to) v ($v \in mv(t_k) \cap cv(t_1)$) and there exists a path from t_1 to t_k along which no other transition controls v (i.e., $\exists [t_1 \cdots t_k](k \geq 1).v \notin cv(t_j)$ for all $1 < j < k$). The path of $[t_1 \cdots t_k](k \geq 1)$ such that $v \notin cv(t_j)$ for all $1 < j < k$ is called a **definition-clear path**. Figure 4.6 illustrates this concept.

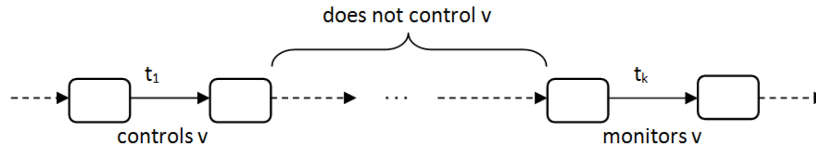


Figure 4.6: Data dependence between t_k and t_1 w.r.t. v

DD is the most important dependence among the three. In general, FormlSlicer is looking for relevant parts in the other FOSMs on which the FOI is (directly or indirectly) data dependent.

[3] A composite state can contain multiple regions and therefore can have multiple default initial child states.

The "inState()" Expression In a FORML transition label, there is a special type of guard condition called "inState()" that holds when the model is in a specific state. In the example shown in Figure 4.7, transition t2 has a guard condition "*inState(A)*", which means that t2 will be triggered when the current state configuration of the model includes the state of A. For consistency, FormlSlicer treats "inState()" expression as a sub-type of data dependence[4]. In this example, FormlSlicer considers t2 to be data dependent on all of the incoming transitions to A (i.e., t1).

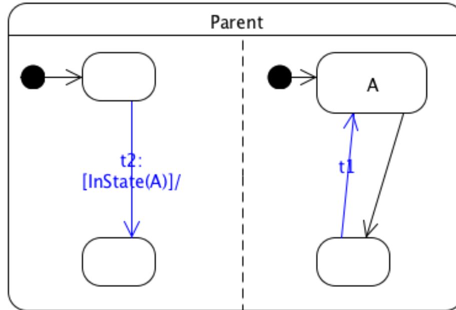


Figure 4.7: The "inState()" expression as a variation of data dependence

Algorithm 4.1 shows how FormlSlicer computes data dependence. Compared to existing algorithms, ours deals with a more complex model with respect to the original model's state-hierarchy structure. The algorithm takes an input of all the nodes and `HDtable2`. It traces from each TNode with a controlled variable `v` to other reachable TNodes that monitor `v`, thereby establishing a data dependency between pairs of TNodes. In each iteration of the `foreach` statement at Line 1, the algorithm checks whether `node1` (a TNode) has controlled variables. If so, it iterates for each such controlled variable at Line 4 to search for other TNodes that are data dependent with respect to this particular variable. In the beginning, the algorithm initializes a queue `qt` with the destination node of `node1` at Line 7; then, at each iteration of the `while` loop at Line 8, it dequeues one node from `qt` and checks whether that node monitors the variable. If so, a data dependency is established between `node1` and the dequeued node, as shown at Line 20. The statement at Line 23 checks whether `currNode` controls the variable `v`. If so, it means there is no longer a definition-clear path between `node1` and nodes reachable from `currNode`, and so there is no need to continue the search beyond `currNode`.

[4]In this thesis, we do not explore some quirky cases of "inState()" expression (e.g., A does not have incoming transitions at all).

Algorithm 4.1: Algorithm of Data Dependence Computation used in FormlSlicer

```
   Input: allNodes, HDtable2
   Output: DDtable
1  foreach node1 in allNodes do
2  |  if node1 instanceof SNode OR node1.controlledVar = null then continue;
3  |  set controlleds := node1.controlledVars // node1's controlled variables
4  |  foreach v in controlleds do
5  |  |  set visitedSNodes := {} // Empty visitedSNodes
6  |  |  set qt := empty unique queue
7  |  |  set qt.enqueue(node1.outgoingNode) // Initialize qt
8  |  |  while qt is not empty do
9  |  |  |  set currNode := qt.dequeue();
10 |  |  |  if currNode instanceof SNode then
11 |  |  |  |  if visitedSNodes contains currNode then continue;
12 |  |  |  |  if HDtable2[currNode] is not null then
13 |  |  |  |  |  qt.enqueue(HDtable2[currNode]) // add currNode's default initial child SNodes
14 |  |  |  |  end
15 |  |  |  |  qt.enqueue(currNode.outgoingNodes) // currNode (an SNode) may have many outgoing node
16 |  |  |  |  ADD currNode to visitedSNodes
17 |  |  |  else
18 |  |  |  |  if currNode == node1 then continue;
19 |  |  |  |  if currNode.monitoredVars contains v then
20 |  |  |  |  |  DDtable[v,currNode].add(node1);
21 |  |  |  |  end
22 |  |  |  |  set isContVRset := false
23 |  |  |  |  if currNode.controlledVars contains v then  set isContVRset := true
24 |  |  |  |  if isContVRset := false then
25 |  |  |  |  |  qt.enqueue(currNode.outgoingNode) // currNode (a TNode) has 1 outgoing node
26 |  |  |  |  end
27 |  |  |  end
28 |  |  end
29 |  end
30 end
```

The computation results are stored in a table:

DDtable a 2D-table mapping a pair (a variable v and a TNode t_j) to a set of TNodes, to indicate that t_j is data dependent on a TNode t_i in the set with respect to v.

Analysis of Algorithm The worst-case time complexity of Algorithm 4.1 is $(|S| + |T|) \cdot |T| \cdot |CV|$ where $|S|$ is the number of SNodes, $|T|$ is the number of TNodes and $|CV|$ is the largest number of controlled variables for any TNode. The algorithm terminates because the set `visitedSNodes` at Line 11 prevents repeated visits to the same SNode in the CFG traversal.

4.3.3 Control Dependence

4.3.3.1 Non-termination Sensitive Control Dependence

As introduced in Chapter 2, control dependence (CD) captures the notion of whether one node can decide the execution of another node. In a CFG, a branching node has more than one outgoing paths, i.e., its outdegree is greater than 1. A branching node is always an SNode, which is equivalent to a state in FORML with multiple outgoing transitions. Generally, a branching node is a decision point that determines whether an execution proceeds along one possible execution branch or another.

If control dependence is not preserved, the sliced model will omit useful information about how an execution reaches an important node, and therefore becomes more imprecise. When control dependence is preserved, then whenever an important node is included in the sliced model, all earlier nodes that act as decision points that lead the execution to the important node will be included in the sliced model too.

As elaborated in Section 2.2.4.1, researchers have proposed many different definitions of control dependence in both program slicing and model slicing. At present, there is no standard algorithm to compute control dependence for slicing an SBM. We use Ranganath et al.'s non-termination sensitive control dependence (NTSCD) [21] because we believe that this definition captures the notion of control dependence with a sufficient degree of precision.

We have explained NTSCD in Section 2.2.4.1. The consideration of maximal paths[5] in the definition implies that NTSCD is applicable to CFGs that do not satisfy the single

[5] A maximal path is any path that terminates in a final transition, or is infinite [25].

end-node property. The precise definition of NTSCD is presented as below (with slight wording changes):

Definition. *In a CFG, n_j is (directly)* ***non-termination sensitive control dependent*** *on node n_i if n_i has at least two successors, n_k and n_l, such that*

1. *all maximal paths from n_k eventually reach n_j and do so before (possibly) reaching n_i;*
2. *there exists a maximal path from n_l on which either (1) n_j is not reached, or (2) n_j is reached but only after reaching n_i.*

In other words, a node n_j is control dependent on a predecessor n_i if n_i has *some* outgoing paths that always lead to n_j and also has *some* outgoing paths that are not guaranteed to lead to n_j. This is illustrated in Figure 4.8. In the definition, an execution returning to the start node is analogous to reaching the end of the execution path.

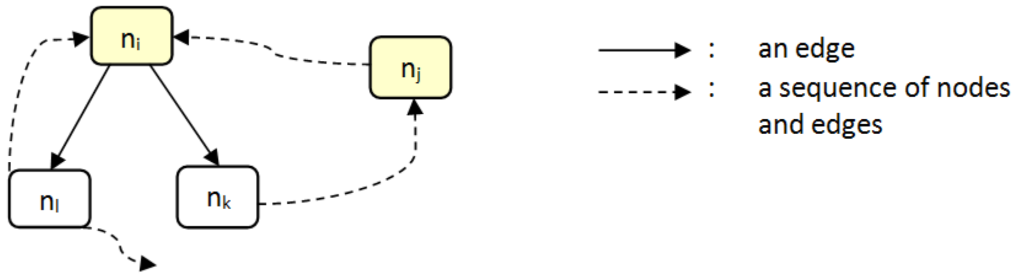


Figure 4.8: An illustration of non-termination sensitive control dependence: n_j is control dependent on n_i

4.3.3.2 CD Algorithm: Paths Representation for a Node

Ranganath et al. [21] presents a dynamic algorithm for computing NTSCD. Its main idea is to search from each branching node in a CFG for any other nodes that can be control dependent on that branching node. The algorithm represents sets of CFG paths symbolically and propagates these symbolic values in a CFG traversal to collect the effects of control-flow choices at each program point in the CFG. Based on the main idea, we adapted the algorithm to our CFGs and designed a dynamic algorithm to fit our own needs. There are a few reasons why we do not directly use Ranganath et al.'s algorithm: (1) The paper

shows only the skeleton of their algorithm and does not explain certain details, such as why the cardinality of sets of paths and a branching node's outdegree can be used to determine control dependence; (2) the CFG used in their algorithm is slightly different from ours[6]. However, the main idea remains the same.

FormlSlicer computes control dependence by first finding CFG paths from each branching node to all reachable nodes, and then for all possible paths from a branching node n_i to a reachable node n_j, performs a special operation called *reduction* and then determines whether n_j is control dependent on n_i.

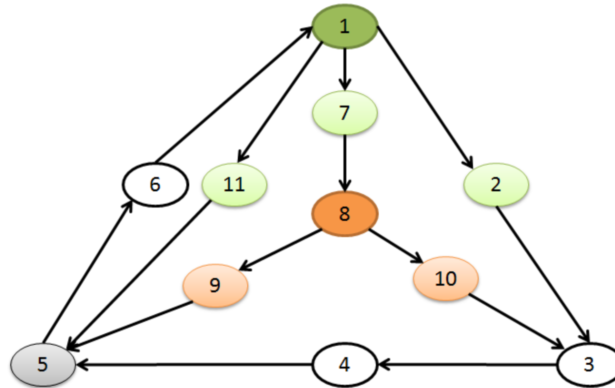


Figure 4.9: A CFG example to illustrate the paths from Node 1 to Node 5: Node 8 has two outgoing nodes (9 and 10), highlighted in orange color; Node 1 has three outgoing nodes (2, 7 and 11), highlighted in green color.

FormlSlicer needs an efficient representation of all possible paths from a branching node n_i to a reachable node n_j. At any point in time, the CD algorithm is traversing paths in the CFG from one of its branching nodes, which we denote `node1`. The algorithm keeps track, for each node reachable from `node1`, of all paths from `node1` to that node. The representation of each path contains only information of branch decisions made along the path.

Consider the CFG in Figure 4.9 and all of the paths from Node 1 to Node 5:

Path 1: "1→11→5";

[6]The CFG in their algorithm represents the structure of the statements in a software program and therefore the loops in their CFG are more predictable. The CFG in our algorithm has a more unpredictable graph structure; but it is not a completely random graph either, because a TNode in our CFG has exactly one source SNode and one destination SNode (which is a useful information in complexity analysis).

Path 2: "1→7→8→9→5";

Path 3: "1→7→8→10→3→4→5";

Path 4: "1→2→3→4→5";

We represent Path 1 as "1:11". Along Path 1, there is only one decision point at Node 1, and the path follows the branch of Node 11 from Node 1.

We represent Path 2 as "1:7,8:9", which comprises two sub-paths "1:7" and "8:9". This means that along Path 2, there are two branching nodes. The first branching node is Node 1, from which the path follows the branch of Node 7. The second branching node is Node 8, from which the path follows the branch of Node 9.

We represent Path 3 as "1:7,8:10". The only difference between Path 3 and Path 2 is that at the second branching node (Node 8), Path 3 follows the branch of Node 10. After these two branching decisions, Path 3 visits Nodes 3 and 4 on its way to Node 5. Because these nodes do not contribute to the determination of Path 3, they are omitted from the path representation.

We represent Path 4 as "1:2". At Node 1, Path 4 follows the branch of Node 2.

Altogether, the paths representation from Node 1 to Node 5 is shown in Figure 4.10. Paths are separated by semicolons. The length of a paths representation is the number of paths. Each path consists of one or more sub-paths, separated by commas. Each sub-path consists of two node indices: the first index (called source index) is the ID of a branching node and the second index (called branch index) is the ID of the branching node's successor node followed in the sub-path.

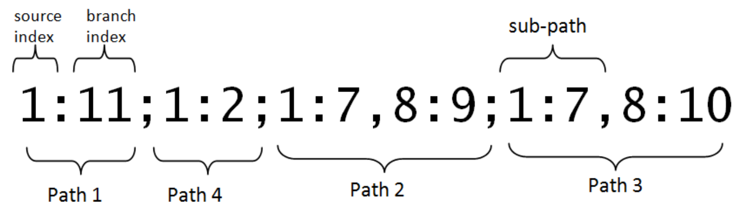


Figure 4.10: The paths representation from Node 1 to Node 5 in the CFG in Figure 4.9

Note that all paths from Node 1 lead to Node 5. Therefore, Node 5 is not control dependent on Node 1.

4.3.3.3 CD Algorithm: Reduction of the Paths Representation

Consider the paths representation in Figure 4.10. FormlSlicer can automatically detect that Node 5 is not control dependent on Node 1 by reducing the representation. Because Node 8 has only two outgoing neighbors (Node 9 and Node 10), the two paths "1:7,8:9" and "1:7,8:10" can be reduced to "1:7". Now the paths representation becomes "1:11;1:2;1:7". Because Node 1 has three outgoing nodes (2, 7 and 11), the paths representation is reduced to "".

We can generalize the rule for paths reduction. Consider a branching node i that has k outgoing branches to nodes $1, 2, \ldots, k$. If a paths representation from Node a to Node b contains at least k paths such that:

- All k paths have a common prefix X, which indicates the same sub-paths from Node a up to Node i;
- Each of the k paths is in the format of "$Xi : j$" for $1 \leq j \leq k$;
- The different suffix of $i : j$ refers to a unique branch of Node i.

Then each sub-path $i : j$ (where $1 \leq j \leq k$) can be reduced to an empty sub-path. Note that the prefix X can be an empty String, in which case Node a is Node i.

After all paths representations from a branching node a are computed, we interpret the corresponding paths representation from Node a to Node b and come up with either of these two conclusions:

1. Node b is control dependent on Node a. For example, the paths representation from Node 1 to Node 3 is "1:2;1:7,8:10"; it cannot be reduced to empty in the end. The presence of path "1:2" implies that all paths from Node 2 (which is one of the three outgoing nodes of Node 1) eventually reach Node 3. The absence of other single-length paths[7] implies that the execution can avoid Node 3 by following other branches from Node 1. Based on the definition of NTSCD, we deduce that Node 3 is control dependent on Node 1.

2. Node b is not control dependent on Node a. It can be the case when they do not satisfy the first condition of the NTSCD definition (i.e., there does not exist a branch from a that is guaranteed to lead to b). For example, consider that the paths representation

[7] A single-length path does not contain comma in its representation.

from Node 1 to Node 10 is "1:7,8:10"; this implies that there is no execution path from Node 1 that is guaranteed to lead to Node 10. Another case is that Node a and Node b do not satisfy the second condition of the NTSCD definition (i.e., there does not exist a branch from a that can possibly avoid leading to b). For example, consider that the paths representation from Node 1 to Node 5 is reduced to empty; this implies that all paths from Node 1 lead to Node 5. In both cases, we determine that Node b is not control dependent on Node a.

4.3.3.4 CD Algorithm: Pseudo-code and Explanations

This section presents FormlSlicer's algorithm to compute control dependence. The main algorithm is shown in Algorithm 4.2.

Algorithm 4.2 examines all the nodes in the CFG in the `foreach` statement at Line 2 and checks whether the node is a branching node at Line 3. For each branching node, `node1`, the algorithm computes an array `p` with each element `i` storing the paths representation from `node1` to the node whose ID is `i`. For each of `node1`'s branches to a successor node `node2`, the element of `p` corresponding to `node2` is initialized with the subpath "`node1:node2`" (Line 6 to Line 9). These successor nodes are added to the `uniqueQ`. Next, in each iteration of the `while` loop at Line 10, a `currNode` is dequeued from the `uniqueQ` and is processed based on one of the three cases discussed below. Here we use the same CFG from Figure 4.9 to explain.

1. If `currNode` is a branching node, as determined at Line 13, then the paths representation of each outgoing node (`n3`) of `currNode` extends each of the paths of `currNode` with a new subpath "`currNode:n3`". Consider `currNode`=Node 8 and `n3`=Node 10 in Figure 4.11a. The paths representation of Node 10 is computed by appending a sub-path "8:10" to every path in Node 8's paths representation (i.e., "1:7"), to show that the paths have branched at Node 8. The function `ExtendPath`, listed in Algorithm B.5, performs this operation.

2. If `currNode` is not a branching node, as determined at Line 19, then the paths representation of the only outgoing node (`n3`) of `currNode` is computed by union-ing pre-existing paths of `n3` with the paths of `currNode`. Union-ing is a necessary step to propagate `currNode`'s paths representation to its successor `n3` without erasing the pre-existing contents of `n3`'s paths representation that are computed from `n3`'s other incoming paths[8]. Consider `currNode`=Node 10 and `n3`=Node 3 in Figure 4.11b. The

[8]Union-ing is performed in this case and not in Case 1. In Case 1, the successor node of a branching node in our CFG must be a TNode and it does not have another incoming path other than the current one.

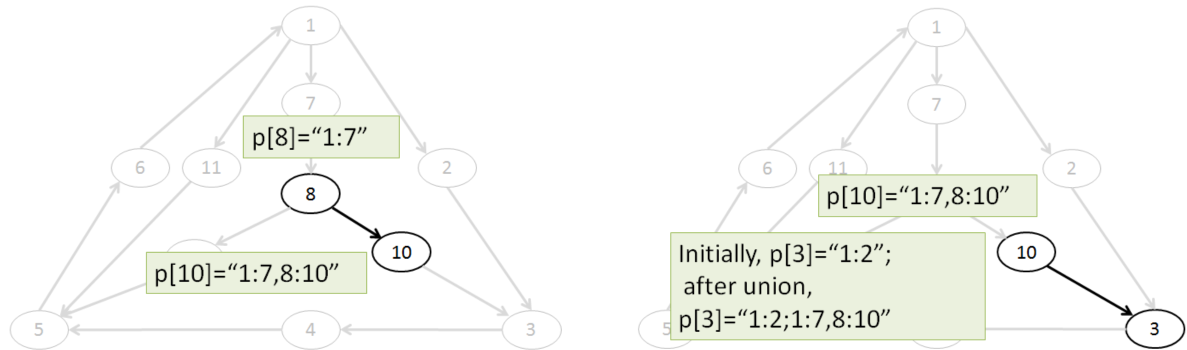
Algorithm 4.2: Main Algorithm of Control Dependence Computation

```
   Input: allNodes
   Output: CDtable
1  set p := array[String]
2  foreach node1 (with ID n1idx) in allNodes do
3      if node1 has 1 outgoing node then continue;
4      set uniqueQ := empty queue // uniqueQ is a queue that ensures uniqueness of elements
5      reset all fields in p to be null
6      foreach node2 (with ID n2idx) in node1.outgoingNodes do
7          p[n2idx] := "n1idx:n2idx"
8          uniqueQ.enqueue(node2)
9      end
10     while uniqueQ is not empty do
11         set currNode := uniqueQ.dequeue();
12         set currIndex := currNode.ID;
13         if currNode.outgoingNodes.size > 1 then
               // currNode has many outgoing nodes
14             foreach n3 (with ID n3idx) in the currNode.outgoingNodes do
15                 ExtendPath(currIndex, n3idx);
16                 ReducePaths(n3idx);
17                 if HasNonEmptyPathsFromNode1 (n3idx) then uniqueQ.enqueue(n3)
18             end
19         else if currNode.outgoingNodes.size == 1 then
20             set n3 (with ID n3idx) := currNode.outgoingNode // currNode has 1 outgoing node
21             if n3 == node1 then continue;
22             set unionPathHappens = UnionPath (currIndex, n3idx)
23             if NOT unionPathHappens then continue;
24             ReducePaths(n3idx);
25             if HasNonEmptyPathsFromNode1 (n3idx) then uniqueQ.enqueue(n3)
26         end
27     end
28     for j from 0 to last index in p do
29         ReducePaths(j);
30         if IsControlDependentOn1 (j) then ADD n1idx to CDtable[j];
31     end
32 end
```

paths representation of Node 3 was initially "1:2" because of Path 4 as shown in Section 4.3.3.2. After union-ing with the paths representation of Node 10, it now becomes "1:2;1:7,8:10". The function `UnionPath`, listed in Algorithm B.4, performs this operation and returns true when there is a change in the paths representation of `n3`.

3. If `currNode` does not have any outgoing nodes, nothing is performed because `currNode` is a terminating node.

At Line 17 and Line 25, the function `HasNonEmptyPathsFromNode1(n3idx)` determines whether `n3` may be a potential node that is control dependent on `node1`, by checking whether the paths representation of `n3` contains non-empty paths after *reduction*.



(a) When `currNode` (Node 8) is a branching node, the paths representation of `currNode`'s outgoing node (Node 10) is computed by extending the paths of `currNode`.

(b) When `currNode` (Node 10) is a non-branching node, add the paths of `currNode` to the paths of `currNode`'s outgoing node (Node 3).

Figure 4.11: Two cases in propagating the paths representation from `currNode` to its neighbor in Algorithm 4.2

As mentioned in Section 4.3.3.3, when all nodes' paths representations from `node1` are computed, the algorithm looks at the final result of `p` to determine whether a node is control dependent on `node1`. This is shown in the `for` loop at Line 28. The function `IsControlDependentOn1` checks whether the paths representation from `node1` to Node `j` contains a single-length path from `node1`; if so, the function returns true because the representation implies that `node1` has multiple outgoing paths such that one path always leads to Node `j` and others are not guaranteed to lead to Node `j`.

The algorithm is long and thus it is modularized into several functions. There are in total six supporting functions for the main algorithm in Algorithm 4.2. Table 4.5 lists all the supporting functions' signatures and their goals.

Function Signature	Goal of Function	Algorithm
`HasNonEmptyPathsFromNode1 (branchIndex)`	It determines whether `p[branchIndex]` has non-empty contents; if so, it is worth to propagate this paths representation to other nodes reachable from Node `branchIndex`.	B.1
`IsControlDependentOn1 (index)`	It interprets `p[index]` to determine whether Node `index` is control dependent on the branching node `node1`.	B.2

ReducePaths (index)	It performs reduction on the paths representation in p[index]. This function utilizes SortPaths(paths) function.	B.3
SortPaths (paths)	It performs an insertion sort on paths, firstly based on length of paths, secondly based on length of sub-paths. Insertion Sort is more efficient because the paths are likely to be in ascending order.	Omitted
UnionPath (prevIdx, nextIdx)	It adds each path in p[prevIdx] as a new distinct path in p[nextIdx] (only when the added path is not a prefix of any path already in p[nextIdx]) so that p[nextIdx] becomes a union of paths from its old value and p[prevIdx]; it returns true when there is a change in the paths representation of nextIdx.	B.4
ExtendPath (srcIndex, branchIndex)	It creates a new paths representation for p[branchIndex] by appending the new sub-path "srcIndex:branchIndex" to each path in p[srcIndex].	B.5

Table 4.5: List of Supporting Functions for Main Algorithm in Algorithm 4.2

The computation results are stored in a table:

CDtable a 1-D table mapping a node n_1 to a set of nodes on which n_1 is control dependent.

Analysis of Algorithm The worst-case time complexity of Algorithm 4.2 is $(|S|+|T|) \cdot |S| \cdot (|T|)^2$ where $|T|$ is the number of TNodes and $|S|$ is the number of SNodes (assuming that all SNodes are branching nodes in the worst case). The `foreach` loop at Line 2 and the `while` loop at Line 10 account for time complexities of $|S|$ and $(|S|+|T|)$, respectively. Among all the helper functions performed in one iteration of the `while` loop, `SortPaths` within `ReducePaths` has a maximum time complexity[9] of n^2, where n is the maximum

[9] Although insertion sort does not have the best time complexity among all sorting algorithms, in practice it is fast in our algorithm because the paths are likely to be in ascending order.

length of a paths representation. Since the length of a paths representation is number of branching nodes times the maximum outdegree of a branching node, and a TNode has only one source SNode, we determine that n is total number of TNodes in worst case. Therefore the complexity at each iteration of the `while` loop is $(|T|)^2$. The algorithm can be terminated because it stops adding more nodes to `uniqueQ`: (1) when there is a loop in the CFG back to the branching node `node1`, shown at Line 21; and (2) when there is a loop back to another branching node other than `node1` along the traversal, shown at Line 23, because it detects that the paths of `n3` are duplicate to the paths of `currNode`.

4.3.4 Summary

In summary, FormlSlicer computes three dependences from the CFGs of an input model and produces the following tables of dependencies:

HDtable1 a 1D-table mapping each SNode to its parent SNode;

HDtable2 a 1D-table mapping each SNode to a set of its default initial child states;

DDtable a 2D-table mapping a pair (a variable v and a TNode t_j) to a set of TNodes, to indicate that t_j is data dependent on a TNode t_i in the set with respect to v;

CDtable a 1-D table mapping a node n_1 to a set of nodes on which n_1 is control dependent.

These tables serve as dictionaries of dependencies that are used by the slicing processes. They are not modified after the preprocessing task.

4.4 Multi-Stage Model Slicing Process

We are now ready to describe the slicing task. FormlSlicer forks off n processes, one for each feature, where each process considers one feature as the FOI, and the other FOSMs as the ROS to be sliced. Each slicing process is divided into multiple stages. FormlSlicer's slicing strategy starts with an empty sliced model of the ROS; at each step in the multi-stage model slicing process, certain model elements (either an SNode or a TNode) from the original model are added to the sliced model. Thus, as slicing progresses, the FOI remains unchanged and the ROS in the sliced model grows gradually.

Figure 4.12 shows the relationship between an original model and a corresponding sliced model. Some FOSMs in the ROS are entirely absent in the sliced model and some are partially absent.

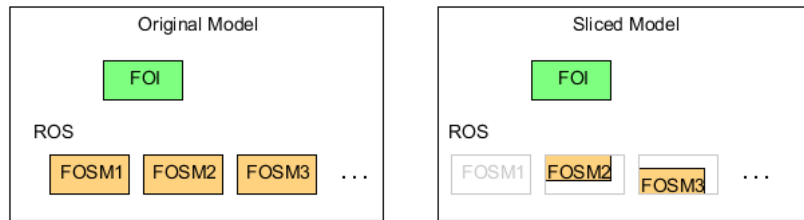


Figure 4.12: A comparison between the original model and the sliced model

We have a multi-stage model slicing process:

Initiation Stage
: This stage adds to the sliced model the initial set of TNodes that perform actions on variables that the FOI directly monitors.

General Iterative Slicing Stage
: This stage adds the SNodes and TNodes needed to preserve the dependencies that were computed from the preprocessing task.

Model Restructuring Stage
: This stage performs additional transformations to ensure that the sliced model is a well-formed state machine.

In the following, we use a running example to illustrate each step of our multi-stage model slicing process, as shown in Figure 4.13. For simplicity, the example model we use

from Section 4.4.1 to Section 4.4.3 consists of two FOSMs—*E1* and *E2*, where *E1* is the FOI and *E2* is an FOSM in the ROS; but the process generalizes to multiple FOSMs in the ROS. Table 4.6 shows the monitored and controlled variables of all transitions in the model.

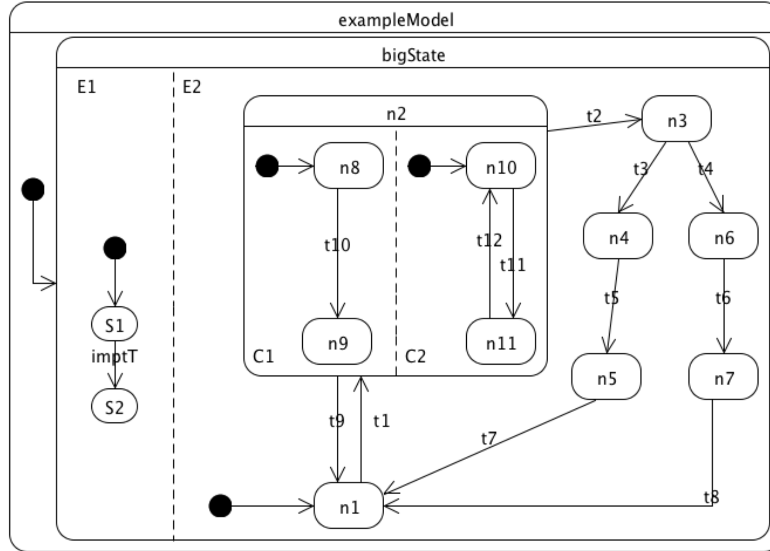


Figure 4.13: The model used as the running example

Transition	Monitored Variables	Controlled Variables
imptT	v1	v2
t1	v3	v2
t2	+WCE1	
t3	+WCE3, WCE3.att	
t4	+WCE3, WCE3.att	
t5	v4	v4
t6	v4	v4
t7	+WCE1	v3
t8	+WCE1	
t9	v5	+WCE4
t10	E2.main.n2.C2.n11	+WCE4
t11	v2	v1, E2.main.n2.C2.n11
t12	+WCE1	

Table 4.6: The Monitored and Controlled Variables of Transitions in the Running Example

Slicing is performed on the CFG structure (Figure 4.14), but we also show the resulting

FOSM side-by-side to show the effects of slicing at each step. From Section 4.4.1 to Section 4.4.3, elements that are newly added to the sliced model are highlighted in red; existing elements in the sliced model are shown in black; elements that are not in the sliced model are colored in grey. For brevity, if an element is in the sliced model, we say that it is *part-of-slice*; we say that an element is *out-of-slice* if it is not in the sliced model.

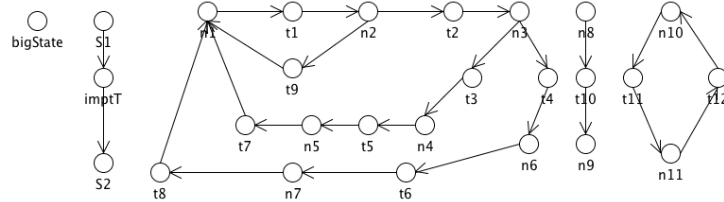


Figure 4.14: The control flow graph of the model used as the running example

4.4.1 Initiation Stage

This stage adds the initial set of TNodes to the sliced model in ROS, based on what variables the FOI monitors.

4.4.1.1 Variable Extraction Step

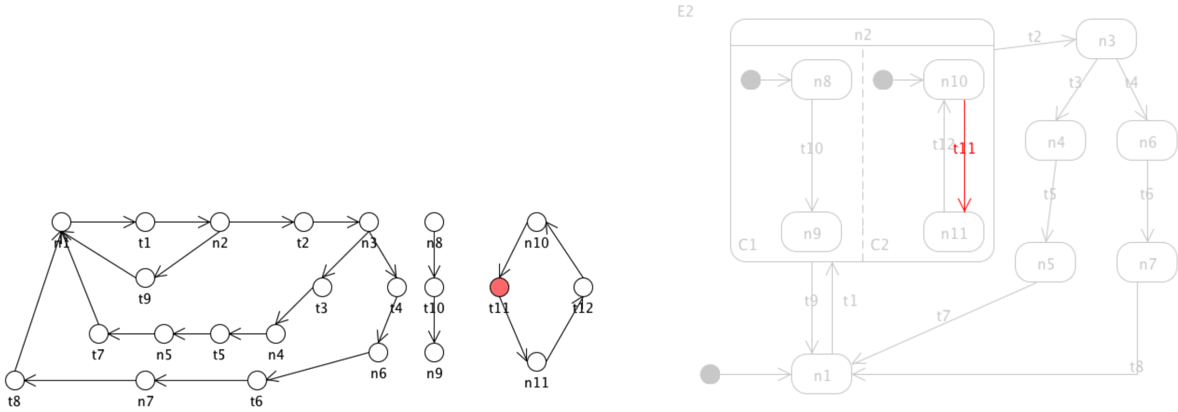
The monitored variables of the FOI are the only slicing criteria used to select the initial set of transitions in the ROS. These variables are used to identify which actions (i.e., transitions) in the ROS can potentially influence the behavior of the FOI.

FormlSlicer includes a *Variable Extractor* that inputs the CFGs of the FOI and extracts all of the TNodes' monitored variables. In the end, it outputs a set of variables that are relevant to the FOI. We denote this set of variables as V_{Rv} and call them **relevant variables**. In our running example (Figure 4.13), $V_{Rv} = [v1]$.

4.4.1.2 Initial Transition Selection Step

This step adds any transitions (TNodes) in the ROS that (potentially) **directly** affects execution of the FOI by acting on one of the FOI's monitored variables.

Initially, the sliced ROS is empty. During the initial transition selection step, Forml-Slicer adds to the sliced ROS all of the TNodes in the ROS that control any relevant variable. From Table 4.6, we know that transition *t11* controls relevant variable $v1$, and thus is added to the sliced model. The monitored variables of *t11* (i.e., $v2$) are added to the set of relevant variables to ensure proper execution of transition *t11*. Now, $V_{Rv} = [v1, v2]$.



(a) The control flow graph of the ROS of the running example

(b) The FOSM representation of the sliced ROS

Figure 4.15: The sliced ROS after performing the initial transition selection step

4.4.2 General Iterative Slicing Stage

This stage adds to the sliced model any TNodes or SNodes that **indirectly** affects the execution of the FOI by directly or indirectly affecting the execution of a part-of-slice transition (TNode) in the ROS. This stage preserves the dependencies identified in the preprocessing task.

The general iterative slicing stage consists of four steps (Figure 4.16):

DD Step
Add more TNodes that any part-of-slice TNode is (possibly transitively) data dependent on with respect to any relevant variable;

Cross-Hierarchy Transition Step
Add all cross-hierarchy transitions that are still out-of-slice to the sliced model but replace their labels;

Transition-to-State Step

Add SNodes that correspond to the source and destination states of part-of-slice TNodes;

CD-HD Step

Add more SNodes that any part-of-slice SNodes and TNodes are control dependent or hierarchy dependent on.

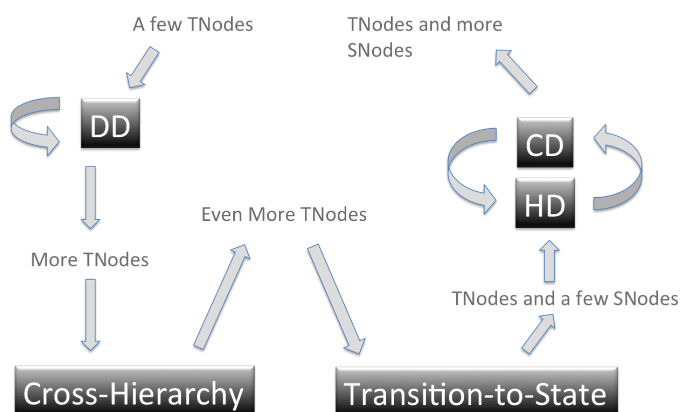


Figure 4.16: Four steps in general iterative slicing stage

Both the DD step and CD-HD step add nodes to the slice **iteratively**, in order to add nodes that any part-of-slice nodes are **transitively** dependent on.

4.4.2.1 DD Step

The DD step ensures that the values of important variables in executions of the sliced model match their values in corresponding executions of the original model. It does so by adding to the slice all transitions that act on important variables, and all transitions that act on variables used in important variables' assignment expressions, and all transitions that act on variables used in these variables' assignment expression, and so on.

The DD step starts with V_{Rv} and the partial sliced model with a few TNodes from the previous step. Then it iterates repeatedly to (potentially) add more TNodes into the sliced model, and (potentially) enlarge the size of V_{Rv}. It terminates when no more changes to V_{Rv} are possible.

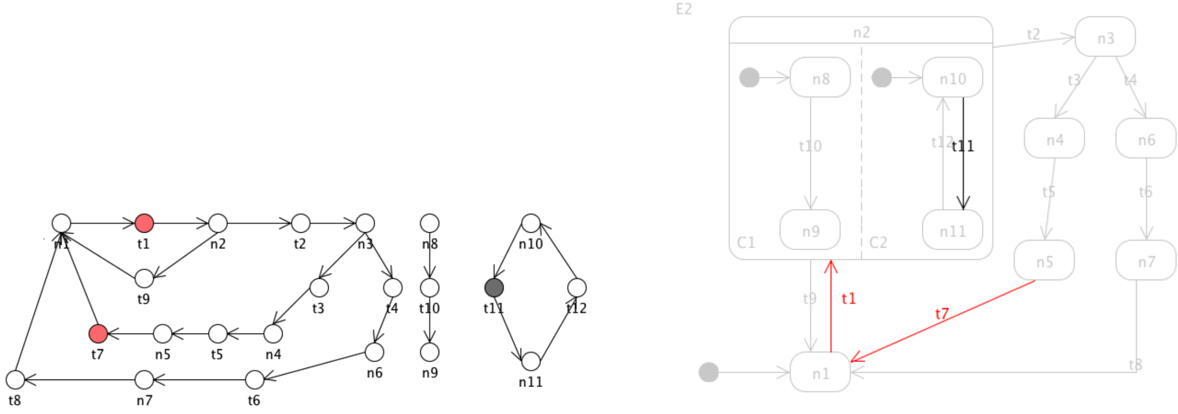
Algorithm 4.3: DD step in general iterative slicing stage

```
   Input: sliceSet, inputRelVars, DDtable
   Output: sliceSet, outputRelVars
 1 set outputRelVars := empty set
 2 repeat
 3    outputRelVars.add(inputRelVars)
 4    set tempRV := empty set
 5    foreach v in inputRelVars do
 6       foreach tnode in sliceSet do
 7          if DDtable[(v,tnode)] is null then continue;
 8          foreach anotherTNode in DDtable[(v,tnode)] do
 9             if sliceSet contains anotherTNode then continue;
10             sliceSet.add(anotherTNode);
11             foreach m in anotherTNode.monitoredVars do
12                if outputRelVars does not contain m then tempRV.add(m);
13             end
14          end
15       end
16    end
17    set inputRelVars := tempRV
18 until inputRelVars is empty
```

Algorithm 4.3 shows how the DD step works. Its inputs include the `inputRelVars` and the `sliceSet`, which represent V_{Rv} and the partial sliced model, respectively. At the end of each iteration of the `repeat` loop (Line 2), the `outputRelVars` contains the cumulative set of relevant variables that are found so far, whilst the `inputRelVars` contains the relevant variables that are newly found in the current iteration and ready to be used in the next iteration. For each `v` in the `inputRelVars` and for each `tnode` in the `sliceSet`, the algorithm consults `DDtable` to determine if `tnode` is data dependent on some other TNodes with respect to `v` (Line 7); if so, each of those other TNodes, `anotherTNode`, is added to the sliced model (Line 10) provided that it is not in the sliced model; in addition, each of the monitored variables of `anotherTNode`, `m`, is added to `tempRV` provided that `m` is not already in `outputRelVars` (Line 12). At the end of each iteration (Line 18), the algorithm checks whether any new variables have been added to `inputRelVars`, because `tempRV` may grow from an empty set to a non-empty set in that iteration before being assigned to `inputRelVars`; if so, the algorithm continues to search for more TNodes.

Analysis of Algorithm The worst-case time complexity of Algorithm 4.3 is $|V| \cdot |T|^2$ where $|V|$ is the number of variables and $|T|$ is the number of TNodes. The two outermost loops (`repeat` loop at Line 2 and `foreach` loop at Line 5) account for only a total complexity of $|V|$ because there is only a finite number of variables; once a variable is added to `inputRelVars` in a particular iteration, it can never be added again. The two `foreach` loops at Line 6 and at Line 8 account for a complexity of $|T|^2$. The statements

within the `foreach` loop from Line 10 to Line 13 execute a maximum of $|T|$ times in the overall algorithm, because once a TNode is added to `sliceSet` (Line 10), it can never be added again because of the check (Line 9); therefore, it does not increase the overall time complexity. Also, the statement at Line 9 is constant because we use a boolean array to flag whether a TNode is part-of-slice.



(a) The control flow graph of the ROS in the running example

(b) The FOSM representation of the sliced ROS

Figure 4.17: The sliced ROS after performing the DD step

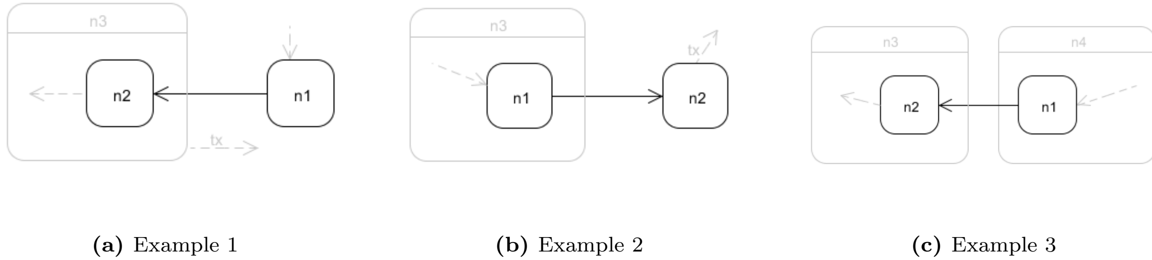
Reconsider the example in Figure 4.15. We know that $V_{Rv} = [v1, v2]$ and $t11$ is in the sliced model from the previous step. In Table 4.6, we can see that transition *t1* has a controlled variable $v2$ and a monitored variable $v3$. Then, *t11* is data dependent on *t1* with respect to $v2$; and thus *t1* is added to the sliced model as well. However, the DD step cannot simply terminate here: because *v3* is monitored by *t1*, this variable is important to the sliced model. If there is another transition (e.g., *t7*) that controls $v3$, then that transition affects *t1* and indirectly affects *t11*, which indirectly affects the FOI. Therefore, we add $v3$ to V_{Rv} and continue searching for more TNodes that any part-of-slice TNode are data dependent on.

At the end of the DD step, the sliced model contains TNodes *t1* and *t7*, as shown in Figure 4.17.

4.4.2.2 Cross-Hierarchy Transition Step

A cross-hierarchy transition is a transition that crosses a hierarchy boundary, such that its source state and destination state do not have a common parent state. A cross-hierarchy

transition may transit from a source state outside of a composite state to a destination state inside the composite state (Figure 4.18a); or it may transit from a source state within a composite state to a destination state that is outside of the composite state (Figure 4.18b); or it may transit from a source state within one composite state to a destination state that lies inside another composite state (Figure 4.18c). In the last case, the destination state's rank may be higher than, lower than, or the same as the source state's rank.



(a) Example 1 **(b)** Example 2 **(c)** Example 3

Figure 4.18: Examples of cross-hierarchy transitions

The sliced model needs to include these cross-hierarchy transitions in order to correctly simulate the original model and be as precise as possible. Consider such a composite state n_{cps} (e.g., *n3* or *n4* in Figure 4.18) and cross-hierarchy transition t_{ch}; note that n_{cps} is neither the source state nor the destination state of t_{ch}, but it is an ancestor state of one of them.

In the first case, t_{ch} crosses into the inside of n_{cps}; if n_{cps} is in the sliced model due to some other reason (e.g., some other node is control dependent on it) and FormlSlicer does not include t_{ch} in the slice, then n_{cps} will not be entered in the same manner as the original model, and as a consequence the execution of any transitions from n_{cps} becomes impossible. This causes the sliced model to become incorrect. For example, in Figure 4.18a, if *n3* and *tx* are in the slice and the cross-hierarchy transition is not, then *tx* cannot be executed correctly in the sliced model after the cross-hierarchy transition.

In the second case, t_{ch} exits from the inside of n_{cps} to reach another state n_{dsch}; if n_{cps} and n_{dsch} are in the sliced model due to some other reason and t_{ch} is not, then n_{dsch} will not be entered in the same manner as the original model, and as a consequence the execution of any transitions from n_{dsch} becomes impossible. As an example, in Figure 4.18b, if *n2*, *n3* and *tx* are in the slice but the cross-hierarchy transition is not, then *tx* cannot be executed correctly in the sliced model after the cross-hierarchy transition. Note that the assumptions in this case is asymmetric to the first case. Although it seems that there are possible ways to fix a cross-hierarchy transition exiting from the inside of a composite

state (e.g., creating a transition from n_{cps} to n_{dsch}), all of them bring more complexities to the later slicing steps that can change the model structure; since we cannot provide a solution of which we can prove the correctness, we decide to adopt a conservative approach by preserving the cross-hierarchy transition and leave this for future investigation.

FormlSlicer performs this step after the DD step, by looking for any out-of-slice cross-hierarchy transitions in the original model, and for each one adds the transition to the slice but labels the added transition with only a "true" guard condition. The added transition is called a **"true" transition**.

We use a "true" transition because this transition is guaranteed not to control any relevant variables. The previous step—DD step—has already identified all the transitions which transitively control the relevant variables, thus any out-of-slice transitions at this step do not affect the relevant variables. Thus, We can safely ignore their monitored or controlled values. However, we note that ignoring monitored variables introduces imprecision because transitions in the slice can be executed under conditions that transitions in the original model will not execute.

4.4.2.3 Transition-to-State Step

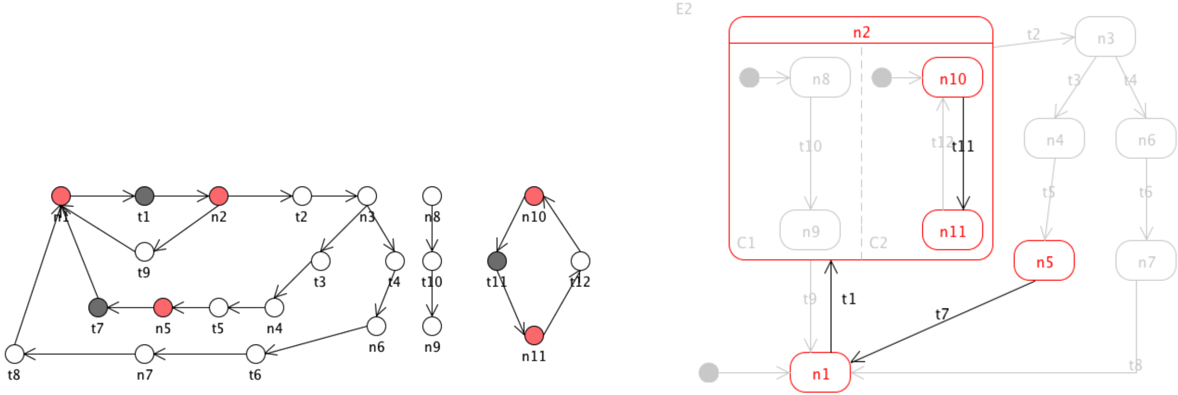
So far, all the model elements that have been added to the sliced model are TNodes (equivalent to transitions). The transition-to-state step adds associated SNodes (equivalent to states). Specifically, for each part-of-slice transition, this step adds its source state and destination state to the slice. Figure 4.19 shows the effect of this step on the running example.

Although this step is simple, it is very important. Firstly, it enriches the fragmented sliced model so that it moves one step closer to being a well-formed model. Secondly, it introduces an initial set of SNodes into the sliced model, providing a starting point for the CD-HD step.

4.4.2.4 CD-HD Step

The CD-HD step adds more SNodes to the sliced model to ensure that (1) any state that a part-of-slice state is control dependent on is also in the sliced model, and (2) all ancestor states of a part-of-slice state are also in the sliced model.

Algorithm 4.4 shows how the CD-HD step works. Similar to the algorithm for the DD step, it is a fixed point computation that terminates only when an iteration makes



(a) The Control Flow Graph of the Example FOSM in the ROS

(b) The Example FOSM in the ROS

Figure 4.19: The Example FOSM in the ROS after Transition-to-State Step

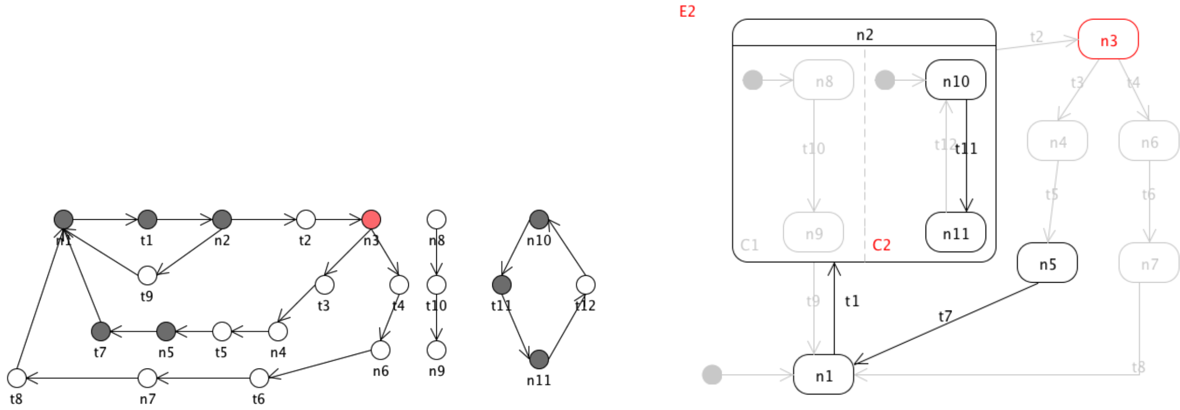
no more changes to the sliced model. In each iteration of the `repeat` loop, there are two dependency lookups: at Line 5, the algorithm consults `HDtable1` and adds an SNode to `changedSliceSet` if one of its child SNodes are part-of-slice; at Line 8, the algorithm consults `CDtable` and adds any SNodes that control the execution of another part-of-slice node to `changedSliceSet`. If there are no more new nodes added to `changedSliceSet`, the algorithm terminates (Line 11).

Analysis of Algorithm The worst-case time complexity of Algorithm 4.4 is $|S|$ where $|S|$ is the number of SNodes. The `repeat` loop has maximum $|S|$ iterations, because there is a finite number of SNodes and once an SNode is added to `changedSliceSet`, it can never be added again. In addition, because we use a boolean array to flag whether an SNode is part-of-slice, the checks performed at Line 5 and at Line 8 have a constant time complexity.

Figure 4.20 shows the sliced ROS of our running example after performing the CD-HD step. The state *n3* has been added to the sliced model because *n5* is control dependent on it.

Algorithm 4.4: CD-HD step in general iterative slicing stage

```
   Input: sliceSet, HDtable1, CDtable
   Output: SliceSet
 1 set changedSliceSet := empty set
 2 repeat
 3 |   sliceSet.add(changedSliceSet)
 4 |   set changedSliceSet := empty set
 5 |   if HDtable1[node] is not null AND sliceSet contains node then
 6 |   |   changedSliceSet.add(HDtable1[node]);
 7 |   end
 8 |   if CDtable[node] is not null AND sliceSet contains node then
 9 |   |   changedSliceSet.addAll(CDtable[node]);
10 |   end
11 until changedSliceSet is empty
```



(a) The control flow graph of the ROS of the running example

(b) The FOSM representation of the sliced ROS

Figure 4.20: The sliced ROS after performing the CD-HD step

4.4.3 Model Restructuring Stage

The purpose of this stage is to rewire the sliced model so that its structure may become different from the original model.

There are two steps in this stage:

State Merging Step that merges two neighbor states if their distinction is not important

to slicing criteria or dependences;

State Connecting Step that connects any two part-of-slice states with a missing path in the sliced model using a "true" transition.

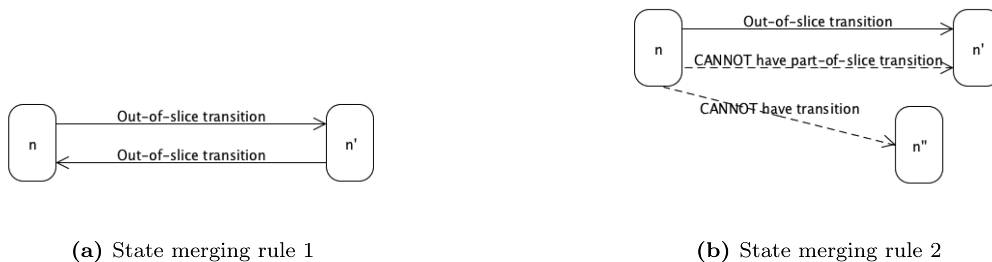
4.4.3.1 State Merging Step

The state merging step merges two states if there is no reason to keep them distinct. FormlSlicer uses Korel et al.'s two state merging rules that satisfy the *traversability* property[10] [6] (Figure 4.21):

Rule 1 If all transitions from state n to n' and also from state n' to n are out-of-slice, these two states are merged into one state "n,n'".

Rule 2 States n and n' can be merged into one state "n,n'" if:

1. There exists an out-of-slice transition from n to n',
2. There does not exist a part-of-slice transition from n to n', and
3. There is no outgoing transition from n to n'' where $n'' \neq n'$.



(a) State merging rule 1

(b) State merging rule 2

Figure 4.21: Illustration of state merging rules

Intuitively, in Rule 1, the distinction between two states is not important if all transitions between the two states are out-of-slice and thus they are not important. In Rule 2, the machine in state n cannot move anywhere except to n'; therefore, it is safe to merge n and n'.

FormlSlicer adjusts the two rules in two minor ways:

[10]See Section 2.3.2 about the traversability property on a slice.

1. It will perform state merging only when at least one of the two states is part-of-slice. Otherwise, it is a waste of effort because in the end the states will not appear in the sliced model.

2. It will perform state merging only when these two states have the same parent state, to preserve the model's state hierarchy.

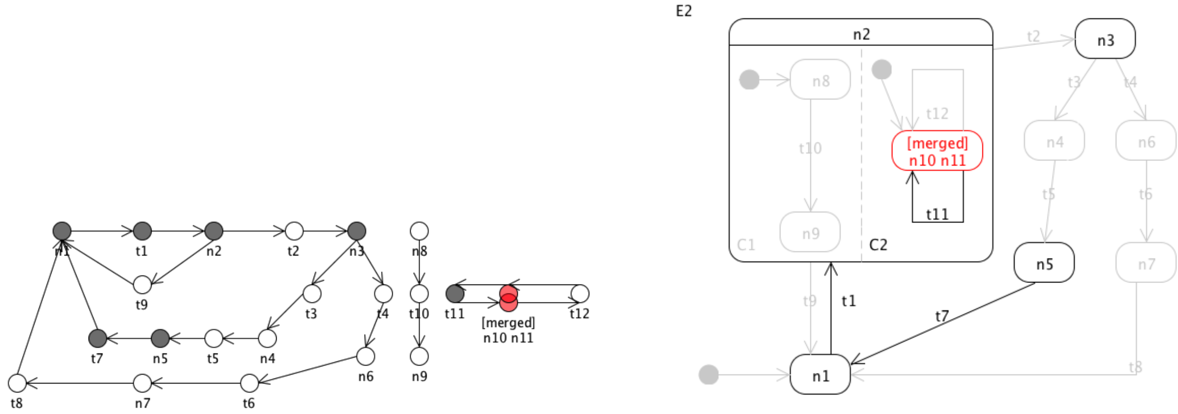
In our implementation, the main algorithm scans through all SNodes and for each part-of-slice SNode n that has not yet participated in state merging, it calls a function *MergeNeighborSNodes*. This function scans through all n's outgoing TNodes to look for a destination SNode n' that is eligible for state merging with n. If the function finds that n' and n satisfy either one of the two rules and have the same parent, it creates a new state n_{merged} such that (1) all of n's and n''s outgoing TNodes become n_{merged}'s outgoing TNodes, (2) all of n's and n''s incoming TNodes become n_{merged}'s incoming TNodes, (3) all of n's (or n''s) child nodes become n_{merged}'s child nodes if n (or n') is a composite state, (4) n_{merged} becomes the default initial state in the region if n or n' is the original default initial state, and (5) n_{merged} has a state name that combines the names of n and n'. After state merging, *MergeNeighborSNodes* continues to look for another successor SNode n'' that satisfy the merging criteria with n_{merged}; if it can find such an SNode, the merging process continues; otherwise, it stops checking and returns the function. The main algorithm then continues the SNode scanning to restarts the whole process on another part-of-slice SNode that has not yet participated in state merging.

The worst-case time complexity is kept within $|S| \cdot |T|$ where $|S|$ is the number of SNodes and $|T|$ is the number of TNodes. Because there is a finite number of states, the algorithm can only perform a maximum number of $|S|$ state merging operation. The complexity of each state merging operation depends on the number of outgoing transitions; here we over-estimate it to be $|T|$.

Figure 4.22 shows sliced ROS of our running example after performing the state merging step. State *n11* has only one out-of-slice transition to *n10* and therefore these two states satisfy Rule 2. They are merged together to become a new state "*[merged] n10 n11*".

4.4.3.2 State Connecting Step

This is the last step of the entire multi-stage model slicing process. The goal of this step is to add transitions between disconnected states in the sliced model, such that the sliced model preserves the reachability among states in the original model.



(a) The control flow graph of the ROS of the running example

(b) The FOSM representation of the sliced ROS

Figure 4.22: The sliced ROS after performing the stage merging step

This step uses the concept of next part-of-slice state: let n be a part-of-slice state, and n' be another part-of-slice state that is reachable from n via a path such that all the states and transitions along this path are missing in the slice and all the states are at the same rank of state hierarchy, then n' is called the **next part-of-slice state** of n through a path missing in the slice.

There are two sub-steps:

Sub-step 1: Search for New Default Initial States Recall that each sub-machine has exactly one default initial state (Section 3.3). If the default initial state is out-of-slice, this sub-step searches for a new default initial state for the sub-machine.

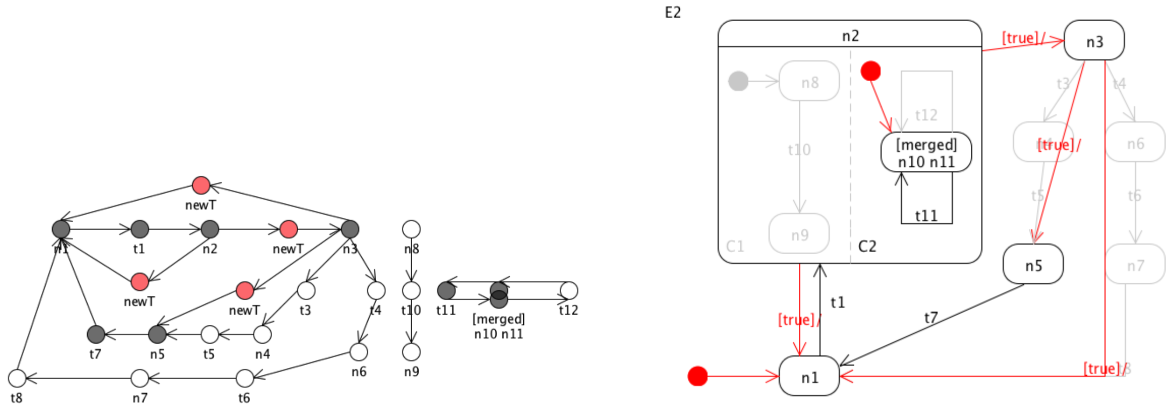
This sub-step benefits greatly from the use of control dependence. Sometimes, a default initial state has more than one outgoing transitions and when it is out-of-slice, FormlSlicer may run into the trouble of determining a new default initial state. Consider a default initial state n_i that has two successor states n_x and n_y (i.e., there is a transition from n_i to n_x and another transition from n_i to n_y). If n_x and n_y are selected into the sliced model, but n_i is not, then both n_x and n_y are eligible to be the new default initial state. However, the use of control dependence prevents this trouble from happening, because n_i must be added to the sliced model during the CD-HD step.

In our implementation, this step scans through all part-of-slice composite states and for each composite state n_{cps}, checks whether the default initial state of each of n_{cps}'s

containing sub-machines is part-of-slice; if not so, it performs a breadth-first search to find the nearest part-of-slice state and appoint it to be the new default initial state in the sliced model. As a result of this step, for each sub-machine in the original model, its default initial state in the sliced model is:

1. Either the same state in the original model,
2. Or the next-part-of-slice state of the original default initial state;
3. Or absent, because the whole sub-machine is out-of-slice.

The worst-case time complexity of this sub-step is $|S + T|$ where $|S|$ is the number of SNodes and $|T|$ is the number of TNodes. The two previous steps—the cross-hierarchy transition step and the transition-to-state step—prevent the breadth-first search in this sub-step from leaving the inside of a composite state, because an out-of-slice path cannot extend beyond the hierarchy boundary, and thus constrain the time complexity to be linear.



(a) The control flow graph of the ROS of the running example

(b) The FOSM representation of the sliced ROS

Figure 4.23: The sliced ROS after the state connecting step

Sub-step 2: Search for Next Part-of-slice State This sub-step performs a depth-first-search from any part-of-slice state and finds all its next-part-of-slice states. It creates a “true” transition between the part-of-slice state and each of its next-part-of-slice states and adds the “true” transition to the sliced ROS.

The worst-case time complexity of this sub-step is $|S| \cdot |S + T|$ where $|S|$ is the number of SNodes and $|T|$ is the number of TNodes. The sub-step can terminate because during the depth-first-search from an SNode, we use a flag to indicate whether a node has been visited in current round in order to prevent repeated visits.

Figure 4.23 shows the slicing example after this step. We can see that there is a "true" transition from *n3* to each of its next part-of-slice states, *n1* and *n5*. In fact, the path between a part-of-slice state and its next part-of-slice state can be as short as one transition. For example, *n2* has an out-of-slice path to *n3* and it consists of only one transition, *t2*; it is replaced by a "true" transition.

4.4.4 More Examples

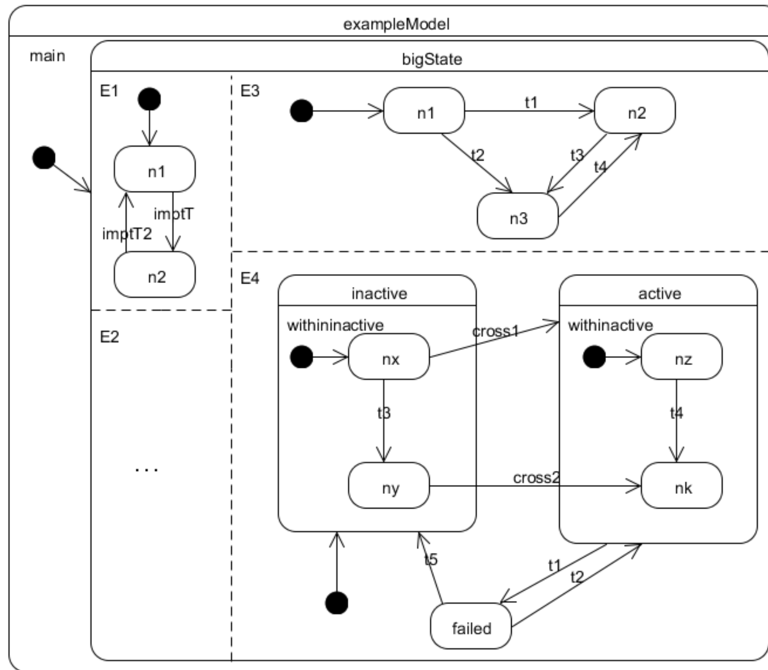


Figure 4.24: The example model with more contents

From Section 4.4.1 to Section 4.4.3, we have explained the majority of steps in the multi-stage model slicing process using a running example. However, there are a few corner cases that are not covered in the running example; so we add two more FOSMs to the ROS of

our example—*E3* and *E4*—and one more transition to the FOI of our example—*E1*, as shown in Figure 4.24. Table 4.7 shows the monitored and controlled variables of the added transitions. Figure 4.25 shows the sliced model with respect to *E1*.

FOSM	Transition	Monitored Variables	Controlled Variables
E1	imptT2	E4.main.failed	
E3	t1	v1	v1
	t2	v1	v1
	t3	+WCE1	
	t4	+WCE6	
E4	t1	failedAtt	E4.main.failed
	t2	failedAtt	
	t3	x	
	t4	y	
	t5	+UnknownEvent	
	cross1	z	
	cross2	z	

Table 4.7: The Monitored and Controlled Variables of More Transitions in the Example

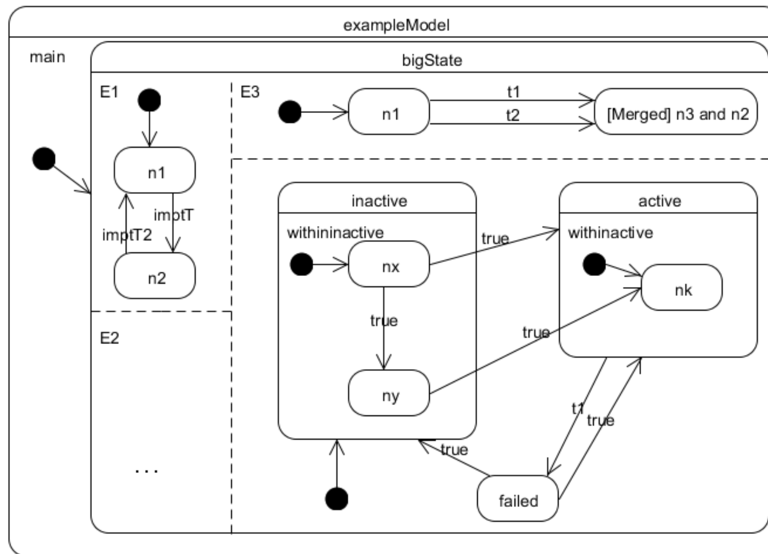


Figure 4.25: The sliced example model w.r.t. *E1*

Here are the steps in the multi-stage model slicing process that are not covered in our running example:

Cross-Hierarchy Transition Step

This is a step in the general iterative slicing stage. As explained in Section 4.4.2.2, any cross-hierarchy transitions will be retained in the sliced model. There are two cross-hierarchy transitions in the FOSM *E4*: *cross1* transits from the state *nx* within the state *inactive* to the state *active*; *cross2* transits from the state *ny* within *inactive* to the state *nk* within *active*. Both *cross1* and *cross2* are preserved in the sliced FOSM.

Hierarchy Dependence in CD-HD Step

Section 4.4.2.4 shows the slicing effect of using control dependence only. In Figure 4.25 we can see the effect of using hierarchy dependence: state *inactive* is added to the sliced model because it is a parent state of *nx* and *ny*.

State Transition Rule 1

Section 4.4.3.1 presents two state merging rules. In Rule 1, if all transitions from state n to n' and also from state n' to n are out-of-slice, these two states are merged into one state "n,n'". This is shown in the FOSM *E3*. All transitions between *n2* and *n3* (i.e., *t3* and *t4*) are out-of-slice; therefore *n2* and *n3* are merged.

Sub-step 1: Search for New Default Initial States

Section 4.4.3.2 presents the state connecting step in the model restructuring stage. There are two sub-steps. The first sub-step is to search for the new default initial state for the sub-machine. In the FOSM *E4*, the state *nz* and the transition *t4* are not part-of-slice, making the state *nk* eligible to become the new default initial state in the region.

4.4.5 Summary

Eventually, FormlSlicer converts the resultant CFGs into an FOSM, as shown in Figure 4.23b and writes the sliced model to a text file in the same format as Table 4.1. **The resultant ROS and the FOI (which remains unchanged throughout the slicing process) form the sliced model.**

In summary, this is a list of all the steps performed by FormlSlicer:

1. Preprocessing Task
    (a) Parse and Convert the input model to CFGs

(b) Compute Hierarchy Dependence
(c) Compute Data Dependence
(d) Compute Control Dependence

2. Slicing Task
    (a) Initiation Stage
        i. Variable Extraction Step
        ii. Initial Transition Selection Step
    (b) General Iterative Slicing Stage
        i. DD Step
        ii. Cross-Hierarchy Transition Step
        iii. Transition-to-State Step
        iv. CD-HD Step
    (c) Model Restructuring Stage
        i. State Merging Step
        ii. State Connecting Step
            A. Sub-step 1: Search for New Default Initial States
            B. Sub-step 2: Search for Next Part-of-slice State

Chapter 5

Correctness of FormlSlicer

This chapter presents a correctness proof for the sliced model produced by the multi-stage model slicing process.

5.1 Overview

As the sliced model is used to replace the original model for safety property checking, the sliced model must guarantee that

$$M_{ROS_{\mathcal{L}}+FOI} \models \varphi \quad \Rightarrow \quad M_{ROS+FOI} \models \varphi$$

whereby $M_{ROS+FOI}$ is the original model comprising the feature of interest (FOI) and the rest of the system (ROS) (i.e., all the feature-oriented state machines except the FOI state machine), $M_{ROS_{\mathcal{L}}+FOI}$ is the sliced model comprising the FOI and the sliced ROS, and φ is the safety property for the FOI. In other words, if a safety property is maintained in all execution traces in the sliced model, we can confidently claim that the safety property is maintained in all execution traces in the original model. This is equivalent to saying that **the execution traces of an FOI in the original model is a subset of the execution traces of the FOI in the sliced model**. It is acceptable if there are some execution traces in the sliced model that are impossible to occur in the original model[1].

In order to show this, we want to prove that any given execution trace in the original model can be *simulated* by an execution trace in the sliced model. As an execution trace

[1] This is the basic correctness property of a sliced model. See Section 2.3 on discussions in the literature about correctness of SBM slicing.

is a sequence of execution steps, this means that for each execution trace in the original model there exists a simulating trace in the sliced model, such that one step in the original model's execution trace can be projected to one step in the sliced model's execution trace.

The rest of this chapter is organized as follows. Section 5.2 defines terminology that will be used in the proof; this section introduces some important concepts, such as state configuration and interpretation. Section 5.3 describes the state transition rule that is standard to a hierarchical and concurrent state machine. In Section 5.4, we formalize the steps in the multi-stage model slicing process using the terminology defined in Section 5.2. Section 5.5 describes the projection of snapshot, transition and execution step from the original model to the sliced model before showing the inductive proof of simulation.

5.2 Terminology

5.2.1 Variable, State, Region, Transition and Model

This section introduces terminology that will be used in the proof.

Variable

A single variable is denoted as v; a set of variables is usually denoted as V with appropriate subscripts. We use V_{env} to refer to the set of environment-controlled variables and V_{sys} to refer to the set of system-controlled variables[2].

State

A state is denoted as either n or m; sometimes p is used to denote a parent state. A set of states is denoted as N when it refers to a state configuration; otherwise, it is denoted as S. Among them, S^{pseudo} is the set of all pseudo states and n^{pseudo} denotes one of them. A composite state p can have many child states $n_1, \ldots, n_k$, denoted as the set $ChildStates(p) = \{n_1, \ldots, n_k\}$. A state's parent state is denoted as $ParentState(n_1) = p$.

[2]See Section 3.1 on definitions of environment-controlled variables and system-controlled variables.

Region

We use r to denote an orthogonal region. If r is inside a composite state n, then $n = ParentStateOfRegion(r)$ and $r \in ChildRegions(n)$. A composite state may have multiple child regions, which are sibling regions:

Definition 1. *Two orthogonal regions r and r' are said to be sibling regions, denoted as $r \parallel r'$, when $ParentStateOfRegion(r) = ParentStateOfRegion(r')$ and $r \neq r'$.*

If region r contains a sub-machine such that n is a state in this sub-machine, then $ParentRegion(n) = r$.

Transition

A transition is a progression in a model's execution from one state (called the transition's source state) to another state (called the transition's destination state).

Definition 2. *We write $n \xrightarrow{t} n'$ to denote a **transition** t from state n to state n'. The source state of t, n, is denoted as $ss(t)$; the destination state of t, n', is denoted as $ds(t)$.*

The source state and destination state of t do **not** need to be at the same rank of the machine's state hierarchy. For simplicity, we do not consider models that have a transition that enters or exits the boundary of an orthogonal region that has a sibling orthogonal region[3].

Monitored and Controlled Variables The set of monitored variables and controlled variables of a transition t are denoted as $mv(t)$ and $cv(t)$, respectively.

Model

As introduced in Section 3.3, FormlSlicer's input model is a big state machine in which many FOSMs are executing in parallel, each modeling the behavior of a distinct feature. We use M to denote a model. As M contains k FOSMs, $\{F_1, \ldots, F_k\}$, we write it as:

$$M = \begin{matrix} F_1 \\ \vdots \\ F_k \end{matrix}$$

[3] See Section 3.3 for more details on this restriction.

5.2.2 State Configuration and Interpretation

An FOSM F is a well-formed state machine. It consists of a finite set of states, S_F, and a finite set of orthogonal regions, R_F. The states and regions form a containment hierarchy: each composite state $n \in S_F$ contains one or more regions; each region contains a sub-machine. S_F^I is the set of default initial states in F and $S_F^I \subseteq S_F$. F also consists of a finite set of transitions, T_F, each starting from a source state $ss(t) \in S_F$ and ending at a destination state $ds(t) \in S_F$. It also contains a set of variables, V_F.

Since a model M is a big state machine, it also consists of a finite set of states (S_M), a finite set of regions (R_M), a finite set of transitions (T_M) and a finite set of variables (V_M). Its set of default initial states is $S_M^I \subseteq S_M$. For any FOSM F contained within one orthogonal region of M, $S_F \subset S_M \wedge R_F \subset R_M \wedge T_F \subset T_M \wedge V_F \subset V_M$.

In this thesis, we can only provide an informal description of the syntax of model M. Due to the complex nature of our model, it is not a trivial task to precisely define the semantics of M.

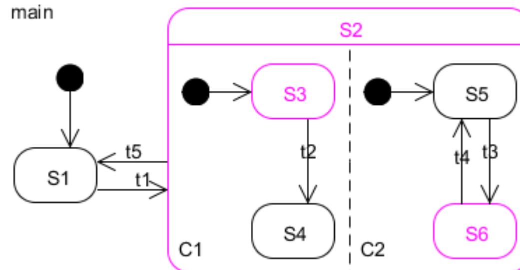


Figure 5.1: The FORML example from Figure 3.1 with its current states highlighted in pink

The set of current states in the execution of a state machine F is its **state configuration**, denoted by N_F (See Section 3.2). The state configuration cannot be any arbitrary combination of states. Rather, N is the set of current states $N \subseteq S_F$ such that if any state n is in N, then so are all of n's ancestors; and if any composite state n is in N, then for each r in $ChildRegions(n)$, one of r's child states must be in N [32]. Figure 5.1 shows an example of an FOSM whose current states are colored pink: $S2$, $S3$, and $S6$. The state configuration is $N = \{S2, S3, S6\}$.

We use σ to denote an **interpretation** which maps variables to their values; thus, $\sigma_M(v)$ represents the current value of the variable v in the model M.

5.2.3 Dependences

As introduced in Section 4.3, there are three types of dependences used by FormlSlicer: hierarchy dependence (HD), data dependence (DD), and control dependence (CD).

Definition 3. *We say $n \in S_M$ is* ***hierarchy dependent*** *on $n' \in S_M$, denoted as $n \xrightarrow{hd} n'$, iff n is a child state of n''s containing region.*

Definition 4. *We say $t \in T_M$ is* ***data dependent*** *on $t' \in T_M$ with respect to v, denoted as $t \xrightarrow{dd}_v t'$, iff there exists a variable $v \in mv(t) \cap cv(t')$ and there exists a path $[t_1 \cdots t_k](k \geq 1)$ such that $t_1 = t'$, $t_k = t$ and $t_j \in T_M \wedge v \notin cv(t_j)$ for all $1 < j < k$.*

Definition 5. *We say $n \in S_M$ (or $t \in T_M$) is* ***control dependent*** *on $n' \in S_M$, denoted as $n \xrightarrow{cd} n'$ (or $t \xrightarrow{cd} n'$), iff n' has at least two outgoing transitions, t_k and t_l,*

1. *all maximal paths from t_k eventually reach n (or t) and do so before (possibly) reaching n';*

2. *there exists a maximal path from t_l on which either (1) n (or t) is not reached, or (2) n (or t) is reached but only after reaching n'.*

Transitivity on hierarchy dependencies is considered: $n_1 \xRightarrow{hd} n_k$ iff there is a set of states $\{n_1, \ldots, n_k\}$ such that $n_i \xrightarrow{hd} n_{i+1}$ for $1 \leq i < k$. Transitivity on data and control dependencies is similarly defined.

5.2.4 Execution Step

Informally, a **snapshot** is an observable point in a model's execution [33]; it refers to the status of a model between execution steps.

Definition 6. *A snapshot of a model M is defined as $\bar{s}_M = (N_M, \sigma_M)$ where N_M is the state configuration in the execution of M and σ_M is the interpretation of variables at that particular point in the execution.*

When the context is clear, we will use (N, σ), or $\bar{s}$, without subscripts to denote the snapshot of an original FORML model.

A model starts executing from an initial snapshot, (N^I, σ^I). An execution step changes the snapshot of a model. Through an execution step, a model effectively exits a subset of

current states (related to the source states of the executing transitions) and enters the set of destination states of the executing transitions; meanwhile, the values of some variables may be changed.

Definition 7. *We write $M \vdash e : (N, \sigma) \Rightarrow (N', \sigma')$ when we refer to an execution step, e, that occurs in a model, M, such that its snapshot (N, σ) evolves to (N', σ') due to state transitions and changes to variable values.*

Each execution step involves a set of transitions which occur concurrently, denoted as:

$$e = \begin{matrix} t_1 \\ \vdots \\ t_k \end{matrix}$$

For any t_i, $1 \leq i \leq k$, we write $t_i \subset e$ to denote that t_i is one of the concurrent transitions in the execution step e. When the execution step e involves only one transition t (i.e., $k = 1$), we write $e = t$. This is a **non-concurrent execution step**.

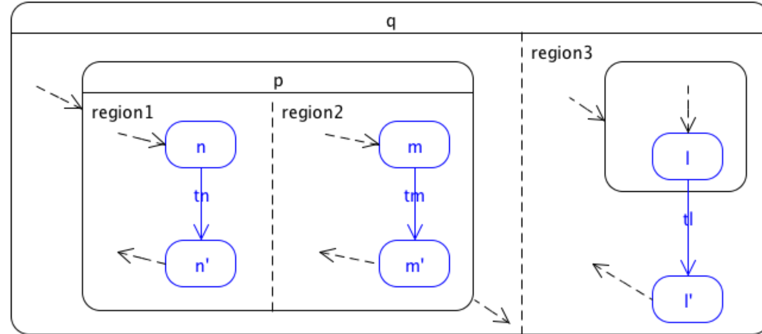


Figure 5.2: Concurrency in orthogonal regions as an execution step (only blue colored components are relevant in the execution)

Figure 5.2 illustrates an example of an execution step that consists of three concurrent transitions t_m, t_n and t_l. In general, we want to describe precisely what combination of transitions can occur concurrently in a single execution step:

Definition 8. *The set of concurrent transitions that are triggered simultaneously in an execution step e in $M \vdash e : (N, \sigma) \Rightarrow (N', \sigma')$ must satisfy:*

- *There are* k *transitions* $\{t_1, \ldots, t_k\} \subseteq T$ *that are triggered simultaneously, such that* $ss(t_j) \in N$ *and* $ds(t_j) \in N'$ *for all* $1 \leq j \leq k$*;*

- $ParentRegion(ss(t_i)) \parallel ParentRegion(ss(t_j))$*, or there exists another state* p *such that* $ss(t_i) \xRightarrow{hd} p$ *and* $ParentRegion(ss(t_j)) \parallel ParentRegion(p)$*, or there exists another state* p' *such that* $ss(t_j) \xRightarrow{hd} p'$ *and* $ParentRegion(p) \parallel ParentRegion(p')$*, for any* $1 \leq i \leq k \wedge 1 \leq j \leq k \wedge i \neq j$*;*

5.3 State Transition Rule

The state transition rule defines how the current state configuration N evolves to the next state configuration N' through a transition t. Intuitively, we know that the current state configuration exits the source state(s) of t and enters the destination state(s) [34]. We use two accessor functions for a transition t:

$exited(t)$**:** states exited when $ss(t)$ is exited, including $ss(t)$'s ancestors and descendants;

$entered(t)$**:** states entered when $ds(t)$ is entered, including $ds(t)$'s ancestors and relevant descendants' default initial states.

Now, we can define a state transition rule for one transition t as:

$$N' = (N - exited(t)) \cup entered(t)$$

where:

- $N \subseteq S_M$ and $N' \subseteq S_M$ are the current and next state configuration of the model.
- $exited(t)$ consists of the following:
  - The source state itself ($ss(t)$).
  - The ancestor states of $ss(t)$ up to the least common ancestor[4] with $ds(t)$. Let this set be $AncTill(ss(t), LCA(ss(t), ds(t)))$.
  - The descendant states of $ss(t)$ and any ancestor states of $ss(t)$ being exited. Let this set be $Desc(ss(t) \cup AncTill(ss(t), LCA(ss(t), ds(t))))$.

[4]The least common ancestor between two states is the state of highest rank that is an ancestor state of both states.

- $entered(t)$ is the set of all states including:
  - The destination state itself ($ds(t)$).
  - The ancestor states of $ds(t)$ to the least common ancestor with $ss(t)$, denoted as $AncTill(ds(t), LCA(ds(t), ss(t)))$.
  - The recursively identified default initial states of $ds(t)$ and its entered descendant states[5] ($InitDesc(ds(t))$).

Thus, a more detailed version of the state transition rule is:

$$\begin{aligned} N' =& (N - ss(t) - AncTill(ss(t), LCA(ss(t), ds(t))) \\ & - Desc(ss(t) \cup AncTill(ss(t), LCA(ss(t), ds(t))))) \\ & \cup ds(t) \cup AncTill(ds(t), LCA(ds(t), ss(t))) \cup InitDesc(ds(t)) \end{aligned}$$

The state transition rule can be generalized to an execution step e comprising multiple concurrent transitions:

$$N' = (N - \bigcup_{t \in e} exited(t)) \cup \bigcup_{t \in e} entered(t)$$

5.4 FormlSlicer's Multi-Stage Model Slicing Process

Section 4.4 describes in detail FormlSlicer's multi-stage model slicing algorithm. This section formalizes some terminology related to the process.

Slice Set This symbol $\mathcal{L}$ is used as a subscript to annotate a model component in the sliced model. The set of states in the sliced model is denoted as $S_{\mathcal{L}}$. The set of transitions in the sliced model is denoted as $T_{\mathcal{L}}$. A sliced model is denoted as $M_{\mathcal{L}}$. A state configuration in the sliced model is denoted as $N_{\mathcal{L}}$. An interpretation of variables in the sliced model is denoted as $\sigma_{\mathcal{L}}$. In addition, an original FOSM F becomes $F_{\mathcal{L}}$ after slicing.

[5] If $ds(t)$ is a composite state, then the default initial state within $ds(t)$ is entered; if the default initial state is also a composite state, then its default initial child state is also entered; and so on until an entered default initial state is a basic state.

5.4.1 Definitions

The concept of relevant variable is used during the initiation stage and the DD step in the general iterative slicing stage (Section 4.4.1 and Section 4.4.2.1). Here we define it formally:

Definition 9. *We define v to be a **relevant variable**, written $v \in Rv$, iff there exists $t \in T_{FOI}$ such that either $v \in mv(t)$ or there exists $t' \in T_{\mathcal{L}}$ such that $t \xrightarrow{dd} t'$ and $v \in mv(t')$.*

In other words, a variable is a relevant variable if it directly or indirectly controls the execution of the FOI.

The state connecting step in the model restructuring stage uses the concept of the next part-of-slice state that is reachable from a state n via a path that is missing the slice (Section 4.4.3.2). Since the state connecting step occurs after the state merging step, it treats any merged state as a part-of-slice state; thus, in Definition 10 we use ***n in $S_{\mathcal{L}}$*** to represent the situation where state n is in the sliced model or the situation where state n is represented by a merged state in the slice.

Definition 10. *The **next part-of-slice state set** of state n, denoted as $\widetilde{npos}(n)$, is the set of states such that:*

- *n' in $S_{\mathcal{L}}$;*
- *$\exists$ sequence of transitions $[t_1 \cdots t_k]$ such that $ss(t_1) = n \wedge ds(t_k) = n' \wedge (\forall i.1 \leq i \leq k.t_i \notin T_{\mathcal{L}} \wedge ParentState(ss(t_i)) = ParentState(ds(t_i))) \wedge (\forall j.1 \leq j < k.ds(t_j) \notin S_{\mathcal{L}})$.*

Note that k can be 1 in Definition 10; in this case, there is exactly one out-of-slice transition between the state n and its next-part-of-slice state n'.

5.4.2 Multi-Stage Model Slicing Process

We denote T_{ROS} to represent $\bigcup_{F \in ROS} T_F$ and S_{ROS} to represent $\bigcup_{F \in ROS} S_F$.

At each step of the slicing process, certain model elements from the original model are added to the sliced model.

1. Initiation Stage

(a) Variable Extraction Step

$$Rv = \bigcup_{t \in T_{FOI}} mv(t)$$

(b) Initial Transition Selection Step

$$T_{\mathcal{L}} = \{t \in T_{ROS} \mid \exists v \in Rv.v \in cv(t)\}$$
$$Rv' = Rv \cup \{mv(t) \mid t \in T_{\mathcal{L}}\}$$

2. General Iterative Slicing Stage

(a) DD Step

$$T'_{\mathcal{L}} = T_{\mathcal{L}} \cup \{t \in T_{ROS} \mid \exists t' \in T_{\mathcal{L}}.t' \neq t \wedge t' \xRightarrow{dd} t\}$$
$$Rv' = Rv \cup \{mv(t) \mid t \in T'_{\mathcal{L}}\}$$

(b) Cross-Hierarchy Transition Step

$$T'_{\mathcal{L}} = T_{\mathcal{L}} \cup \{ss(t) \xrightarrow{t_{true}} ds(t) \mid t \in T_{ROS} \wedge t \notin T_{\mathcal{L}} \wedge ParentState(ss(t)) \neq ParentState(ds(t))\}$$

(c) Transition-to-State Step

$$S_{\mathcal{L}} = \{n \in S_{ROS} \mid \exists t \in T_{\mathcal{L}}.n = ss(t) \vee n = ds(t)\}$$

(d) CD-HD Step

$$S'_{\mathcal{L}} = S_{\mathcal{L}} \cup \{n \in S_{ROS} \mid \exists n' \in S_{\mathcal{L}}.n' \neq n \wedge n' \xRightarrow{hd} \xRightarrow{cd} n\}$$

3. Model Restructuring Stage

(a) State Merging Step

i. Rule 1

Consider two states n, n'. Let $T_{n,n'}$ to be the set of all transitions t such that $ss(t) = n$ and $ds(t) = n'$, and $T_{n',n}$ to be the set of all transitions t such that $ss(t) = n'$ and $ds(t) = n$. Let $merged(n, n')$ be a merged state of n and n'.

$$\begin{aligned} S'_{\mathcal{L}} = &(S_{\mathcal{L}} - \{n, n' \mid n, n' \in S_{ROS} \wedge n \in S_{\mathcal{L}} \wedge T_{n,n'} \neq \emptyset \wedge T_{n',n} \neq \emptyset \\ &\wedge (\forall t \in T_{n,n'} \cup T_{n',n} \Rightarrow t \notin T_{\mathcal{L}})\}) \cup merged(n, n') \end{aligned}$$

ii. Rule 2
Consider two states n, n'. Let $T_{n,n'}$ to be the set of all transitions t such that $ss(t) = n$ and $ds(t) = n'$. Let T_n be the set of all transitions t such that $ss(t) = n$. Let $merged(n, n')$ be a merged state of n and n'.

$$S'_{\mathcal{L}} = (S_{\mathcal{L}} - \{n, n' \mid n, n' \in S_{ROS} \wedge (n \in S_{\mathcal{L}} \vee n' \in S_{\mathcal{L}}) \wedge T_{n,n'} \cap T_{\mathcal{L}} = \emptyset \\ \wedge T_{n,n'} = T_n\}) \cup merged(n, n')$$

(b) State Connecting Step

$$T'_{\mathcal{L}} = T_{\mathcal{L}} \cup \{n \xrightarrow{t_{true}} n' \mid n \in S_{ROS} \wedge n \; in \; S_{\mathcal{L}} \wedge n' \in \widetilde{npos}(n)\}$$

5.5 Proof

We perform an inductive proof: we show that the original model's initial snapshot is projected to the sliced model's initial snapshot; and that assuming after any prefix execution the original model's snapshot can be projected onto the sliced model's snapshot, then the sliced model can simulate the next step in the original model, such that the original model's resulting snapshot can be projected onto the sliced model's resulting snapshot.

Ideally, we want to formally prove the simulation relation between the original model and the sliced model based on the semantics of the model. As discussed in Section 2.3.1, Milner [26] proposed the technique to prove simulation between two programs; this technique requires a precise definition of the program which enables simulation properties to be stated and proved succinctly. However, we are unable to provide a precise definition of the semantics of our model due to its complexity. This results in a gap between what we have proved (Theorem 1) and what we ideally want to prove (i.e., a semantically simulation relation between M and $M_{\mathcal{L}}$); this will be explained in Subsection 5.5.4.

Our correctness claim is that each step in a given execution trace in the original model can be projected to a step in some execution trace in the sliced model. To prove this, we need to:

1. Define projection of snapshots (Section 5.5.1);
2. Prove that if after any prefix execution the original model's snapshot can be projected onto the sliced model's snapshot, then the sliced model can simulate the next step in the original model, such that the original model's resulting snapshot can be projected onto the sliced model's resulting snapshot;

(a) First show this holds for individual transitions (Section 5.5.2) and then show it holds for a collection of transitions in an execution step (Section 5.5.3).

3. Prove that any initial snapshot in the original model can be projected onto a corresponding initial snapshot in the sliced model, so that together with the proof of claim 2, prove the correctness claim (Section 5.5.4).

5.5.1 Projection of Snapshot in the Original Model to Snapshot in the Sliced Model

We want to define the projection of the original model's snapshot onto the sliced model's snapshot. From Definition 6 we know that a snapshot consists of a state configuration and an interpretation of variables, therefore the projection of snapshot include two notions:

1. The state configuration of the original model must map to a corresponding state configuration in the sliced model;
2. the value of any relevant variable is the same in both the original model and the sliced model.

For the first notion, we note that the sliced model is not likely to include all of the states from the original model, thus "corresponding state configurations" cannot mean equivalent state configurations. Instead, we write that the state configuration N of the original model corresponds to the state configuration $N_{\mathcal{L}}$ in sliced model if the two state configurations agree with respect to states that are in the slice:

$$N \cap S_{\mathcal{L}} \subseteq N_{\mathcal{L}}$$

For the second notion, we only consider the interpretations of system-controlled variables. Because the environment-controlled variables are influenced by the external environment (in both the original and the sliced models), any change that the environment makes to the environment-controlled variables in the original model is equivalent to the environment-controlled variables in the sliced model. Thus we do not consider them any further.

Definition 11. *We define that a snapshot in the original model $\bar{s} = (N, \sigma)$* ***is projected to*** *another snapshot in the sliced model, $\bar{s}_{\mathcal{L}} = (N_{\mathcal{L}}, \sigma_{\mathcal{L}})$, when*

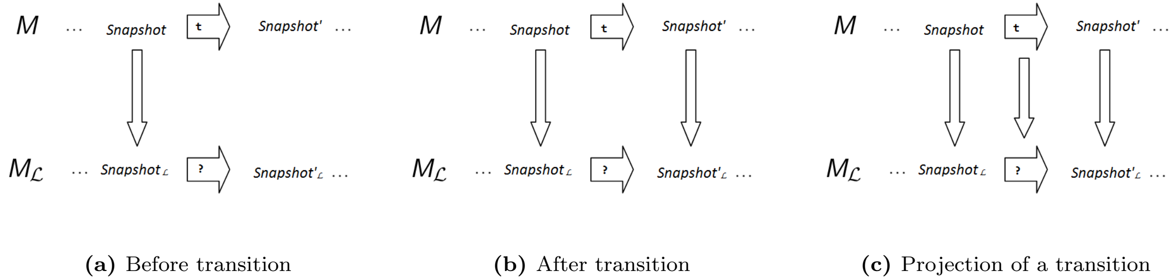
- $N \cap S_{\mathcal{L}} \subseteq N_{\mathcal{L}}$;
- $\forall v \in Rv,\ \sigma(v) = \sigma_{\mathcal{L}}(v)$.

We write it as $P((N_{\mathcal{L}}, \sigma_{\mathcal{L}})) = (N, \sigma)$ *or* $P(\bar{s}) = \bar{s}_{\mathcal{L}}$.

5.5.2 Projection of One Transition in the Original Model to Epsilon or One Transition in the Sliced Model

Recall from Section 5.2 that a transition is a progression in a model's execution; in contrast, epsilon in a model is a non-progression in a model's execution (i.e., no transitions executes and the model stays in its snapshot).

We want to prove that one transition t in the original model can be projected to one transition $t_{\mathcal{L}}$ or epsilon in the sliced model (Figure 5.3c), such that if the snapshot in the original model is projected to the snapshot in the sliced model before the transition (Figure 5.3a), then the snapshot in the original model is still projected to the snapshot in the sliced model after the transition in the original model (Figure 5.3b).



(a) Before transition **(b)** After transition **(c)** Projection of a transition

Figure 5.3: Projection of a transition from the original model to the sliced model

This can be proved by case-based analysis. Before this, we want to explain that a state that is represented as a merged state in the sliced model is not different from any part-of-slice state that is not merged with others. To explain this, we first prove that a merged state in the sliced model has the same properties as either of the constituent states in the original model. This is expressed in Lemma 1.

Lemma 1. *Consider a merged state* n_{merged} *in the sliced model. Let* n, n' *be two constituent states of* n_{merged}*. Then* $n_{merged} = (n \vee n')$.

Proof. During the state merging step, any two states n and n' that satisfy either of the state merging rules are deleted from the sliced model and a new merged state n_{merged} is added to the sliced model. According to the state merging step that we describe in Section 4.4.3.1, n_{merged} in the sliced model has the same properties as n in the original model:

- Each of all n's outgoing transitions exits from n_{merged};
  - Thus, whenever n has an out-of-slice outgoing path that reaches another state x that is in the slice, $x \in \widetilde{npos}(n_{merged})$.
- Each of all n's incoming transitions enters n_{merged};
  - Thus, whenever n has an out-of-slice incoming path from another state y, $n_{merged} \in \widetilde{npos}(y)$.
- Both n_{merged} and n have the same rank of state hierarchy;
  - Thus, each of all n's descendant states is also n_{merged}'s descendant state if it is part-of-slice,
  - Each of all n's ancestor states is also n_{merged}'s ancestor state.
- If n is a default initial state, n_{merged} is also a default initial state of the same sub-machine.

In conclusion, $n_{merged} \in S_{\mathcal{L}}$ preserves all properties of n. The proof for state n' is symmetric. ■

For a state n that is merged with another state during the state merging step, we cannot simply denote $n \in S_{\mathcal{L}}$ because the original n is not in the sliced model. We cannot simply denote $n \notin S_{\mathcal{L}}$ either; as what Lemma 1 shows, there exists another state $n_{merged} \in S_{\mathcal{L}}$ which preserves all properties of n and n's behavior is not different from any other part-of-slice state. Based on this, we can merge the case when n is in the sliced model and the case when n is not in the sliced model but is represented as a merged state in the sliced model; we denote $n\ in\ S_{\mathcal{L}}$ to represent either of the two cases.

We are now ready to show the case-based analysis. There are seven different types of transition shown in the decision tree in Figure 5.4. For brevity, we denote $SP(t)$ to represent the condition of $ParentState(ss(t)) = ParentState(ds(t))$ and denote $merged(t)$ to represent the condition when $\exists n_{merged}.(n_{merged} = ss(t) \vee ds(t))$.

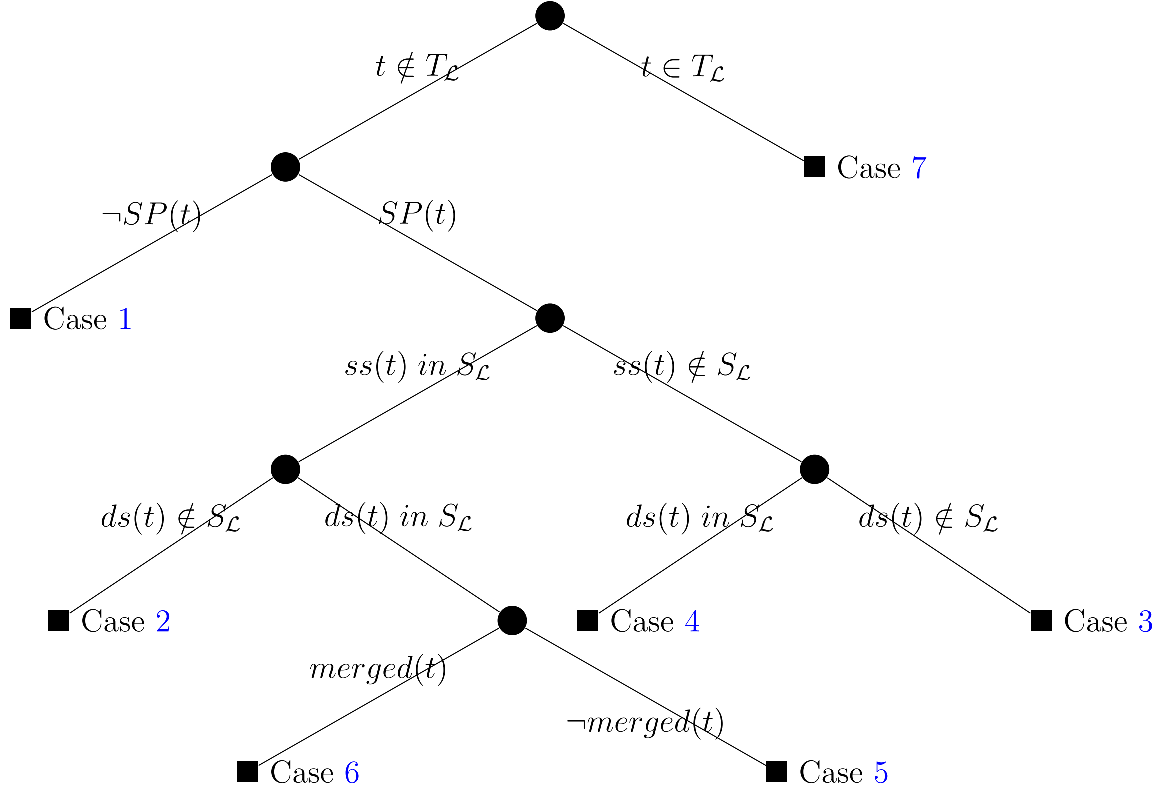


Figure 5.4: The decision tree for case-based analysis in projection of transitions

Lemma 2. *Consider a transition t in the original model M, $M \vdash t : (N, \sigma) \Rightarrow (N', \sigma')$. The projection function of $P(t)$ is as follows:*

$$P(t) = \begin{cases} t_{true} & if \;\; t \notin T_{\mathcal{L}} \wedge \neg SP(t) & (1) \\ \epsilon & if \;\; ss(t) \; in \; S_{\mathcal{L}} \wedge ds(t) \notin S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SP(t) & (2) \\ \epsilon & if \;\; ss(t) \notin S_{\mathcal{L}} \wedge ds(t) \notin S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SP(t) & (3) \\ t_{true} & if \;\; ss(t) \notin S_{\mathcal{L}} \wedge ds(t) \; in \; S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SP(t) & (4) \\ t_{true} & if \;\; ss(t) \; in \; S_{\mathcal{L}} \wedge ds(t) \; in \; S_{\mathcal{L}} \wedge \neg merged(t) \wedge t \notin T_{\mathcal{L}} \wedge SP(t) & (5) \\ \epsilon & if \;\; ss(t) \; in \; S_{\mathcal{L}} \wedge ds(t) \; in \; S_{\mathcal{L}} \wedge merged(t) \wedge t \notin T_{\mathcal{L}} \wedge SP(t) & (6) \\ t & if \;\; t \in T_{\mathcal{L}} & (7) \end{cases}$$

$$M_{\mathcal{L}} \vdash t_{\mathcal{L}} : (N_{\mathcal{L}}, \sigma_{\mathcal{L}}) \Rightarrow \begin{cases} (N_{\mathcal{L}}, \sigma_{\mathcal{L}}) & \text{for the cases of 2, 3 and 6} \\ (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}}) & \text{for the cases of 1, 4, 5 and 7} \end{cases}$$

Given that $P((N,\sigma)) = (N_{\mathcal{L}}, \sigma_{\mathcal{L}})$. *In the former case,* $P((N',\sigma')) = (N_{\mathcal{L}}, \sigma_{\mathcal{L}})$. *In the latter case,* $P((N',\sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$.

Proof.

Given Conditions At the start of the execution step, we know that $P((N,\sigma)) = (N_{\mathcal{L}}, \sigma_{\mathcal{L}})$. Thus,

$$N \cap S_{\mathcal{L}} \subseteq N_{\mathcal{L}} \tag{8}$$

$$\forall v \in Rv.\sigma_{\mathcal{L}}(v) = \sigma(v) \tag{9}$$

The Case of Cross-Hierarchy Transition Case 1

When $(t \notin T_{\mathcal{L}}) \wedge (\neg SP(t))$, the cross-hierarchy transition step in the general iterative slicing stage will replace t with a "true" transition $t_{true} \in T_{\mathcal{L}}$. Then because of the transition-to-state step, we obtain:

$$ss(t), ds(t) \ in \ S_{\mathcal{L}} \tag{10}$$

Because of (10) and the CD-HD step in the general iterative slicing stage:

$$AncTill(ss(t), LCA(ss(t), ds(t))) \subset S_{\mathcal{L}} \tag{11}$$

$$AncTill(ds(t), LCA(ds(t), ss(t))) \subset S_{\mathcal{L}} \tag{12}$$

We can also observe that:

$$Desc(ss(t) \cup AncTill(ss(t), LCA(ss(t), ds(t)))) \cap S_{\mathcal{L}} = Desc_{\mathcal{L}}(ss(t)) \tag{13}$$

$$InitDesc(ds(t)) \cap S_{\mathcal{L}} = InitDesc_{\mathcal{L}}(ds(t)) \tag{14}$$

Because of condition (10), (11), and (13), we deduce that through t, each state that is exited in the original model is either exited in the sliced model if it is part-of-slice, or is out-of-slice (and therefore does not participate in the state transition in the sliced model).

Because of condition (10), (12), and (14), we deduce that through t, each state that is entered in the original model is either entered in the sliced model if it is part-of-slice, or is out-of-slice.

Based on these deductions, the subset relation between the sliced and the original models' state configurations, shown at (8), is maintained:

$$N' \cap S_{\mathcal{L}} \subseteq N'_{\mathcal{L}}. \tag{15}$$

Next, we want to prove that $\forall v \in Rv.\sigma'(v) = \sigma'_{\mathcal{L}}(v)$. To do so, we first prove by contradiction that $\sigma(v) = \sigma'(v), \forall v \in Rv$. Assume that there is a variable $v \in Rv$ whose value is changed by t, so that $\sigma(v) \neq \sigma'(v)$. Then $v \in cv(t)$. Based on Definition 9 of relevant variables, there must be another transition t_x such that $v \in mv(t_x)$ and t_x is either a transition in the FOI or the sliced ROS. Based on Definition 4 on data dependence, $t_x \xrightarrow{dd}_v t$. According to the DD step in the general iterative slicing stage, t must be in the sliced model; this contradicts the Case 1 condition that $t \notin T_{\mathcal{L}}$. Therefore:

$$\forall v \in Rv.\sigma(v) = \sigma'(v) \tag{16}$$

Because t_{true} does not change the values of any variables in the sliced model,

$$\sigma_{\mathcal{L}} = \sigma'_{\mathcal{L}} \tag{17}$$

Given condition (9), (16) and (17), we get

$$\forall v \in Rv.\sigma'_{\mathcal{L}}(v) = \sigma_{\mathcal{L}}(v) = \sigma(v) = \sigma'(v) \tag{18}$$

Given both (15) and (18), we deduce that $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$.

The Case of Having Only Source State in the Slice   Case 2

When $ss(t)\ in\ S_{\mathcal{L}} \wedge ds(t) \notin S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SP(t)$, the state configuration in the sliced model does not change whilst the state configuration in original model changes from N to N' through transition t. Because of $SP(t)$, we deduce that:

$$AncTill(ss(t), LCA(ss(t), ds(t))) = AncTill(ds(t), LCA(ds(t), ss(t))) = \emptyset \tag{19}$$

Because of $ds(t) \notin S_{\mathcal{L}}$ and the CD-HD step, we deduce that:

$$InitDesc(ds(t)) \cap S_{\mathcal{L}} = \emptyset \tag{20}$$

Because of (19) and (20), we get:

$$entered(t) \cap S_{\mathcal{L}} = \emptyset \tag{21}$$

States that are exited are $ss(t)$ and its descendants. Since the state configuration in the sliced model remains unchanged, the states that are exited in the original model do

not affect the subset relation in (8) (because a subset of $N_{\mathcal{L}}$ remains as a subset after some elements are deleted from it). Altogether, we deduce that:

$$N' \cap S_{\mathcal{L}} \subseteq N_{\mathcal{L}}. \tag{22}$$

Next, we want to prove that $\forall v \in Rv.\sigma'(v) = \sigma_{\mathcal{L}}(v)$. To do so, we first prove by contradiction that $\sigma(v) = \sigma'(v), \forall v \in Rv$. This proof will be exactly the same as that in Case 1 because of the DD step. Given condition (9) and condition (16), we can deduce that

$$\forall v \in Rv.\sigma_{\mathcal{L}}(v) = \sigma'(v) \tag{23}$$

Given both (22) and (23), we can deduce that $P((N', \sigma')) = (N_{\mathcal{L}}, \sigma_{\mathcal{L}})$.

The Case of Having Nothing in the Slice Case 3

When $ss(t) \notin S_{\mathcal{L}} \wedge ds(t) \notin S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SP(t)$, the state configuration in the sliced model does not change whereas the state configuration in the original model changes from N to N' through transition t. This only difference between this case and Case (2) is that $ss(t)$ is not in the sliced model. In Case 2, we deduce that $entered(t) \cap S_{\mathcal{L}} = \emptyset$ because $AncTill(ds(t), LCA(ds(t), ss(t))) = \emptyset$ and $InitDesc(ds(t)) \cap S_{\mathcal{L}} = \emptyset$; we also explained that states that are exited (which are $ss(t)$ and its related ancestors and descendants) do not matter. Therefore, the proof about state configuration is exactly the same as Case 2. Thus:

$$N' \cap S_{\mathcal{L}} \subseteq N_{\mathcal{L}}. \tag{24}$$

Next, we want to prove that $\forall v \in Rv.\sigma'(v) = \sigma_{\mathcal{L}}(v)$. To do so, we first prove $\forall v \in Rv.\sigma(v) = \sigma'(v)$; this proof is exactly the same as Case 1 because of the DD step. Given condition (9) and condition (16), we deduce that

$$\forall v \in Rv.\sigma_{\mathcal{L}}(v) = \sigma'(v) \tag{25}$$

Given both (24) and (25), we can deduce that $P((N', \sigma')) = (N_{\mathcal{L}}, \sigma_{\mathcal{L}})$.

The Case of Having Only Destination State in the Slice Case 4

When $ss(t) \notin S_{\mathcal{L}} \wedge ds(t)\ in\ S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SP(t)$, we know that there must be another state $n\ in\ S_{\mathcal{L}}$ such that $ds(t) \in \widetilde{npos}(n)$. The state connecting step in the model restructuring stage creates a "true" transition $n \xrightarrow{t_{true}} ds(t)$. Because all the states along the path

$[n_1 \cdots n_k]$ ($n_1 = n$, $n_k = ds(t)$) are at the same rank of state hierarchy, we know that from the analysis of Case 2 that the subset relation between the original model's state configuration and the sliced model's state configuration (8) holds in state n_2 (after the first transition along the path $[n1 \cdots n_k]$); and we know from our analysis of Case 3 that the subset relation of the state configurations between the original model and the sliced model holds for any state in $\{n_3, \ldots, n_{k-1}\}$. Therefore, condition (8) holds in this case.

Because the states are all at the same rank of state hierarchy, we also know that:

$$\forall (i, j \in \{1, \ldots, k\}).(i \neq j) \Rightarrow AncTill(n, LCA(n, ds(t))) = AncTill(n_i, LCA(n_i, n_j)) = \emptyset \quad (26)$$

Also, because $ds(t)\ in\ S_{\mathcal{L}}$, we get:

$$InitDesc(ds(t)) \cap S_{\mathcal{L}} = InitDesc_{\mathcal{L}}(ds(t)) \quad (27)$$

Through the "true" transition, the state configuration in the sliced model changes such that $ds(t)$ and $InitDesc_{\mathcal{L}}(ds(t))$ are entered and n and $Desc(n)$ are exited. If we add this knowledge to conditions (26), (27), and (8), we deduce that:

$$N' \cap S_{\mathcal{L}} \subseteq N'_{\mathcal{L}}. \quad (28)$$

Next, we want to prove that $\forall v \in Rv.\sigma'(v) = \sigma'_{\mathcal{L}}(v)$. To do so, we first prove $\sigma(v) = \sigma'(v), \forall v \in Rv$; this proof is exactly same as that in Case 1 because of the DD step. Together with condition (9), we deduce that

$$\forall v \in Rv.\sigma'_{\mathcal{L}}(v) = \sigma'(v) \quad (29)$$

Given both (28) and (29), we deduce that $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$.

The Case of Having Source and Destination States in the Slice Case 5

When $ss(t)\ in\ S_{\mathcal{L}} \wedge ds(t)\ in\ S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SP(t)$, the state connecting step in the model restructuring stage adds a "true" transition t_{true} to the sliced model connecting $ss(t)$ and $ds(t)$. Therefore, the state configuration in the sliced model will change via t_{true} in the same manner that the state configuration in the original model changes via t. Given condition (8), we get:

$$N' \cap S_{\mathcal{L}} \subseteq N'_{\mathcal{L}}. \quad (30)$$

Next, we want to prove that $\forall v \in Rv.\sigma'(v) = \sigma'_{\mathcal{L}}(v)$. To do so, we first prove that $\sigma(v) = \sigma'(v), \forall v \in Rv$; this proof is exactly the same as that in Case 1 because of the DD step. Together with condition (9), we deduce that

$$\forall v \in Rv.\sigma'_{\mathcal{L}}(v) = \sigma'(v) \tag{31}$$

Given both (30) and (31), we deduce that $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$.

The Case of Having a Merged State in the Slice   Case 6

In this case, the sliced model contains a merged state n_{merged} that represents both $ss(t)$ and $ds(t)$. Thus, $n_{merged} = (ss(t) \vee ds(t))\ in\ S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SP(t)$. We know that:

$$n_{merged} = (ss(t) \vee ds(t)) \tag{32}$$

$$n_{merged} \in N_{\mathcal{L}} \tag{33}$$

$$\text{Because (32) and (33), } ds(t) = n_{merged} \in N_{\mathcal{L}} \tag{34}$$

Because of $SP(t)$,

$$AncTill(ds(t), LCA(ds(t), ss(t))) = AncTill(ss(t), LCA(ss(t), ds(t))) = \emptyset \tag{35}$$

Because during the state merging step in the model restructuring stage, the merged state contains all child regions of the original states if they are composite states. Therefore:

$$InitDesc(ds(t)) \subseteq N_{\mathcal{L}} \tag{36}$$

Since the state configuration in the sliced model remains unchanged, the states that are exited in the original model do not affect the subset relation in (8) (because a subset of $N_{\mathcal{L}}$ remains as a subset after some elements are deleted from it).

Because of (34) and (36), the states that are entered in the original model's state configuration are pre-existing in the sliced model's state configuration. Therefore:

$$N' \cap S_{\mathcal{L}} \subseteq N_{\mathcal{L}} \tag{37}$$

Next, we want to prove that $\forall v \in Rv.\sigma'(v) = \sigma'_{\mathcal{L}}(v)$. When $t \notin T_{\mathcal{L}}$, the proof is the same as Case 2, Case 3, Case 4, and Case 5. Therefore:

$$\forall v \in Rv.\sigma'(v) = \sigma'_{\mathcal{L}}(v) \tag{38}$$

Given both (37) and (38), we can deduce that $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$.

The Case when the Transition is Part-of-Slice Case 7

When $t \in T_{\mathcal{L}}$, according to the transition-to-state step in the general iterative slicing stage, $ss(t)\ in\ S_{\mathcal{L}}, ds(t)\ in\ S_{\mathcal{L}}$. The state configuration in the sliced model changes in the same manner as the state configuration in the original model. Because $ss(t)$ and $ds(t)$ are part-of-slice, due to CD-HD step we know that $AncTill(ds(t), LCA(ds(t), ss(t)))$ and $AncTill(ss(t), LCA(ss(t), ds(t)))$ are also part-of-slice; we also know that $Desc(ss(t)) \cap S_{\mathcal{L}} = Desc_{\mathcal{L}}(ss(t))$ and $InitDesc(ds(t)) \cap S_{\mathcal{L}} = InitDesc_{\mathcal{L}}(ds(t))$. In other words, each state that is exited in the original model is either exited in the sliced model if it is part-of-slice, or is out-of-slice (and therefore does not participate in the state transition in the sliced model); and each state that is entered in the original model is either entered in the sliced model if it is part-of-slice, or is out-of-slice.

Because $t \in T_{\mathcal{L}}$, the value of any relevant variable in the sliced model will change in the same way as the relevant variables in the original model. Therefore, $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$.

■

5.5.3 Projection of One Execution Step in the Original Model to One Execution Step in the Sliced Model

We want to prove that if the original model's snapshot is projected onto the the sliced model's snapshot and the original model performs an execution step e whereas the sliced model executes the corresponding projected step $P(e)$, then the resulting snapshot in the original model can be projected onto the resulting snapshot of the sliced model.

Lemma 3. *Consider an execution step in the original model* $M \vdash e : \bar{s} \Rightarrow \bar{s}'$ *and its projected execution step in the sliced model* $M_{\mathcal{L}} \vdash P(e) : \bar{s}_{\mathcal{L}} \Rightarrow \bar{s}'_{\mathcal{L}}$. *If* $P(\bar{s}) = \bar{s}_{\mathcal{L}}$*, then* $P(\bar{s}') = \bar{s}'_{\mathcal{L}}$.

Proof. As mentioned in Section 5.2.4, each execution step involves a set of transitions which occur concurrently, denoted as:

$$e = \begin{array}{c} t_1 \\ \vdots \\ t_k \end{array}$$

In Lemma 2, we have proved that for all types of transitions t in the original model, there exists a transition $P(t) = (\epsilon \vee t_{\mathcal{L}})$ in the sliced model such that if the original model's

snapshot before executing t is projected to the sliced model's snapshot before executing $P(t)$, then the original model's resulting snapshot after t is projected to the sliced model's resulting snapshot after $P(t)$.

According to Definition 8, each concurrent transition of an execution step e occurs in a distinct orthogonal region; they do not interfere with each others' state configurations. In addition, the concurrent transitions' actions in the sliced model either do not interfere with each other (i.e., making changes to different relevant variables), or interfere with each other in the same way as that in the original model (i.e., making changes to the same relevant variable). If two transitions are making changes to the same variable and one is part-of-slice while the other one is not, that variable is definitely not a relevant variable because of DD step. Based on these, we conclude that the projection of snapshots of individual transitions in e can be composed together to show the projection of snapshots of e:

$$P(e) = P\begin{pmatrix} t_1 \\ \vdots \\ t_k \end{pmatrix} = \begin{pmatrix} P(t_1) \\ \vdots \\ P(t_k) \end{pmatrix} = e_{\mathcal{L}} \tag{39}$$

such that $P(t_j) = (\epsilon \vee t_{j\mathcal{L}})$ for all $j.1 \leq j \leq k$.

In other words, an execution step e in the original model ($M \vdash e : \bar{s} \Rightarrow \bar{s}'$) has a projected execution step $e_{\mathcal{L}}$ in the sliced model ($M_{\mathcal{L}} \vdash e_{\mathcal{L}} : \bar{s}_{\mathcal{L}} \Rightarrow \bar{s}'_{\mathcal{L}}$) such that if $P(\bar{s}) = \bar{s}_{\mathcal{L}}$, then $P(\bar{s}') = \bar{s}'_{\mathcal{L}}$.

■

5.5.4 Simulation

Theorem 1. *Consider an execution trace in the original model as an infinite sequence* $\{e_i \mid i > 0\}$ *and some simulating execution trace in the sliced model as an infinite sequence* $\{e_{i\mathcal{L}} \mid i > 0\}$*. Also,* $M \vdash e_i : \bar{s}_{i-1} \Rightarrow \bar{s}_i$ *and* $M_{\mathcal{L}} \vdash e_{i\mathcal{L}} : \bar{s}_{(i-1)\mathcal{L}} \Rightarrow \bar{s}_{i\mathcal{L}}$ *for all* $i > 0$*. Then* $P(\bar{s}_i) = \bar{s}_{i\mathcal{L}}$ *for all* $i \geq 0$*.*

Proof. This is a proof by induction.

Base Case: The initial snapshot of the original model is $\bar{s}_0 = (N^I, \sigma^I)$ where N^I is the set comprising the root state and all the default initial states of relevant descendant states of the root state, which are identified recursively; and σ^I is an interpretation

of all variables to their respective default or uninitialized values. Similarly, the initial snapshot of the sliced model is $\bar{s}_{0\mathcal{L}} = (N^I_{\mathcal{L}}, \sigma^I_{\mathcal{L}})$.

Intuitively, we know that the initial state configuration in the sliced model $N^I_{\mathcal{L}}$ will be:

$$N^I_{\mathcal{L}} = (N^I \cap S_{\mathcal{L}}) \cup N^{newI}$$

where N^{newI} is the new default initial states in the sliced model[6]. Therefore, $N^I \cap S_{\mathcal{L}} \subseteq N^I_{\mathcal{L}}$.

In addition, we know that the values of all relevant variables in the sliced model are at their default or uninitialized values, because no transitions have executed yet, thus no transitions' actions have been performed. Therefore, $\forall v \in Rv . \sigma^I(v) = \sigma^I_{\mathcal{L}}(v)$

Because $N^I \cap S_{\mathcal{L}} \subseteq N^I_{\mathcal{L}}$ and $\sigma^I(v) = \sigma^I_{\mathcal{L}}(v), \forall v \in Rv$, we can conclude that $P(\bar{s}_0) = \bar{s}_{0\mathcal{L}}$.

Inductive Case: In Lemma 3, we prove that for an execution step in the original model $M \vdash e : \bar{s} \Rightarrow \bar{s}'$ and its projected execution step in the sliced model $M_{\mathcal{L}} \vdash P(e) : \bar{s}_{\mathcal{L}} \Rightarrow \bar{s}'_{\mathcal{L}}$, if $P(\bar{s}) = \bar{s}_{\mathcal{L}}$, then $P(\bar{s}') = \bar{s}'_{\mathcal{L}}$. Based on Lemma 3, $P(\bar{s}_0) = \bar{s}_{0\mathcal{L}}$ leads to $P(\bar{s}_1) = \bar{s}_{1\mathcal{L}}$; then it leads to $P(\bar{s}_2) = \bar{s}_{2\mathcal{L}}$ and so on.

Together with the base case and the inductive case, we know that the original model's snapshot is always projected to the sliced model's snapshot for the whole execution trace in the original model. ■

5.5.4.1 A Gap between the Theorem and the Simulation Relation

As discussed in Section 5.2.2 and the beginning of Section 5.5, the fact that we miss out a precise semantic definition of our model makes it impossible to formally prove the simulation relation between models. Thus, in Theorem 1 we only claim what we have proved using definition of projection of snapshots. We do not claim that this leads to the conclusion that the sliced model semantically simulates the original model.

In Section 2.3.1, we explain how Milner defines simulation of two programs [26]. Here we want to formalize the definition of simulation between the original model and the sliced model in a similar way.

[6] Recall from the sub-step 1 of the state connecting step in Section 4.4.3.2 that sometimes when the original default initial state is not in the sliced model, FormlSlicer searches for the next part-of-slice state to be the new default initial state.

Definition 12. *Let $R \subseteq D \times D_{\mathcal{L}}$ where D is the domain of all possible snapshots in M and $D_{\mathcal{L}}$ is the domain of all possible snapshots in $M_{\mathcal{L}}$. Let $Next()$ be a next-step function that takes in current snapshot of M and outputs the next snapshot through an execution step; similarly, $Next_{\mathcal{L}}()$ is a next-step function for $M_{\mathcal{L}}$. Then we say that R* ***is a simulation of*** *M* ***by*** *$M_{\mathcal{L}}$ if $\forall\ \bar{s}, \bar{s}_{\mathcal{L}}$, $(\bar{s}, \bar{s}_{\mathcal{L}}) \in R \Rightarrow (Next(\bar{s}), Next_{\mathcal{L}}(\bar{s}_{\mathcal{L}})) \in R$.*

Figure 5.5 is a diagram that illustrates Definition 12.

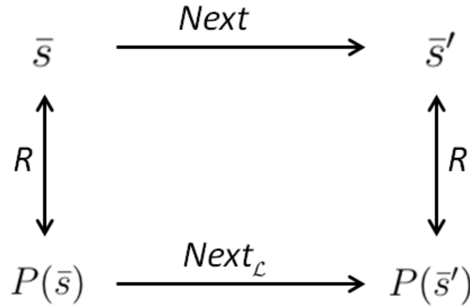


Figure 5.5: The simulation relation between the execution traces of M and $M_{\mathcal{L}}$

In order to prove that $M_{\mathcal{L}}$ and M satisfy the simulation relation in Definition 12, we need to formalize the next-step function $Next()$ semantically. Intuitively, the state transition rule which we introduce in Section 5.3 is part of the formalism of $Next()$; but it is not sufficient, because the state transition rule simply states part of the model's behavior syntactically. We need the semantics of the model so that we can precisely formalize its behavior. The semantics of the model is usually represented as a tuple that consists of domains of model elements, relations between model elements, functions of the behavior of the model that involve the model elements, and possibly more. However, we note that formally defining the semantics of our model is not a trivial task and proving the simulation between models using the formal semantics is even more complex. We will leave these to future work.

Chapter 6

Empirical Evaluations of FormlSlicer

This chapter demonstrates that FormlSlicer significantly reduces the size of the original model. FormlSlicer is a tool implemented in Java for the workflow described in Chapter 4. At the time of writing, it has 2743 lines of code.

6.1 Choosing a Model for Empirical Evaluation

In order to evaluate how well FormlSlicer reduces model sizes, we must use a model that represents requirements of a feature-rich software system in a real-world domain; otherwise, it is not convincing to demonstrate the reduction effects of FormlSlicer.

It is not a trivial task to create such a model in FORML with sufficient content. In Shaker's thesis (the original work in FORML [28]), there are only two FORML models—the telephony case study and the automotive case study. The telephony case study involves communications among different products; that means that a telephone's behavior is dependent on another telephone's behavior. FormlSlicer does not consider communications among different products, and therefore cannot perform slicing on such a model. On the other hand, the FORML model in the automotive case study, called *Autosoft*, does not involve communications among different vehicles; this makes it a better target for our empirical evaluation.

The *Autosoft* model consists of many feature modules. Figure 6.1 shows the feature model of *Autosoft*. A feature model is a tree that depicts the constraints among features: a feature is dependent on another feature if the former is a child node of the latter [35]. In FORML, the feature modules that are dependent on others are expressed in terms of

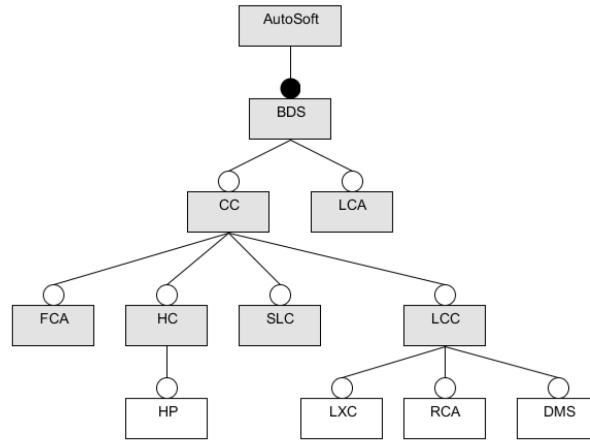


Figure 6.1: A feature model that constraints the relationships between features in *Autosoft*

state-machine fragments. Because FormlSlicer cannot slice on state-machine fragments, we need to compose the feature modules to form complete state machines, called FOSMs[1]. In this empirical study, an FOSM is composed of at most three feature modules, which are colored in grey in Figure 6.1; the feature modules as leave nodes (e.g., HP) in the feature model are small state-machine fragments which enhance the conditions of a few transitions in the base feature modules, and thus including them in the composition does not make much difference to our statistics and conclusion. As a result, we obtain five FOSMs from *Autosoft*:

- Adaptive cruise control (ACC) (Figure A.2), which is composed from features BDS, CC and headway control (HC);
- Forward collision alert (FCA) (Figure A.3), which is composed from feature BDS, CC and FCA;
- Lane change alert (LCA) (Figure A.4), which is composed from feature BDS and LCA;
- Lane centring control (LCC) (Figure A.5), which is composed from feature BDS, CC and LCC;
- Speed limit control (SLC) (Figure A.6), which is composed from feature BDS, CC and SLC.

[1] See Section 3.3 for the description about feature-oriented state machine (FOSM).

Because all of them are based on BDS and four of them are based on CC, they contain artificial redundancies. In fact, all features from *Autosoft* are related to a vehicle's motion. We find that they are a bit repetitive and want to add some variety to the model. In light of this, we create two additional FOSMs that are related to a vehicle's internal air conditions:

- Air quality system (AQS) (Figure A.7);
- Air conditioning (AC) (Figure A.8).

These two features are derived from Dietrich's mode-based state machines that model the behavioral requirements of production-grade automotive features [36]. However, although both Dietrich's state machines and Shaker's feature modules are modeling the behaviors of features in an automotive system, they have very different styles. Thus, we need to modify Dietrich's state machines before using them in our empirical study. These include rewriting some transition labels in Dietrich's state machines to conform to the FORML syntax (e.g., rewriting "*AC_DISENGAGE AND Compressor.pressureGradient* > *25.5*" to "*AC_DISENGAGE+(o) [Compressor.pressureGradient* > *25.5]/*"), removing undefined functions (e.g., *performDiagnostics()* or *sensorBroken()*), changing in-state actions to become part of the labels of self-looping, incoming or outgoing transitions (e.g., removing the texts "*entry/waitTime:=0;*" written within the *Wait_Exit* state in AQS and adding an action "*a1:waitTime:=0;*" to *Wait_Exit*'s incoming transition), matching the naming of similar variables (e.g., changing the variable "*Vehicle.speed*" to "*car.speed*" to become consistent with the other five features), and so on.

Altogether, these seven FOSMs form the input model to FormlSlicer, as shown in Figure A.1.

6.2 Reduction of Model Size

In Section 1.1.2, we mentioned that a useful sliced model must be smaller than the original model. FormlSlicer is able to achieve this. Table 6.1 shows the statistics about the model size comparison. Because there are four types of model elements in FORML—state, region, transition and variable, we use them to measure a model's size. The original model consists of 109 states, 125 transitions, 58 regions and 84 variables. As FormlSlicer has multiple slicing processes[2], each considering a different feature as the FOI, we therefore have 7

[2] See Section 4.1 on the different slicing processes in the slicing task.

different sliced models. Each sliced model consists of the original FOI and the ROS with a reduced size.

Model	States			Transitions			Regions			Variables		
	S#	O#	S/O	S#	O#	S/O	S#	O#	S/O	S#	O#	S/O
Slice w.r.t. ACC	30	95	31.6%	24	108	22.2%	23	50	46.0%	20	79	25.3%
Slice w.r.t. LCA	17	101	16.8%	5	117	4.27%	13	53	24.5%	5	77	6.5%
Slice w.r.t. FCA	32	92	34.8%	27	107	25.2%	24	48	50.0%	23	81	28.4%
Slice w.r.t. LCC	36	93	38.7%	31	106	29.2%	28	49	57.1%	25	77	32.5%
Slice w.r.t. SLC	31	95	32.6%	26	108	24.1%	23	50	46.0%	22	79	27.8%
Slice w.r.t. AQS	6	90	6.7%	5	99	5.05%	3	51	5.9%	10	69	14.5%
Slice w.r.t. AC	0	97	0%	0	107	0%	0	50	0%	0	62	0%
AVG			23.0%			15.7%			32.8%			19.3%

Table 6.1: Empirical Results of FormlSlicer on the Automotive Case Study

Table Legend		
	S#	Number in the ROS of the Sliced Model
	O#	Number in the ROS of the Original Model
	S/O	S# / O#
	AVG	Average Percentage of S/O

Table 6.1 shows the size comparison of the ROS before and after slicing. We can see that the sliced ROS is smaller than the original ROS among all the sliced models. On average, the ROS of a sliced model has 23.0% of states, 15.7% of transitions, 32.8% of regions and 19.3% of variables of the ROS of the original model. This is a significant reduction on the model size.

The first observation from Table 6.1 is that not every sliced model has the same degree of reduction. Some sliced models are much smaller than the others. This is because the FOI in the sliced model has fewer feature interactions with the other features. For example, the sliced model with respect to AQS and the sliced model with respect to AC are both very small. The reason is that these two features are concerned with the air condition within the vehicle, whilst the other five features are concerned the vehicle's motion; in other words, they do not interact with the other five features. Thus, in these two sliced model, only a small portion of the ROS is retained and all the components related to ACC, LCA, FCA, LCC and SLC are not present. In addition, because the AQS feature is dependent on the AC feature (i.e., it constantly monitors whether AC is active or inactive), the sliced model with respect to AQS contains a small portion of the original AC. Another example is the LCA feature. It is concerned with the vehicle's steering direction (e.g., monitoring

the variable *car.steerDirection*), whilst the majority of features in the model (ACC, FCA and SLC) are concerned with the vehicle's moving speed and acceleration; thus, the sliced model with respect to LCA is smaller.

The second observation from Table 6.1 is that the number of regions is not as significantly reduced as other model elements in the sliced model. The reason is that FormlSlicer does not change the state hierarchy structure; sometimes, in order to keep a transition within an innermost region, FormlSlicer keeps many layers of state hierarchy. Figure A.12 is a good example on this problem. In order to keep a transition *LCCt5* within the region *centerCar*, all the parent states and regions (e.g., region *LCC*) are kept in the sliced model as well. In an ideal sliced model, these parent states and regions should be "sliced away" so that the transition *LCCt5* can be directly placed under the region *CCmain*. The same problem affects the reduction of states in the sliced model as well; but because of our state merging step, the reduction in the number of states is still better than the number of regions. This will be left for future work.

6.3 Properties of a Useful Sliced Model

In Section 1.1.2, we introduce that a useful sliced model must be correct, small and precise. We have demonstrated that a sliced model produced by FormlSlicer is correct (Chapter 5) and smaller than the original model (Section 6.2). However, in this thesis we do not measure precision of a sliced model in absolute terms. In some literature, the property of precision can be defined in a stronger sense that not only must the sliced model simulate the original model, but also the original model must simulate all observable actions of the sliced model[3]. FormlSlicer does not enforce this; but it has several steps that attempt to preserve the more useful portion of the original model in order to make the sliced model as precise as possible. For example, if a state has multiple outgoing transitions and one of the transitions leads to another state in the sliced model, then this state is preserved in the sliced model due to control dependence. Another example is that we use Korel et al.'s state merging rules that have been proved to preserve the precision of the original model [6]; in this way, we prevent producing a slice that is over-minimized due to state merging.

[3]See Section 2.3.2 for more details on how researchers define the correctness property of a model slicer.

Chapter 7

Conclusion

This chapter presents a summary of this thesis and its contributions, as well as possible directions for future work.

7.1 Summary of Thesis and Contributions

This thesis mainly consists of four parts:

Literature Survey on SBM Slicing
Chapter 2 presents the history, the challenges, the various existing slicing techniques, the various dependences' definitions and discussions on slicing correctness that are related to SBM slicing.

A Detailed Workflow
Chapter 3 and Chapter 4 propose a workflow of slicing on a feature-rich state-machine model. The workflow includes two tasks: a preprocessing task and a slicing task. The preprocessing task converts a feature-oriented state-machine model, expressed in FORML semantics, to a simpler intermediate representation—CFG—and computes three types of dependences within CFG. The slicing task forks off multiple processes, where each works on a different feature as the FOI and undergoes a multi-stage model slicing procedure.

Correctness
Chapter 5 presents the proof to show the correctness of the sliced model. It proves

that an execution trace in the original model can be simulated by at least one execution trace in the sliced model.

The Tool

The slicing tool—FormlSlicer—implements the proposed workflow. Chapter 6 presents empirical evaluations on this tool to show that the proposed slicing workflow can generate sliced models that are smaller than the original model.

In a nutshell, this thesis proposes a unique workflow on a hierarchical and concurrent state-machine model for requirements of feature-rich software systems, by coalescing various slicing techniques from the literature, and then evaluates the workflow theoretically and empirically.

7.1.1 Contributions

This thesis has the following contributions.

First, our slicing approach works on a state-machine model that is (1) hierarchical (including cross-hierarchy transitions), (2) concurrent, and (3) non-terminating. As explained in Section 2.2.2, these complex modeling constructs pose challenges and the existing techniques in literature only overcome these challenges partially. FormlSlicer tackles all challenges by adopting the following approaches:

- The challenge of having hierarchical construct is solved by converting the model to a group of control flow graphs and using hierarchy dependence to connect different layers of hierarchy. Therefore, the difficult task of manipulating a hierarchical structure is decomposed to many simple tasks of manipulating a flat graph structure.
- The challenge of having concurrent construct is also solved by using hierarchy dependence. Each parent state is mapped to multiple child states; each is the default initial state of a sub-machine inside the parent state.
- The challenge of having cross-hierarchy transitions is solved by preserving these transitions in the cross-hierarchy transition step in general iterative slicing stage.
- The challenge of a non-terminating state machine is solved by using the non-termination sensitive control dependence (NTSCD).

Second, we contribute a novel slicing algorithm that employs dependences to *construct* a model slice from the relevant model elements and then enriches the model slice until its states preserve the reachability properties of the original model. As explained in Section 2.2.2, due to the difficulty of maintaining the well-formedness property of a model slice, many model slicing approaches are very conservative about removing transitions or states that can break the graph connectivity; thus, the reduction in their model slicers can be minor. In our approach, we use the state connecting step to avoid this problem and thus we can afford to be more aggressive in reducing model size.

Third, we coalesce different existing slicing techniques to form a unique workflow. We directly borrow the following concepts:

- Korel et al.'s state merging rules [6],
- Korel et al.'s definition in data dependence [6],
- Wang et al.'s definition in refinement control dependence [20] (equivalent to the "HDtable2" in this thesis),
- Ranganath et al.'s definition in NTSCD [21];

We adapt the following for our own use:

- Ojala's idea of converting a state-machine model to CFGs [18],
- Ranganath et al.'s computation algorithm of NTSCD [21],
- and Kamischke et al.'s use of a model enrichment step in the slicing algorithm [19].

Fourth, our approach employs a novel decomposition of dependence analyses and slicing tasks that enables parallel construction of multiple model slices. It is advantageous for any slicing works that aim to produce n slices from an original model with respect to n components within the original model.

Fifth, we make automation of slicing easier by simplifying a complex transition label in FORML. We tackle the challenges of a complex transition label in the following ways:

- The "inState()" expression in guard condition becomes a subtype of data dependence (DD);

- The event generation and receiving in the WCE and WCA are transformed to two groups of variables—monitored variables and controlled variables.

Sixth, we prove (by simulation) that all behaviors in the original model are preserved in the sliced model. To our knowledge, this is the first proof of correctness of a construction-based slicer (vs. a slicer that forms a model slice by simply removing model elements from the original model).

Lastly, we implement a tool to show that this workflow is feasible and it can produce model slices that are significantly smaller than their original model.

7.2 Future Work

The following are possible directions for extending the work presented in this thesis.

7.2.1 Extending to Slicing on Software-Product-Line Model

We plan to lift our approach so that it operates on software-product-line models. At present, the input to the slicer is a model of a *product* comprising multiple features. Instead, the composition of a collection of features could be a *product line* representing a set of products that are differentiated by which features are present and which are absent. One idea is to generalize data and control dependencies to be conditional on the presence of the features that execute data or control actions. Exploring this idea is left for future work.

7.2.2 Bridging the Gap between FormlSlicer's Input Model and FORML model

We mentioned in Section 3.3 that FormlSlicer does not perform slicing on the world model. Because the theme of this thesis is about slicing on SBMs, we do not consider slicing on the world model, which is based on UML class-diagram constructs. However, we believe that slicing on the world model is just an implementation work. We can use the set of relevant variables in FormlSlicer's slicing output and match them against the class fields in the world model. If a class field is matched, it is preserved in the sliced model; otherwise it is sliced away. If all fields in a class are sliced away, the class is also sliced away. This can be done either manually, or by writing a small program to automate it.

FormlSlicer's input model also have some other differences with FORML model. For example, FormlSlicer poses a restriction on the input model that no state within an orthogonal region that has a sibling orthogonal region can have an outgoing transition that exits the region. In future work, this restriction can be relaxed. Another example is that FORML has some semantics that are specifically used for communications among different products; but FormlSlicer does not consider that. As explained in Section 6.1, this is the reason why the telephony case study in Shaker's thesis [28] is not suitable for our empirical evaluations. This will be left for future work.

In addition, FormlSlicer only deals with a model of all features currently making up a software system. If new features are added to the software system, then slicing needs to be performed from scratch. To enable slicing on an evolving software system, perhaps we can design a mechanism to save past slicing results.

We also plan to extend our dependence analyses and slicing algorithms to accommodate feature-oriented models in which some features are *machine fragments* rather than standalone parallel machines. In such models, a fragment will necessarily depend on the feature machine that it extends, but it may be possible to slice away other (irrelevant) fragments of the same base machine.

7.2.3 Improving the Workflow

Improving the Dependences

The computation algorithm of CD can be further optimized. Algorithm 4.2 re-starts the computation of paths representations for each different branching node. However, by traversing the path from a branching node n_i to a reachable node n_j, the algorithm may have encountered another branching node n_x along the path and have collected partial information about the dependence relationship between n_j and n_x; if we can find some ways to reuse this information, we may not need to start from scratch in computing the paths representations from n_x to n_j in another iteration. This will be left for future work.

Improving the Multi-Stage Model Slicing Process

As mentioned in Section 6.2, the number of regions is not as significantly reduced as other model elements in the sliced model, because our model slicing process does not distort the state hierarchy of the original model at all. However, in some special cases (e.g., Figure A.12), it is preferable to merge several layers of state hierarchy together so that

the sliced model looks more neat. In future work, we can study on how to merge different layers of state hierarchy of a model without making the sliced model incorrect.

Another issue is that currently we preserve all cross-hierarchy transitions in the sliced model[1]. Taking away any cross-hierarchy transitions while maintaining the correctness of the sliced model will be a non-trivial task. We can conduct another independent project to specifically study on slicing a hierarchical state machine with cross-hierarchy transitions.

7.2.4 Customization of Slicing in FormlSlicer

The FormlSlicer tool can be extended by adding more customization options for the users.

Only Interested in a Few Features but not All Although the use of concurrent slicing processes is efficient in producing multiple sliced models with respect to different features, this is an overkill to someone who just want to focus on one feature. FormlSlicer can be extended to allow users to specify which features they want to focus and then only forks off a few slicing processes.

Adjusting Degree of Reduction and Precision A more advanced FormlSlicer can allow users to customize the degree of reduction and precision in the sliced models, depending on the users' needs. More aggressive state merging rules can be implemented in FormlSlicer; they all aim to reduce the size of the sliced model further. Users can be allowed to select which state merging rules to use during the multi-stage model slicing process. The more state merging rules they choose to use, the smaller and (potentially) more imprecise the sliced models will become. In this way, we can leave the decision over trade-off between degree of reduction and precision to users.

7.2.5 Making the Correctness Proof More Rigorous

Currently, we do not provide a precise definition of the semantics of the model due to its complexity. The semantics of the model is usually represented as a tuple that consists of domains of model elements, relations between model elements, functions of the behavior

[1] We determine that due to the complexities brought by any cross-hierarchy transitions in the FOSM, these transitions need to be preserved in order for the sliced model to correctly simulate the original model. See Section 4.4.2.2 for a more detailed explanation.

of the model that involve the model elements, and possibly more. As explained in Subsection 5.5.4.1, missing such a precise definition results in a gap between the theorem we have proved and the simulation relation that we want to prove. A possible future work can make the correctness proof more rigorous by defining the concepts formally. This requires a non-trivial amount of efforts.

Appendix A

Automotive: A Slicing Example

The original model, as shown in Figure A.1, consists of seven FOSMs—ACC (Figure A.2), FCA (Figure A.3), LCA (Figure A.4), LCC (Figure A.5), SLC (Figure A.6), AQS (Figure A.7) and AC (Figure A.8).

Figure A.9 shows the sliced automotive model with respect to the feature of LCA. It consists of the original LCA (because it is the FOI) and a sliced ROS. The sliced FOSMs in the ROS are shown in Figure A.10, Figure A.11, Figure A.12 and Figure A.13. The FOSMs of feature AQS and AC are not present in the sliced model.

Figure A.14 shows the sliced automotive model with respect to the feature of ACC. It consists of the original ACC (because it is the FOI) and a sliced ROS. The sliced FOSMs in the ROS are shown in Figure A.15, Figure A.16, Figure A.17 and Figure A.18. The FOSMs of feature AQS and AC are not present in the sliced model.

Figure A.1: The original automotive model

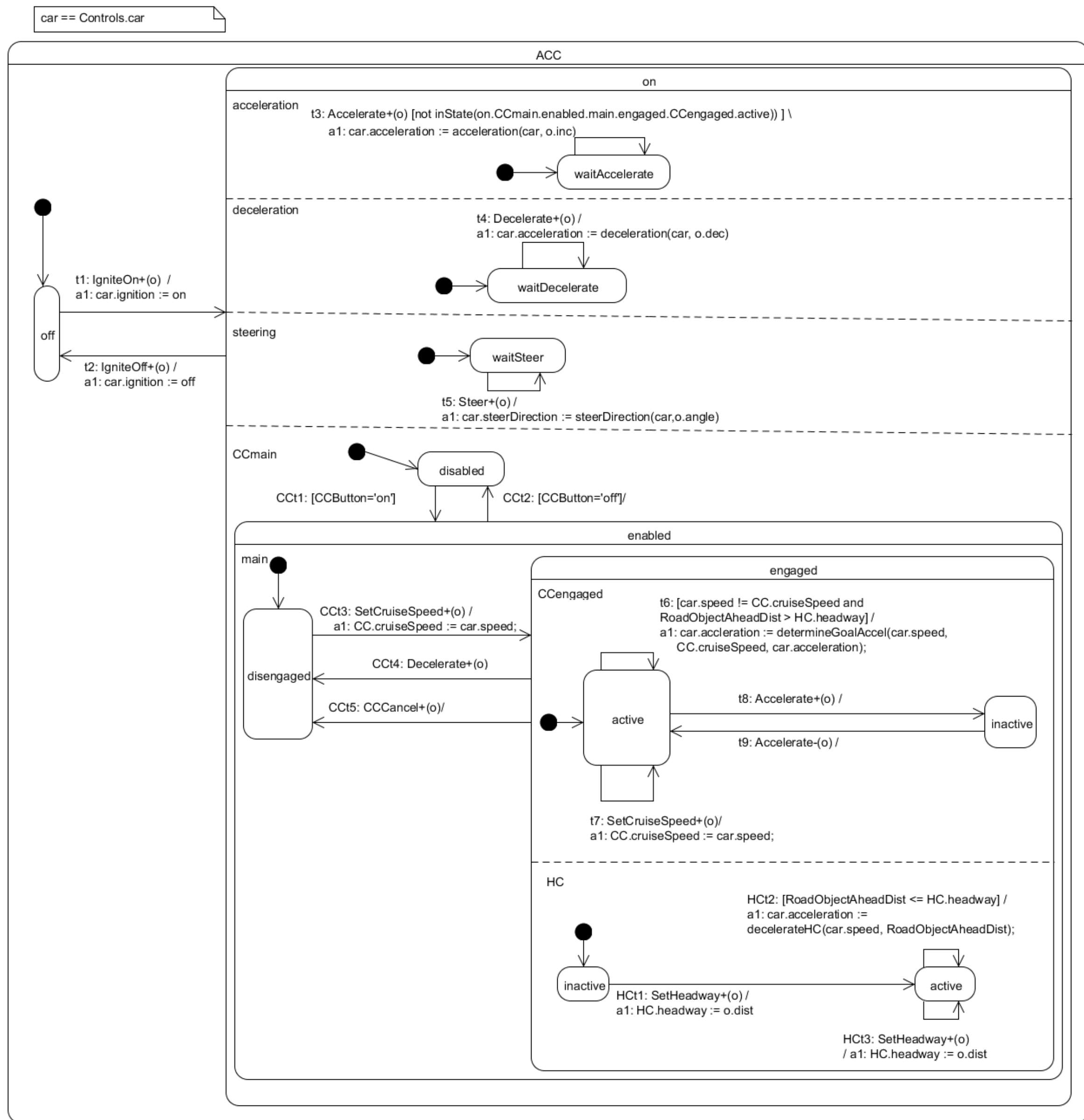


Figure A.2: Original ACC feature of the automotive model

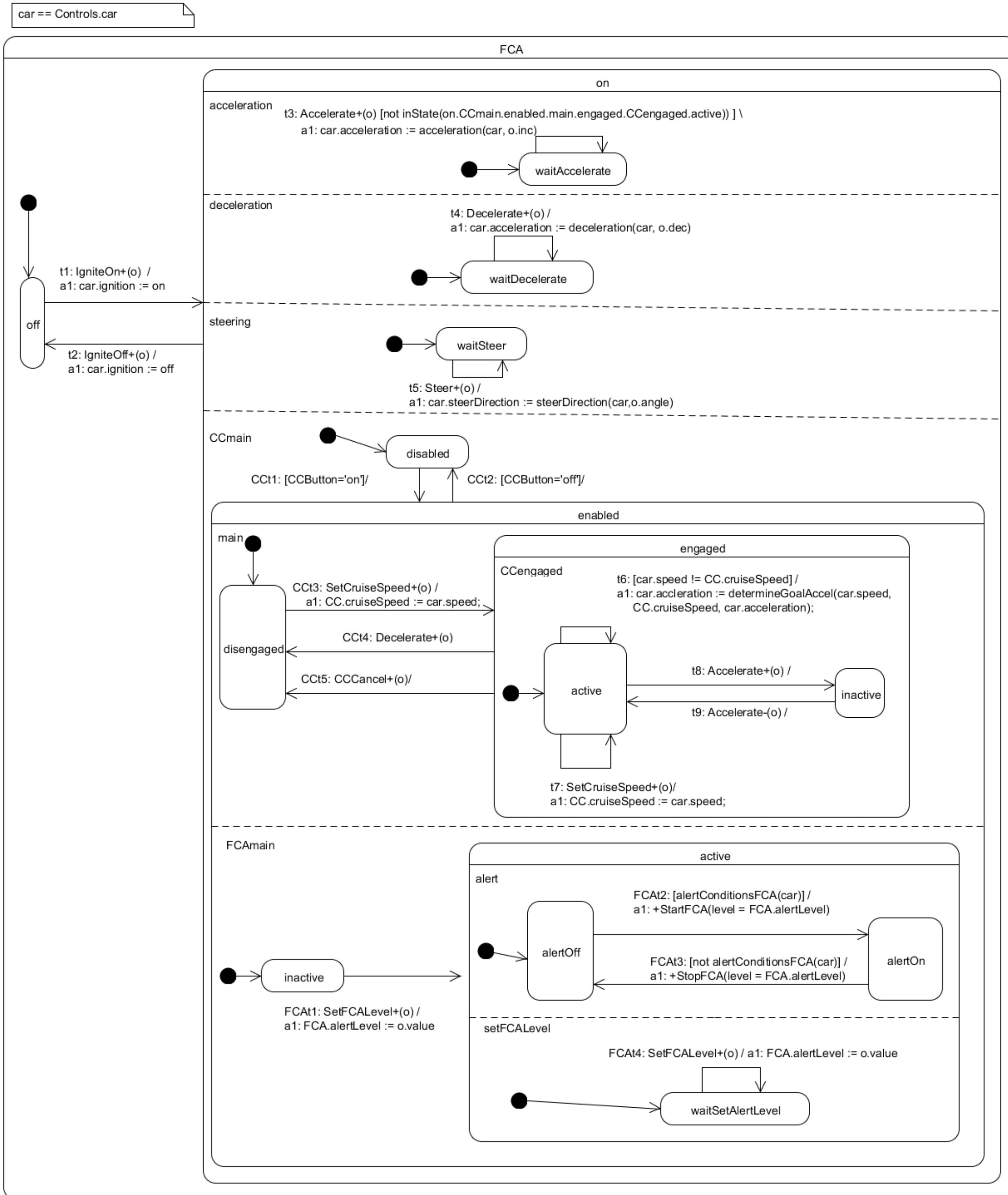


Figure A.3: Original FCA feature of the automotive model

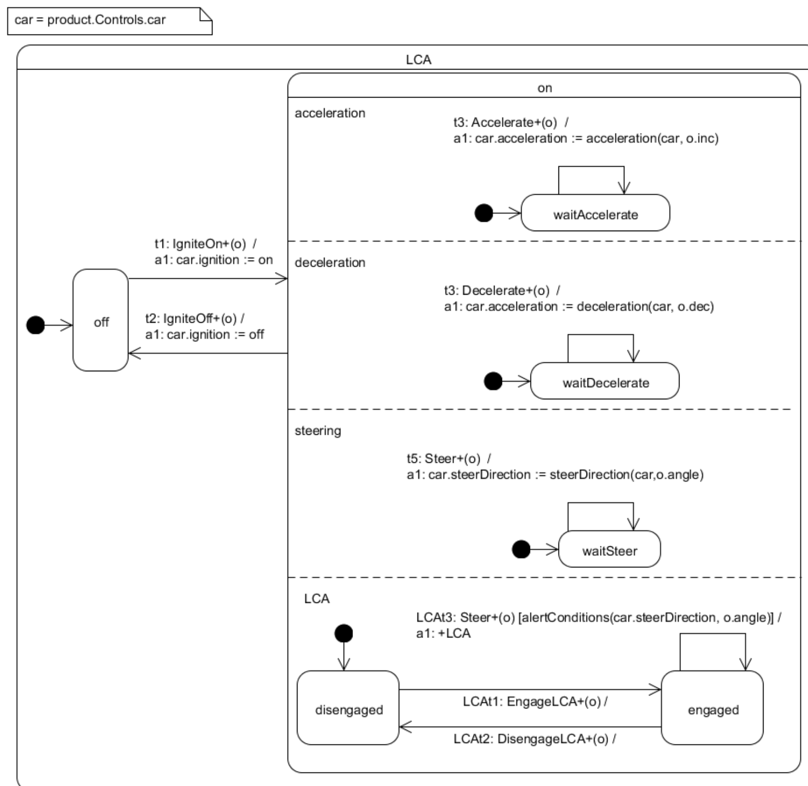


Figure A.4: Original LCA feature of the automotive model

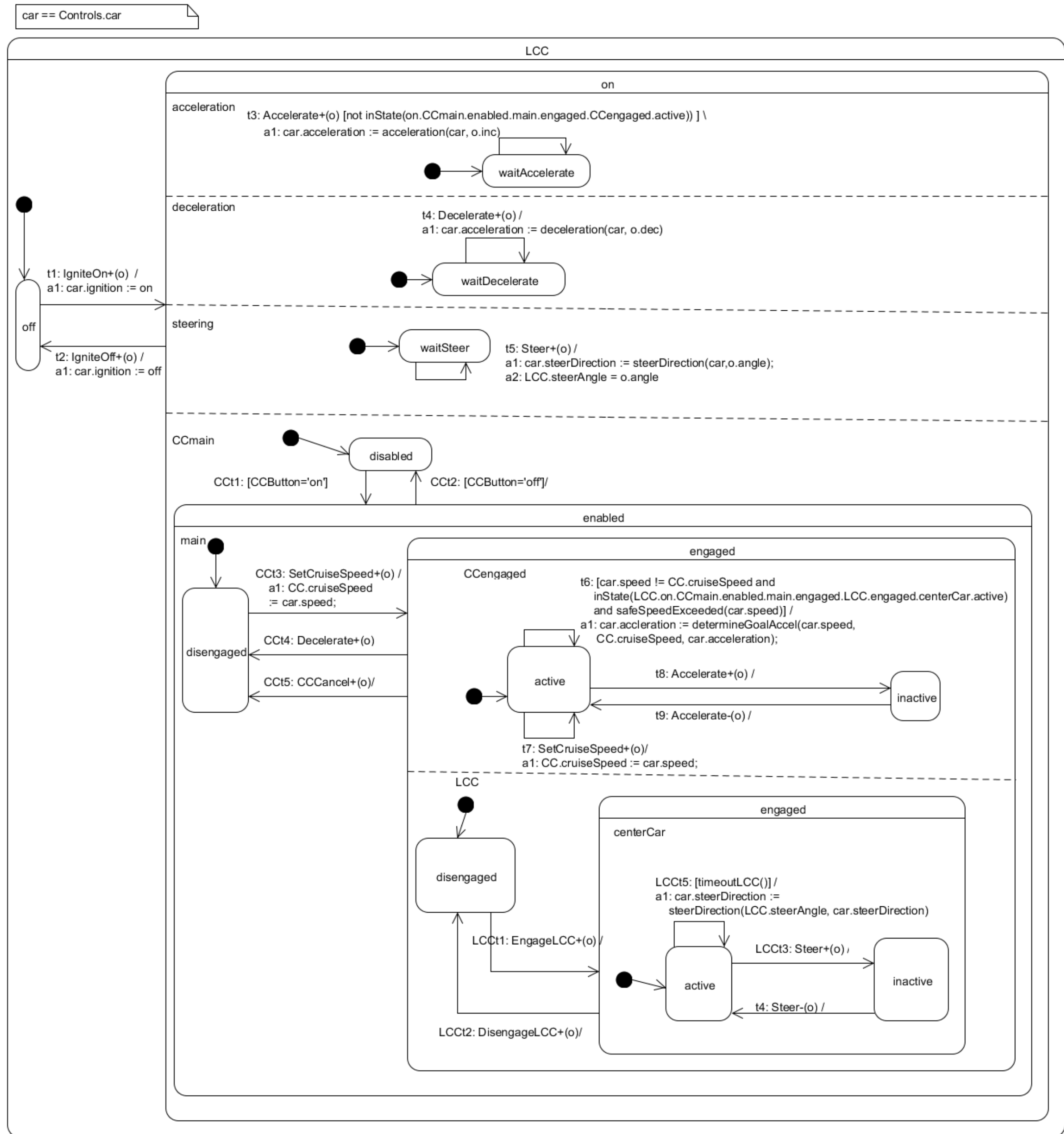


Figure A.5: Original LCC feature of the automotive model

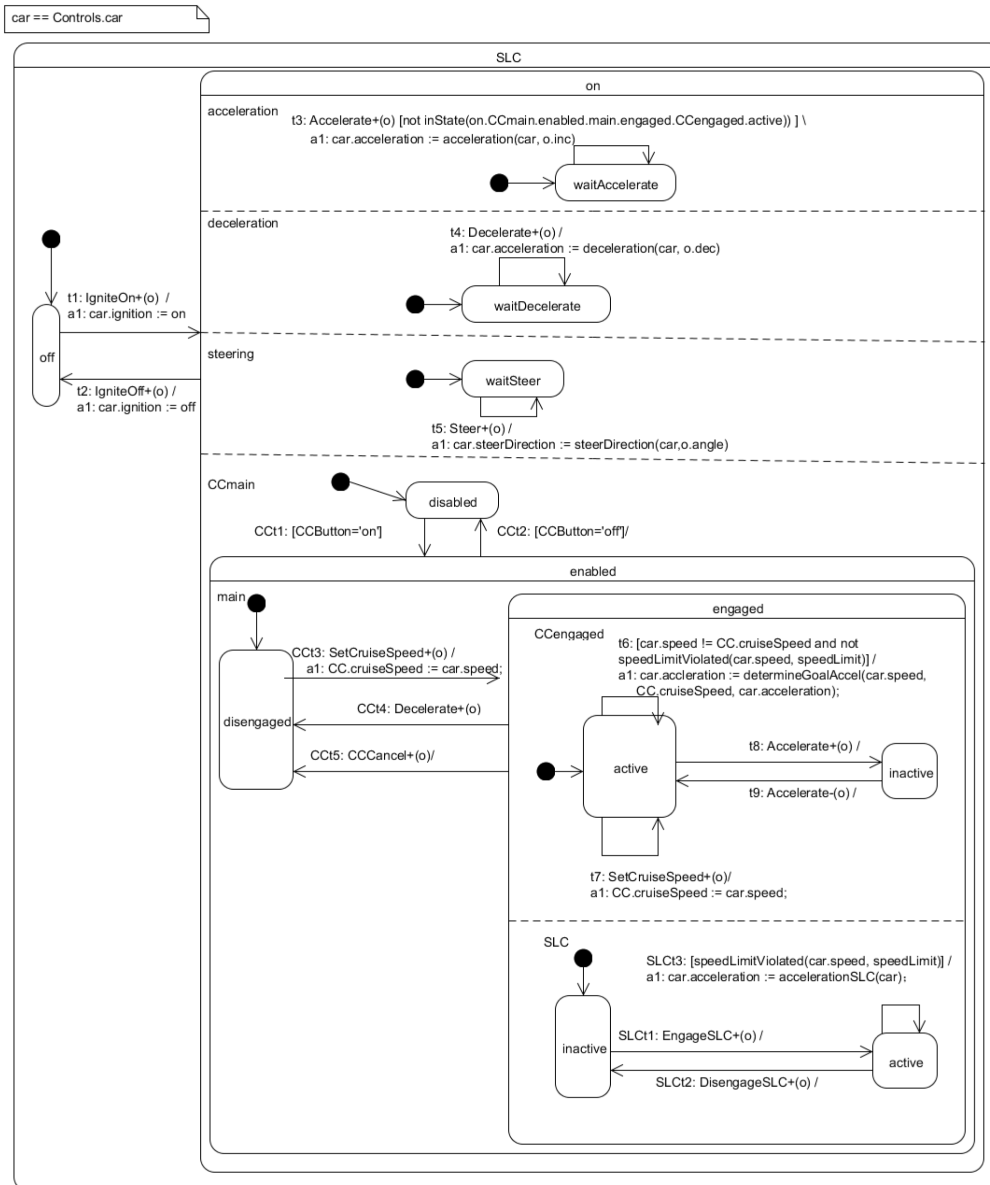


Figure A.6: Original SLC feature of the automotive model

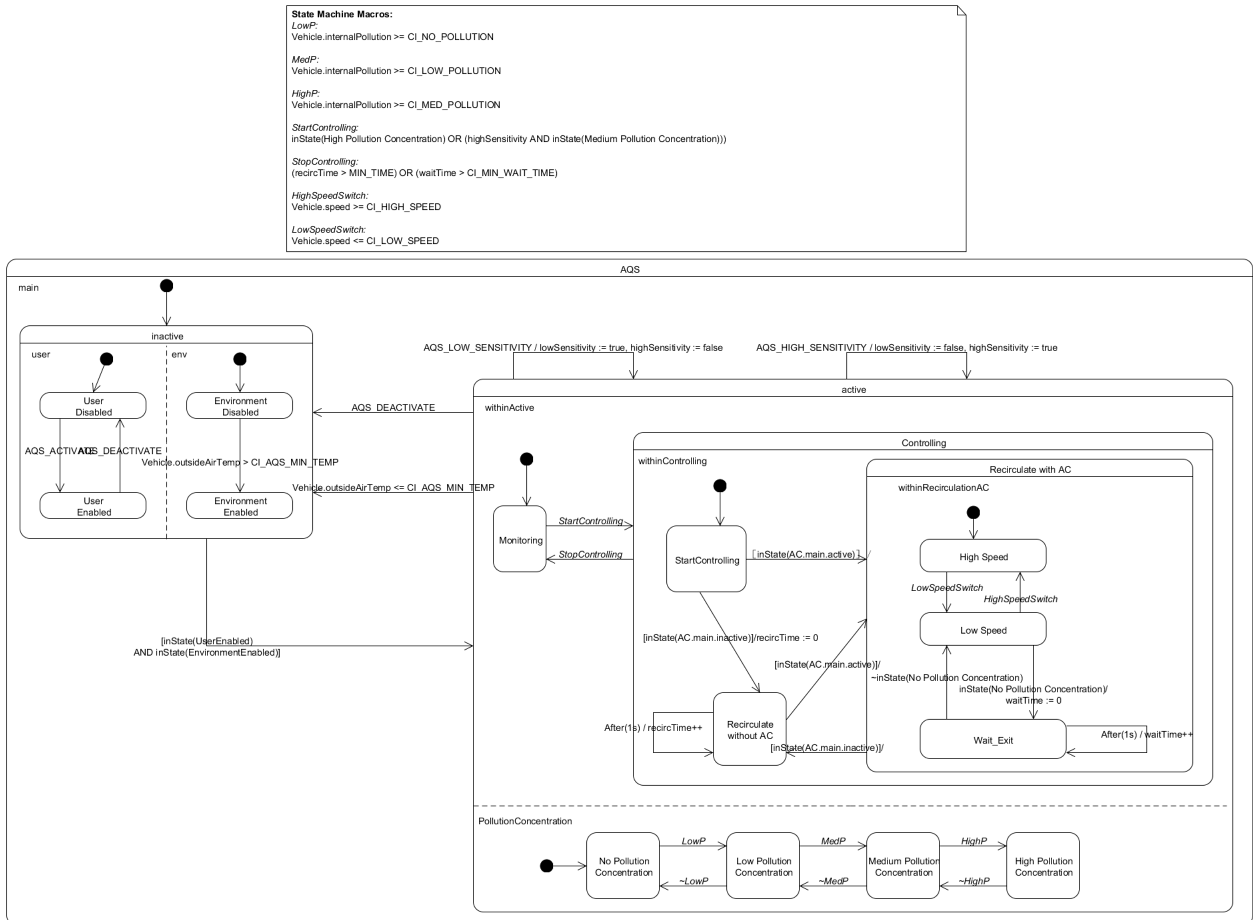


Figure A.7: Original AQS feature of the automotive model

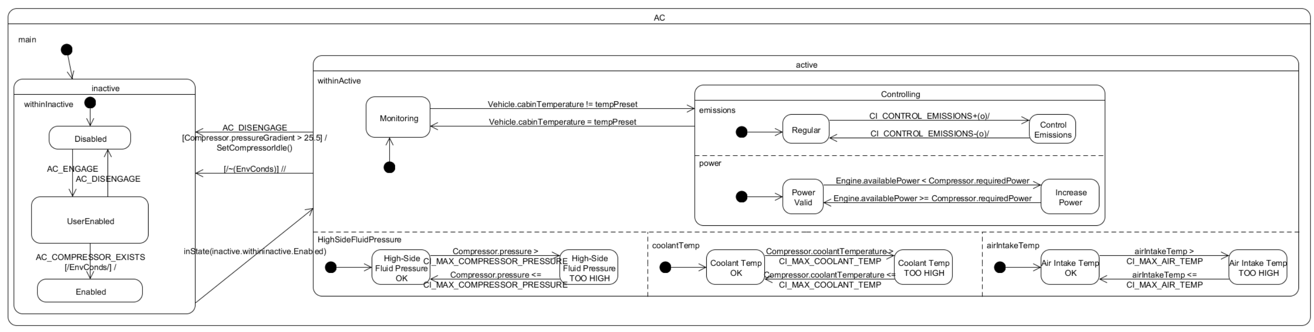


Figure A.8: Original AC feature of the automotive model

Figure A.9: The sliced model w.r.t. LCA

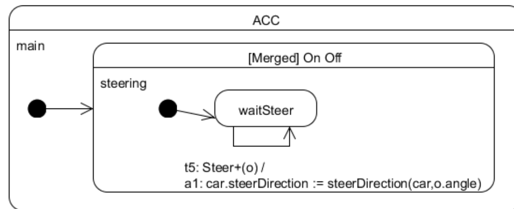


Figure A.10: Sliced ACC feature w.r.t. LCA

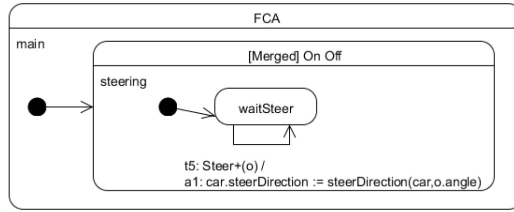


Figure A.11: Sliced FCA feature w.r.t. LCA

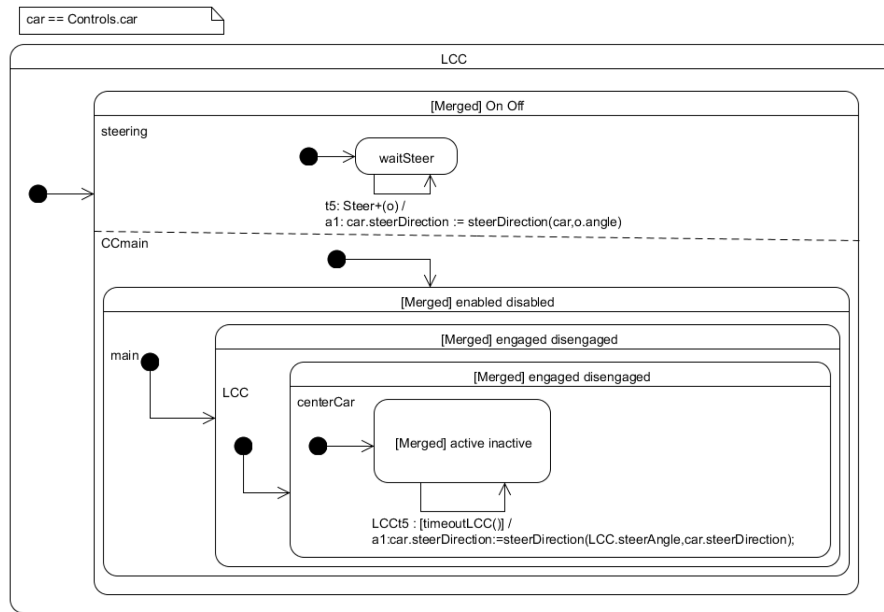


Figure A.12: Sliced LCC feature w.r.t. LCA

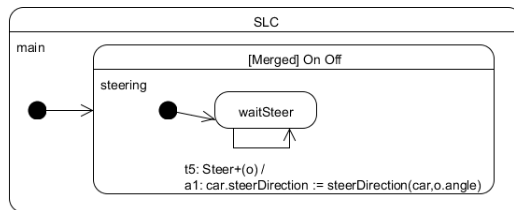


Figure A.13: Sliced LCC feature w.r.t. LCA

Figure A.14: The Sliced Model w.r.t. ACC

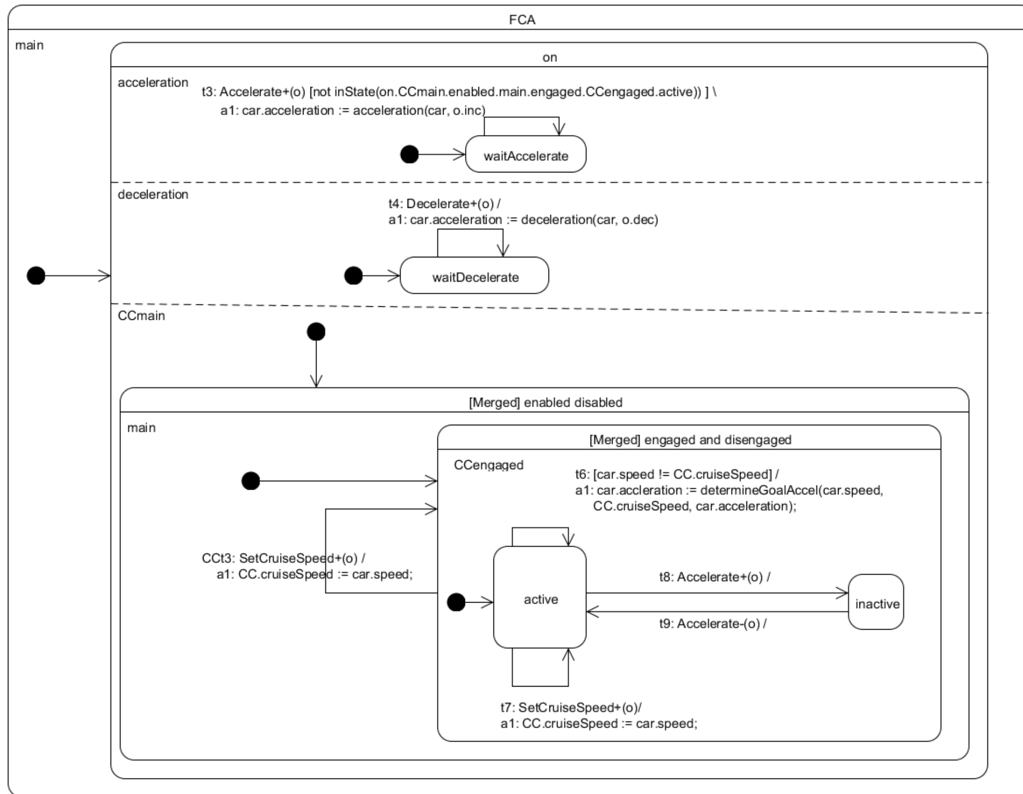


Figure A.15: Sliced FCA feature w.r.t. ACC

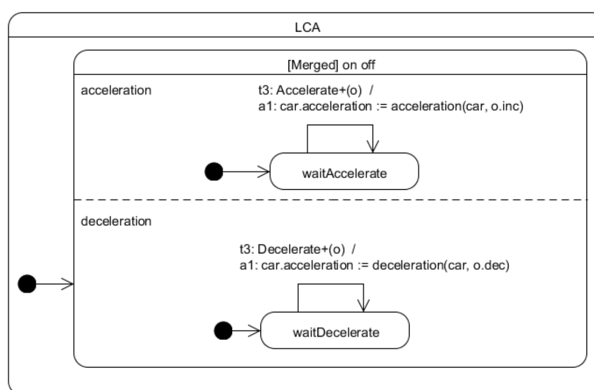


Figure A.16: Sliced LCA feature w.r.t. ACC

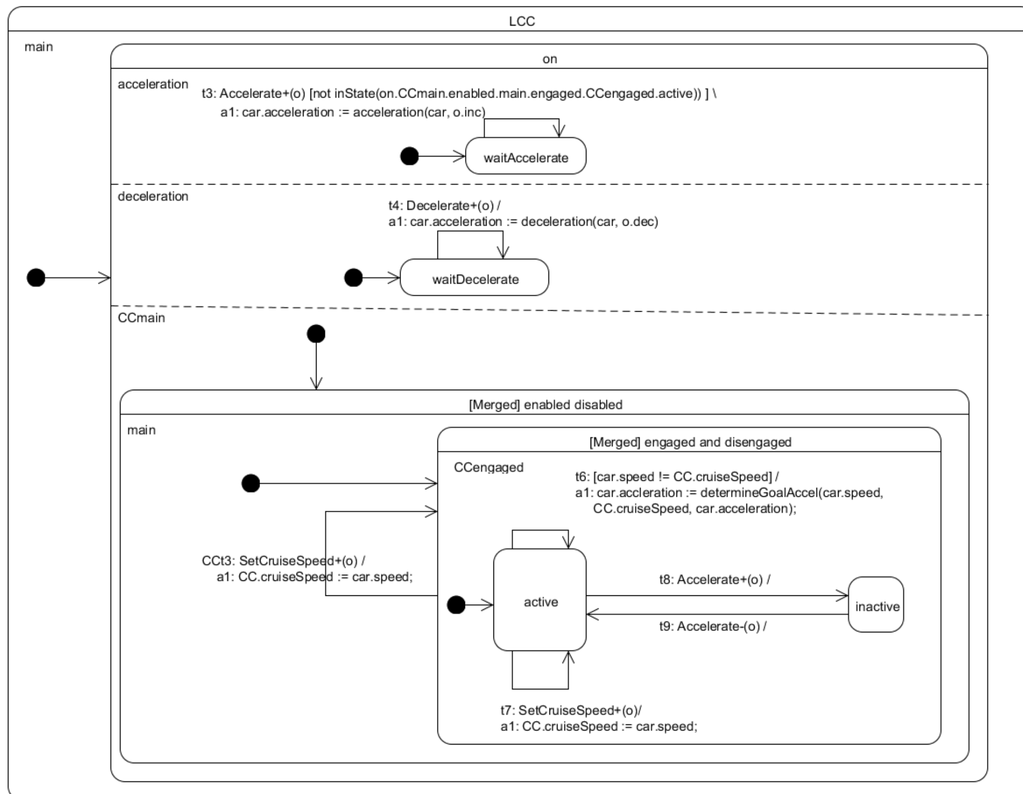


Figure A.17: Sliced LCC feature w.r.t. ACC

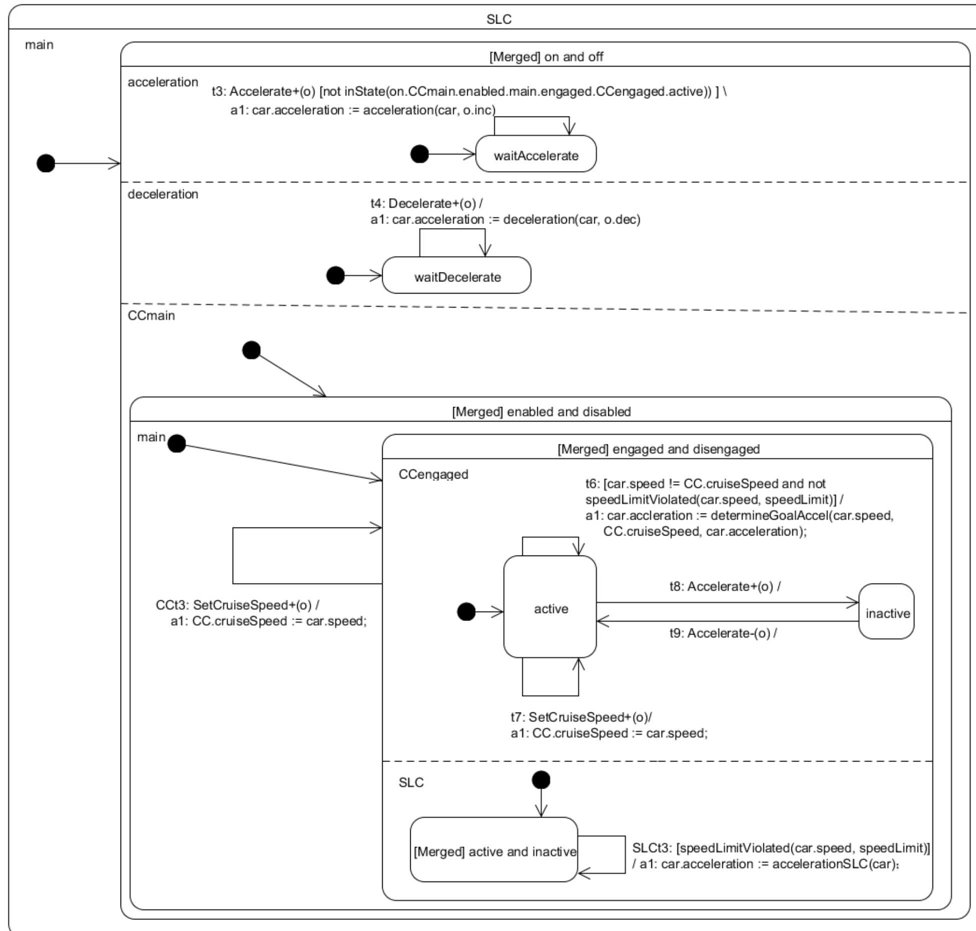


Figure A.18: Sliced SLC feature w.r.t. ACC

Appendix B

Supporting Functions for Control Dependency Algorithm

This chapter lists all the supporting functions for Algorithm 4.2. Their respective function goals have been listed in Table 4.5.

Algorithm B.1: HasNonEmptyPathsFromNode1

```
Function HasNonEmptyPathsFromNode1 (branchIndex) :
    if p[branchIndex]=="!" OR p[branchIndex] is null then return false
    else return true
end
```

Algorithm B.2: IsControlDependentOn1

```
Function IsControlDependentOn1 (index) :
    if p[index]=="!" OR p[index] is null then return false
    set paths := array.split(p[index], ";");
    foreach path in paths do
        if path is single-length then return true
    end
    return false
end
```

Algorithm B.3: ReducePaths

```
Function ReducePaths (index) :
    if p[index] is null then return;
    set paths := array.split(p[index], ";");
    if paths.length ≤ 1 then return;
    SortPaths (paths);
    set start := paths.lastIndex;
    set i : = start - 1;
    set targetIndices := ∅;
    ADD the target index of the last subpath of paths[start] into targetIndices
    while true do
        if paths[i] and paths[start] are the same except the last target index then
            ADD the last target index of paths[i] to targetIndices Decrement i;
        else
            if targetIndices.size > 1 then
                set srcIndex := source index of last subpath of paths[start];
                set n := allNodes[srcIndex];
                if n.outgoingNodes.size == targetIndices.size then
                    // reduction occurs
                    for j from i+2 to start do
                        set paths[j] := null;
                    end
                    Delete the last subpath in paths[i+1]
                end
            end
            if i == -1 then break;
            reset targetIndices := ∅;
            reset start := index of the last unprocessed path in paths
            ADD the target index of the last subpath of paths[start] to targetIndices;
            reset i := start-1;
        end
    end
    p[index] = paths.join(";");
    return
end
```

Algorithm B.4: UnionPath

```
Function UnionPath (prevIdx, nextIdx) :
    if p[nextIdx] == "!" then return false;;
    if p[nextIdx] is null OR NOT HasNonEmptyPathsFromNode1 (prevIdx) then
        p[nextIdx] := p[prevIdx];
        return false;
    end
    set prevIndexSet := array.split(p[prevIdx], ";");
    set nextIndexSet := array.split(p[nextIdx], ";");
    set changed := false;
    foreach prevP in prevIndexSet do
        set duplicate := false;
        foreach nextP in nextIndexSet do
            if prevP starts with nextP then
                // If next path is prefix of previous path, it indicates a cycle
                reset duplicate := true;
                reset changed := true;
                break
            end
        end
        if duplicate == false then APPEND prevP to p[nextIdx];;
    end
    return changed
end
```

Algorithm B.5: ExtendPath

```
Function ExtendPath (srcIndex, branchIndex) :
    if p[srcIndex] == null then set p[branchIndex]:="srcIndex:branchIndex";
    if p[srcIndex] == "!" then set p[branchIndex]:="!";
    set srcIndexArr := array.split(p[srcIndex], ";")
    foreach i from 1 to srcIndexArr.size do
        APPEND "srcIndex:branchIndex" to srcIndexArr[i];
    end
    set p[branchIndex]:=srcIndexList.join(";")
end
```

References

[1] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[2] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[3] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. State-based model slicing: A survey. *ACM Computing Surveys (CSUR)*, 45(4):53, 2013.

[4] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[5] Torben Amtoft, Kelly Androutsopoulos, and David Clark. Correctness of slicing finite state machines. *RN*, 13:22, 2013.

[6] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state-based models. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 34–43. IEEE, 2003.

[7] C Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, 1999.

[8] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[9] Cecylia Bocovich. A feature interaction resolution scheme based on controlled phenomena. Master's thesis, University of Waterloo, 2014.

[10] Grady Booch. *Object Oriented Analysis & Design with Application*. Pearson Education India, 2006.

[11] Pourya Shaker, Joanne M Atlee, and Shige Wang. A feature-oriented requirements modelling language. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 151–160. IEEE, 2012.

[12] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[13] Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing*, 20(6):563–595, 2008.

[14] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[15] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[16] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[17] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[18] Vesa Ojala. *A slicer for UML state machines*. Helsinki University of Technology, 2007.

[19] Jochen Kamischke, Malte Lochau, and Hauke Baller. Conditioned model slicing of feature-annotated state machines. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, pages 9–16. ACM, 2012.

[20] Wang Ji, Dong Wei, and Qi Zhi-Chang. Slicing hierarchical automata for model checking uml statecharts. In *Formal Methods and Software Engineering*, pages 435–446. Springer, 2002.

[21] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):27, 2007.

[22] Kelly Androutsopoulos, David Clark, Mark Harman, Zheng Li, and Laurence Tratt. Control dependence for extended finite state machines. In *Fundamental Approaches to Software Engineering*, pages 216–230. Springer, 2009.

[23] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*, volume 289. Morgan Kaufmann San Francisco, 2002.

[24] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *Software Engineering, IEEE Transactions on*, 16(9):965–979, 1990.

[25] Kelly Androutsopoulos, Nicolas Gold, Mark Harman, Zheng Li, and Laurence Tratt. A theoretical and empirical study of efsm dependence. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 287–296. IEEE, 2009.

[26] Robin Milner. *An algebraic definition of simulation between programs*. Citeseer, 1971.

[27] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifying safety properties of a powerpc- microprocessor using symbolic model checking without bdds. In *Computer Aided Verification*, pages 60–71. Springer, 1999.

[28] Pourya Shaker. *A feature-oriented modelling language and a feature-interaction taxonomy for product-line requirements*. PhD thesis, University of Waterloo, 2013.

[29] Joanne M Atlee, Sandy Beidu, Nancy A Day, Fathiyeh Faghih, and Pourya Shaker. Recommendations for improving the usability of formal methods for product lines. In *Formal Methods in Software Engineering (FormaliSE), 2013 1st FME Workshop on*, pages 43–49. IEEE, 2013.

[30] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[31] Cecylia Bocovich and Joanne M Atlee. Variable-specific resolutions for feature interactions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 553–563. ACM, 2014.

[32] Alexander Knapp and Stephan Merz. Model checking and code generation for uml state machines and collaborations. *Proc. 5th Wsh. Tools for System Design and Verification*, pages 59–64, 2002.

[33] Jianwei Niu, Joanne M Atlee, and Nancy A Day. Template semantics for model-based notations. *Software Engineering, IEEE Transactions on*, 29(10):866–882, 2003.

[34] Dániel Varró. A formal semantics of uml statecharts by model transition systems. In *Graph Transformation*, pages 378–392. Springer, 2002.

[35] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[36] David Dietrich. A mode-based pattern for feature requirements, and a generic feature interface. Master's thesis, University of Waterloo, 2013.

[37] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. A comparison of product-based, feature-based, and family-based type checking. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 115–124, New York, NY, USA, 2013. ACM.

[38] David Dietrich and Joanne M Atlee. A mode-based pattern for feature requirements, and a generic feature interface. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 82–91. IEEE, 2013.

[39] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal*, 20(3-4):567–604, 2012.

[40] Paul Clements and Linda Northrop. *Software product lines: practices and patterns.* Addison-Wesley, 2002.