FormlSlicer: A Model Slicing Tool for Feature-oriented Requirements in Software Product Line

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

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 used in feature-oriented requirements in software product line (SPL) usually contains a large number of non-trivial features; each feature may have unintended interactions with one another. It is difficult to comprehend or verify such a model. Model slicing is a useful approach to overcome such a challenge by reducing the model size while preserving 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, slicing an SBM is much more challenging than program slicing. Among a diverse range of SBM slicing approaches, dependence-based analysis is the most popular; it relies on the computation of dependence relations among states and transitions in order to determine what model elements of the original SBM can be in the sliced SBM.

We propose a unique slicing workflow on the behavior models in feature-oriented requirements modeling language (FORML), which is a language for modeling feature-oriented requirements in an SPL. Each feature module in the model is considered as a complete state machine with states, regions and transitions; this is called a feature-oriented state machine (FOSM). The slicing workflow has two tasks—the preprocessing task and the slicing task. The preprocessing task mainly computes three types of dependences: hierarchy dependence (HD), which represents the state hierarchy relation among states; data dependence (DD), which captures the define-use relationship among transitions with respect to a variable; and control dependence (CD), which captures the notion on whether one state can decide the execution of another state or transition. The slicing task employs multiple slicing processes on the model; 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). Each slicing process constructs a sliced model which preserves the portion of the original model that interact with the FOI. The construction process is called Multi-Stage Model Slicing Process; it firstly identifies an initial set of transitions in the ROS that affect the FOI, then finds more states and transitions in the transitive closure of dependences and eventually enriches the model elements in the slice set to create a well-formed, executable model.

A correctness proof is performed to show that a sliced model produced by this slicing workflow can 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 conduct an empirical evaluation on FormlSlicer and demonstrate that on average the sliced model has 45.8% of states, 51.2% of variables, 57.8% of regions and 37.6% of transitions of the original model. This shows that our proposed slicing workflow results in a significantly smaller 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

List of Figures

List of Algorithms

Glossary

ACC adaptive cruise control. xii, xiii, 84, 86, 91, 94, 96, 101, 103–106

BDS basic driving service. 22

CC cruise control. 22

CD control dependence. iii, 11, 14, 38, 41, 70

CFG control flow graph. 7, 10, 15–18, 29, 30, 32, 33, 38, 39, 45, 48, 60, 88, 90

DD data dependence. iii, 11, 14, 30, 34, 35, 70, 72, 89

EFSM extended finite-state machine. 2, 11, 14, 17, 19

FCA forward collision alert. xii, xiii, 84, 86, 94, 97, 102, 104

FOI feature of interest. iii, 5–8, 25, 28, 29, 35, 46–48, 60, 62, 72, 85, 86, 88, 94

FORML feature-oriented requirements modeling language. iii, 6–8, 12, 21, 22, 24, 27, 34, 35, 38, 65, 66, 84, 85, 88–90, 93

FOSD feature-oriented software development. 4, 21

FOSM feature-oriented state machine. iii, xi, 7, 8, 22–26, 28–31, 33, 35, 47–49, 56, 60, 63, 65–67, 72, 76, 84–87, 90, 92, 94, 107, 108

HC headway control. 4

HD hierarchy dependence. iii, 30, 34, 70, 89

LCA lane change alert. xii, xiii, 84, 86, 91, 94, 98, 101, 102, 104

LCC lane centring control. xiii, 85, 94, 99, 102, 105

NTSCD non-termination sensitive control dependence. 16–18, 38, 39

ROS rest of the system. iii, 5–7, 25, 28, 29, 47–49, 62, 63, 86, 94, 108

SBM state-based model. iii, 2, 3, 6, 7, 11, 12, 14, 21, 22, 27, 38, 51, 62, 88–90

SLC speed limit control. xiii, 4, 85, 86, 94, 100, 106

SNode A Node in CFG representing a state in FORML model. 32, 33, 45, 47, 54

SPL software product line. iii, 3–6, 8, 21, 23, 25, 89, 92, 93

TNode A Node in CFG representing a transition in FORML model. 32, 33, 36, 45, 47, 54

WCA world change action. 24, 25, 27

WCE world change event. 24, 25, 27

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 machine (EFSM), state charts, 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 can simulate 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. Some may define correctness in a stronger sense such that not only must the sliced model

[1] We summarize these based on information learnt from various literature on model slicing. See Section 2.2.5 for the literature survey.

simulate the original model, but also the original model must simulate all observable actions of the sliced model[2] [4]. In this thesis, we will only enforce the first correctness property.

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 [5]. 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.

1.2 Feature-oriented Model Slicing

Model slicing techniques can be applied in many different domains. This thesis focuses on one specific domain—feature-oriented requirements for a software product line (SPL).

This section will present some background about feature-oriented requirements of an SPL, as well as feature interactions that exist in most SPLs. 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 SPL Requirements

Many organizations develop a family of similar software systems in the same domain, called an SPL. For example, an automotive manufacturer can design a series of new vehicle models

[2] See Section 2.2.5 for more details on how researchers define the correctness property of a model slicer.

[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.

that are close variants of one another. These software systems have different combinations of features. For example, the 2015 Buick Encore premium model is equipped with the Forward Collision Alert feature whilst the convenience model in the same SPL is not; yet both convenience and premium models have many other common features, such as the Traction Control feature.

In light of this, the feature-oriented software development (FOSD) paradigm is usually adopted in SPL requirements. The paradigm itself advocates the use of system features as the primary criterion to identify separate concerns when developing a single software system or an SPL. A feature is defined as a cohesive set of system functionality, that can be represented as a grouping or modularization of individual requirements within the system specification [6]. By applying the FOSD paradigm in SPL requirements, an SPL model can be modeled as a finite set of feature modules, each representing one feature's requirements. Based on that, a system variant in the SPL can be obtained by selecting a subset of these feature modules.

One well-known challenge of applying the FOSD paradigm in SPL requirements is managing feature interactions. In software requirements, features are usually modeled in isolation by different groups of software engineers; this is especially the case when there are hundreds of non-trivial features. When these features are added to a system, different features can influence one another in determining the overall properties and behaviors of the overall system [7].

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 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 acceleration may be set to an unpredictable value [8].

In a single system, such feature interactions can be pre-determined during the modeling stage. But in an SPL, different system variants can be equipped with different subsets of features, resulting in a possibly exponential number of feature combinations to assess for quality assurance; therefore, detecting feature interactions in an SPL is a complicated task.

1.2.2 Model Slicing used in Feature-oriented Requirements in SPLs

Model slicing is a useful technique to simplify the SPL 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 an SPL model with respect to a single feature, called the feature of interest (FOI), so that the rest of the huge SPL 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 SPL model form the rest of the system (ROS) in the SPL model; only a relevant portion of ROS will be kept after slicing. The (possibly) smaller ROS in a sliced SPL will form a "smaller context" for the FOI.

In Subsection 1.2.2.1 and Subsection 1.2.2.2, we explain why this is useful in detecting unintended feature interactions in an SPL from the perspectives of model comprehension and model checking, respectively.

1.2.2.1 Model Comprehension of Feature-oriented Requirements in SPLs

As described in Subsection 1.2.1, features are usually modeled in isolation by different groups of software engineers. When there are hundreds of features, which is typically the case in an SPL, 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, engineers will be overwhelmed if they try to comprehend the whole SPL model and all of its variants. 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 [9].

Feature-oriented model slicing can facilitate model comprehension of one feature (i.e., the FOI) at a time. 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.

Thus, with the help from model slicing, the complex task of detecting unintended feature interactions in a huge SPL model is decomposed into many smaller tasks, where

each task focuses on detecting unintended feature interactions between one feature and the others.

1.2.2.2 Compositional Verification of SPLs

To ensure safe operation of all system variants in an SPL, there is a need to provide effective and scalable means of identifying and managing feature interactions. Formal method techniques like model checking have been proposed to detect feature interactions in an SPL. In the past, several strategies for SPL analysis have been proposed, in particular, family-based, product-based and feature-based strategies [10]. A family-based strategy exploits the commonalities among products in an SPL and delivers sound and complete analysis results; but it is often computationally infeasible as the whole SPL is too huge to be analyzed all at once. A product-based strategy performs model checking on individual products in an SPL; although the product-based models are relatively small, there are exponential number of them to be analyzed and thus this approach faces severe scalability problems. A feature-based strategy analyzes a feature in isolation, ignoring other features; it is fast because feature modules are relatively small and there are linear number of them to be analyzed, but it yields incomplete analysis result.

The compositional verification is proposed as an improved analysis strategy from the feature-based approach. It verifies each feature with its smaller, relevant ROS. In each iteration of analysis, a feature (i.e., the FOI) and its smaller ROS are fed into a model checker to check whether a set of safety properties related to the FOI is maintained in all executions. The benefits of this compositional verification are evident: there are only a linear number of models to check (with respect to the number of features), the checks can be performed in parallel, and the analysis is complete with respect to feature interactions because analysis of each FOI includes the relevant portion of the ROS.

1.3 Thesis Overview

Generally, a feature-oriented requirements model should model each feature's requirements as a separate feature module [11]. There are many standard software-engineering modeling languages that might be suitable to declare a feature module. Among them, state-based models (SBMs) are suitable languages to express the behavior of a feature. In feature-oriented requirements modeling language (FORML), which is the SPL modeling language that we have chosen to perform slicing, the syntax of its feature modules is based on UML state machines, and therefore many existing slicing practices on SBM can be applied on it.

This thesis proposes a workflow that aims to slice the feature modules in FORML with respect to one feature module at a time. In this workflow, each feature module in the FORML is treated as a complete SBM, which is called a feature-oriented state machine (FOSM). We implement this workflow in a slicing tool, which is called **FormlSlicer** because it is slicing on FORML.

The workflow consists of two tasks: the preprocessing task and the slicing task. In the preprocessing task, the whole original model is parsed and transformed into a set of control flow graphs(CFGs) so that dependence relationships can be computed among the nodes in the CFGs easily. The slicing task employs multiple slicing processes on the model; each process considers one of the FOSMs as the FOI—which is the slicing criterion—and the rest of FOSMs as the ROS. Each slicing process constructs a sliced model which preserves the portion of the original model that interact with the FOI. The construction process is called Multi-Stage Model Slicing Process; it firstly identifies an initial set of transitions in the ROS that affect the FOI, then finds more states and transitions in the transitive closure of dependences and eventually enriches the model elements in the slice set to create a well-formed, executable model that preserves the reachability properties of the original model.

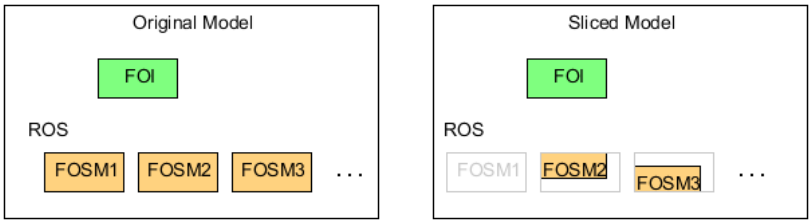


Figure 1.1: A Comparison of the SPL Model before and after Slicing by FormlSlicer

Figure 1.1 shows a side-to-side comparison of the SPL model before and after slicing is performed by FormlSlicer in one process of the slicing task. The FOI remains unchanged in the sliced model. The portions of ROS that are irrelevant to the FOI are removed; this results in some FOSMs in the ROS that are completely removed and some FOSMs partially removed.

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,
- and a slicing task, that gradually adds model elements from the original model into the sliced model and enriches it in the end;

in order to produce a well-formed sliced model that

- simulates the original model,
- achieves a satisfactory 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.

Chapter 2

Related Work

2.1 Program Slicing

To understand model slicing, it is better 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*.

Figure 2.1a [2] is an example program to show the effect of program slicing. It computes both the sum and the product of the first n positive numbers. Figure 2.1b shows its program slice with respect to a slicing criterion of (12,product). We can see that all computations that are irrelevant to the value of variable `product` at Line 12 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.

```
1  read(n);
2  i := 1;
3  sum := 0;
4  product := 1;
5  while i <= n do
6  begin
7    sum := sum + i;
8    product := product * i;
9    i := i + 1
10 end;
11 write(sum);
12 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. (12,product)

Figure 2.1: An Example Program

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. 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 [14]:

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 12 appears before Line 4, the value of `product` written will be incorrect at Line 12. Therefore we know that the statement at Line 12 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 8 is dependent on the predicate at Line 5 since the condition in the while loop at Line 5 determines whether the statement at Line 8 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., EFSM, state charts, 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 [15].

The basic definition of an SBM comes from Mealy machine [16]. 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 [15]. 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 identified 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) [15]. 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.

In this thesis, we have chosen a modeling language—FORML—which feature module is an advanced SBM that encompasses all the above-mentioned advanced constructs. More details will be elaborated in later chapters.

2.2.2 Challenges of SBM Slicing Compared to Program Slicing

It may seem easy to directly apply the techniques from program slicing to SBM slicing. But anyone who attempts to do so will find it a naive approach. There are several differences of SBM slicing compared to program slicing, each posing a non-trivial challenge in designing correct and useful SBM slicing techniques.

The first difference is the level of granularity between a program and an SBM [3]. Program slicing often operates at a line of code, which is a natural level of granularity. Program slices are thus typically subsets of the lines of code in the original program. However, the level of granularity in an SBM is unclear: an individual node may represent

the equivalent of several lines of code, or several nodes may represent the equivalent of a single line of code. Because of this difference of granularity, it is very hard to automate the translation of an SBM into a program.

The second difference is the well-formedness property between a program and an SBM. In program slicing, after removing some lines of code from the original model, 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. The well-formedness of an SBM is formally defined slightly differently depending on statistical constructs and related semantics of the different state machine modeling approaches [17]; but in this thesis, we will simply use an informal notion that a well-formed SBM slice is a standalone SBM itself. A well-formed SBM must not have orphaned transitions or states; in other words, all its states must be reachable from the default initial state.

An SBM slice must be "rewired" to prevent nodes being orphaned [3]. Among the model slicing techniques we have surveyed, researchers adopt different approaches to maintain the well-formedness of an SBM slice. One approach is to completely avoid removing transitions in an SBM and only replace the triggers, guards and actions with some dummy values (e.g., a "NONE" value) [18]; in this way, the sliced SBM retains the same structure as the original SBM and therefore does not risk becoming not well-formed. Another approach is to start with an SBM slice as same as the original SBM and delete only transitions that are self-looping or unreachable [5]; in this way, at each step of slicing, the sliced SBM is always well-formed. One more approach is to add an "enrichment" step as a post-processing step of a normal slicing step, so as to potentially further enrich the resulting slice with model elements in order to satisfy the well-formedness properties [17]. This thesis will follow the last approach.

There are many other differences 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". As a summary, SBM slicing is still in the early stages and there are still many challenges yet to be tackled.

2.2.3 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 dependences to compute slices.

DD is the most commonly mentioned dependence in the model slicing works. DD in an EFSM (one type of SBM languages) has been defined by Korel et al. [5]: DD captures the 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* [5]. We will use this definition in this thesis.

CD in an SBM is a lot more complicated. It will be elaborated separately in Subsection 2.2.3.1.

Besides DD and CD, researchers have used a large variety of different dependences in their model slicers [17, 18]. 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.

2.2.3.1 Control Dependence

In program slicing, control dependence is traditionally defined in terms of a postdominance relation in a CFG [14, 19]. 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 [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.

Although this definition of control dependence is widely used in program slicing, there are some other slightly different definitions as well. Podgurski and Clarke [20] 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 postdominates n_l, another successor of n_i.

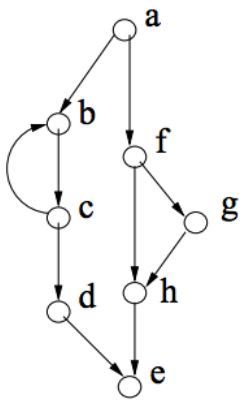


Figure 2.2: A Simple CFG with a Single End Node e

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 dependence [20]. In Figure 2.2, d is weakly control dependent on c but not strongly control dependent on c.

[1]The immediate postdominator of n_i is the postdominator of n_i that doesn't postdominate any other postdominators of n_i.

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 the CFG, all of these definitions of control dependence assume that the single end node property in CFG is satisfied.

The traditional definition on control dependence has been applied in model slicing. Korel et al. [5] presents the definition for control dependence between transitions in terms of the concept of postdominance: a state Z postdominates another state Y iff Z is on every path from Y to the exit state; Z postdominates a transition T, which is an outgoing transition of Y, iff Z is on every path from Y to the exit state through T. Based on the concept of postdominance, control dependence is defined as:

Definition (Control Dependence [5])**.** *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.*

This definition captures the traditional notion of control dependence. However, as the traditional notion is only applicable to program CFGs with single end node, this definition is also limited to SBMs with single end node.

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 (non-termination sensitive control dependence (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,*

1. *for all maximal paths from n_k, n_j always occurs and it occurs before any occurrence of n_i;*
2. *there exists a maximal path from n_l on which either n_j does not occur, or n_j is strictly preceded by 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 [20] 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 [22]. 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 [22].

- 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 [23].

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 is determining 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 follow Ranganath et al.'s algorithm of computing NTSCD [21] with some modifications 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]

2.2.4 Relevant SBM Slicing Techniques

One of the main tasks for this thesis project is to implement a model slicer. Therefore our literature survey focuses more on implementations. In Ojala's implementation document for a slicer of UML State Machine [18], a CFG is constructed to capture all the possible executions of the UML state machines: there are different types of CFG nodes, including BRANCH, SIMPLE and SEND. We observe that the conversion from model to CFG makes it easy for the slicer to directly use the notion of NTSCD defined by Ranganath et al. [21].

Many literature on SBM slicing present their respective slicing algorithms. In general, there are two approaches.

1. In the first approach, the algorithm incrementally removes model elements. For example, Korel et al.'s slicing algorithm [5] starts with an initial slice which is as same as the original model. Initially, all transitions are marked as *non-contributing* transitions. Then the algorithm repeatedly marks some transitions as *contributing* transitions based on pre-computed dependences. At the end, only unreachable transitions or self-looping, *non-contributing* transitions are deleted from the slice.

2. In the second approach, the slicing algorithm incrementally adds model elements. For example, Kamischke et al.'s slicing algorithm [17] starts with an initial slice which solely contains the model elements of the slicing criterion, and adds more model elements incrementally based on pre-computed dependences; after each addition, a model enrichment step is carried out as a post-processing step to satisfy the well-formedness properties of the slice. At the end of all iterations, one more step is performed to remove unreachable states in the slice.

In Section 2.2.3, we have also shown a wide range of dependences used in many slicing algorithms. They show 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.

Altogether, these literature have given us several ideas about designing our own model slicing algorithm: (1) slicing needs to be performed in an iterative way to incrementally remove or add more model elements into slice (we will choose the second approach in this thesis); (2) there can be more types of dependences besides data and control dependences, if there more complex constructs in the SBMs' semantics; (3) dependences are pre-computed and can be used repeatedly during slicing steps; (4) converting a model into its CFG representation makes the data structure lightweight for dependences computation; (5) an enrichment step can be added as a post-processing step to maintain the well-formedness of the model.

2.2.5 Correctness of SBM Slices

In Section 1.1.2, we mention that correctness is one of the desirable properties of a useful sliced model.

First of all, the basic correctness property has been mentioned in various literature. The first condition for a sliced EFSM, as listed by Korel et al. [5] is as follows:

> "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."

In Amtoft et al.'s research note on SBM slicing correctness [4], the same condition is phrased 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."

In other words, the basic correctness property is that the sliced model can simulate the original model with respect to the slicing criterion. Chapter 5 will prove this basic correctness property on the model slicer we have developed.

However, many have recognized that such a correctness property is not sufficient. If a slice only satisfies the basic correctness property, it may be "over-minimized" making it useless, if not misleading, for comprehension purposes [5]. Korel et al. has illustrated an example of such an "over-minimized" slice: the SBM becomes a single state such that all relevant transitions become self-looping transitions on this state. Certainly, this sliced model can satisfy the basic correctness property, because there definitely exists an execution in the sliced model that can simulate an execution in the original model. However, this sliced model is not useful.

Because of that, researchers have imposed a second condition on the sliced model; that means the correctness property will become "stronger" if both conditions are satisfied. The second condition is to ensure that the original model can simulate the sliced model. Korel et al. refers to this second condition as the *traversability property* and proposed two state merging rules that satisfy this condition.

This raises another problem. In order to satisfy this "stronger" correctness property, the sliced model needs to be as close as possible to the original model. In the worst case, slicing does not remove anything from the original model (or adds the whole original model to the slice in the approach of incremental addition). If the sliced model is as same as the original model, it will be correct but not useful. We still want to reduce the model size as much as possible. This prompts us to separate the two properties of precision and reduction.

Precision property corresponds to the second condition, with some differences: a sliced model does or does not satisfy the second condition (i.e., a yes-or-no answer), but can be precise to a different degree. In this thesis, we will use the two state merging rules proposed by Korel et al. which satisfy both the first and second conditions of the "stronger" correctness property. But in all other cases, we will favor reduction over precision.

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 [11]. Section 3.3 discusses the scope of FormlSlicer and explains how FormlSlicer treats feature modules in FORML.

3.1 Terminology

Software Product Line (SPL) A software product line is a family of software systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission [24]. A common approach to SPL requirements and development advocates the use of features as the primary criterion to identify separate concerns. In this approach, an SPL is considered as a finite set of features and a *product* in the SPL is obtained by selecting a subset of these features.

System and Environment In this thesis, we use "system" to refer to the collection of features in an SPL and "environment" to refer to any external factors that can influence the system. This divides the variables in an SPL model into two groups. The first group is the set of environment-controlled variables, i.e., their values can only be changed by the external environment. As an example, consider the actual speed of a car; its 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, they can only monitor it. The second group is the set of system-controlled variables, i.e., their values can be influenced by one or more features. For example, the acceleration of a car is a system-controlled variable.

3.2 What Is FORML

FORML is a modeling language designed to specify feature-oriented models of an SPL's requirements. It is based on the paradigm of FOSD and accompanied by modeling-language constructs for explicitly expressing intended feature interactions [25]. A model expressed in this language consists of two views [11]:

- A world model is a description of the world comprising products of an SPL and the environment in which the products 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 an SPL's products. 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 [11]:

- 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 composite state contains one or more *orthogonal regions* (*regions* for short), where each region holds a concurrent sub-machine. 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*. The state and region form a hierarchical containment relation (i.e., each composite state contains one or more regions; each region contains a sub-machine). A black solid circle, called *pseudo state*, points to the default initial state of a sub-machine. For example, *S1*, *S3* and *S5* in Figure 3.1 are default initial states in their respective machines.

Throughout the thesis, we use the following definitions adapted from Shaker's PhD thesis [11] to access information about state hierarchy in a state machine:

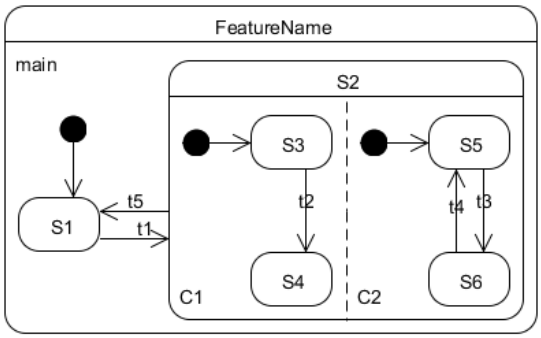


Figure 3.1: The Structure of a State Machine in FORML's Behavior Model

> "The root of the state hierarchy is a composite state that represents the 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."

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) a new message object is generated; (2) a set of existing objects are destroyed; and (3) an object's attribute or a global variable is set 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 [26]. 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 [26]—a modeling language used by the NuSMV model checker [27]. It is beneficial to the community in further extending this tool set by creating a model slicer on FORML.

3.3 Scope of FormlSlicer

In Section 2.2.2, we discuss the challenges of slicing on SBMs. Because the behavior model of FORML is a highly complex type of SBMs, FormlSlicer faces many challenges. We have solved some of the challenges in this thesis project; but there are still a few limitations, as described in this section. Chapter 7 will present possible extensions of this work to overcome these limitations.

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. We can simply identify those class fields that are not in the set of relevant variables and slice them away from the world model; if all fields in a class are sliced away, we slice away the class as well. This can be done either manually, or by writing a small program to automate it.

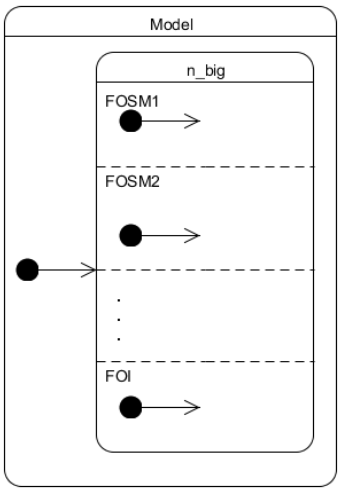


Figure 3.2: An Example Model

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 format of a state machine.

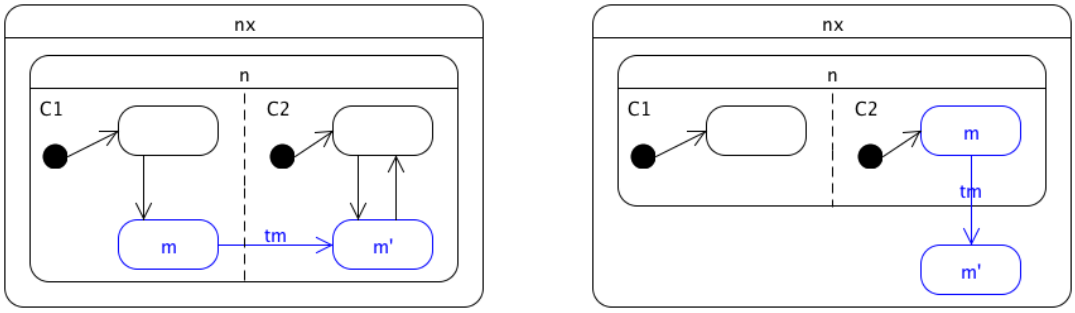
FormlSlicer treats the whole SPL model as one big state machine. This big state machine comprises a composite state containing many orthogonal regions. Each orthogonal region contains one sub-machine, which is an FOSM. Figure 3.2 shows an example of such a big state machine. We can see that the model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

Normally, an SPL model contains *presence conditions* in some of its transition labels. A presence condition is introduced for each optional feature; it is a boolean variable that is named after its corresponding feature. Presence conditions make each requirement conditional on whether the feature that introduced the requirement is present in the product. However, the model used by FormlSlicer does not contain presence conditions and therefore it represents a product with all the features, which is not necessarily a valid product as there are mutually exclusive features. Strictly speaking, this is more like a 150% model [28] than an SPL model. A 150% model contains the behavior of all features, no matter if those features exclude each other within a configuration of the SPL. A 150% model can be converted to an SPL model by adding the presence conditions in conjunction to the transitions' guard conditions. This will be left for future work.

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

FormlSlicer imposes two 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 state within an orthogonal region that has a sibling orthogonal region can have an outgoing transition that exits the 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.

Chapter 4

A Full Picture of FormlSlicer's Internals

This chapter explains the design of FormlSlicer in the order of its workflow.

4.1 Overview of FormlSlicer's Workflow

Figure 4.1 shows the big picture of FormlSlicer's workflow. It consists of two major tasks: the preprocessing task and the slicing task[1].

The original model, which consists of multiple FOSMs, is input to the preprocessing task. The task converts this model into a group of CFGs. Based on these CFGs, the task computes three types of dependences and stores the generated results in different tables.

Next, FormlSlicer forks off multiple slicing processes to perform the slicing task. All the slicing processes are sharing the same resources on dependences and CFGs generated from the preprocessing task. Each slicing process considers a different FOSM as its FOI and treats the rest of FOSMs as the ROS. Then it goes through a multi-stage model slicing process to slice the FOSMs in ROS with respect to the selected FOI. Eventually, it outputs a sliced model.

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

The different slicing processes are doing their slicing jobs independently, although they are reading the same inputs. The use of concurrent threads in FormlSlicer greatly saves the time that will otherwise be spent on slicing the big model linearly.

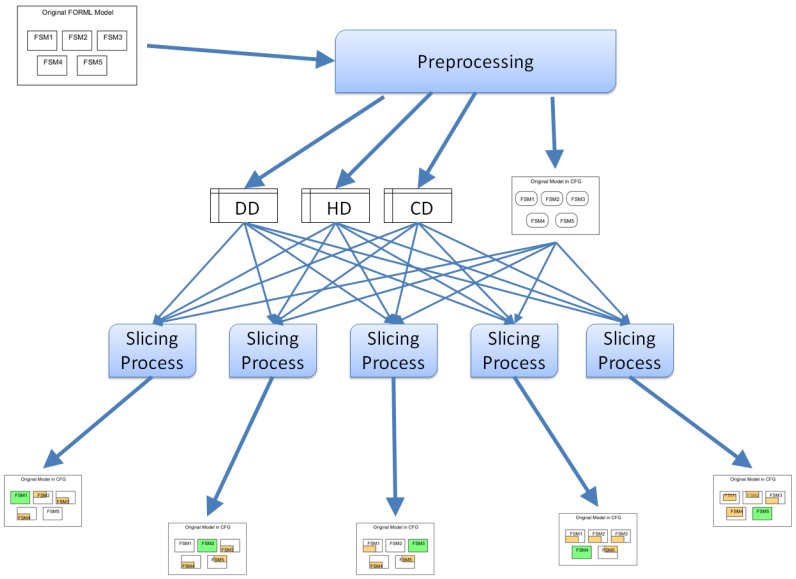


Figure 4.1: Overview of FormlSlicer's Workflow: the Preprocessing Task and the Slicing Task

The next three sections explore further into the internals of the preprocessing task and the slicing task. Section 4.2 and Section 4.3 presents the preprocessing task. It starts with a parser and a converter from the input model to CFG, then presents three types of dependences, including hierarchy dependence (HD), data dependence (DD) and control dependence (CD). Section 4.4 presents what a single slicing process has undergone—the Multi-Stage Model Slicing process.

This is a list of the sub-tasks performed in 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. Replacing Cross-Hierarchy Transitions
      iii. Transition-to-State Step
      iv. CD-HD Step

   (c) Model Enrichment Stage

      i. Step I: State Merging Step
      ii. Step II: True Transitions Creation

4.2 Preprocessing: Model Parsing and Conversion from FORML to CFG

FormlSlicer is equipped with a simple parser to read an input model. Each FOSM in the input model must be written in the format as shown in Table 4.1 and placed in the same folder. We need to declare all the components inside an FOSM, including the states, the regions, the transitions and the macros.

A simple input example is shown below to declare an FOSM from Figure 3.1.

Feature.txt

```
feature FeatureName
region main FeatureName
state S1 FeatureName.main true false
state S2 FeatureName.main false true
transition t1 t1:E1+(o)/a1:x:=1; FeatureName.main.S1 FeatureName.main.S2
transition t5 t5:E1-(o)/a1:x:=0; FeatureName.main.S2 FeatureName.main.S1
region C1 FeatureName.main.S2
region C2 FeatureName.main.S2
state S3 FeatureName.main.S2.C1 true false
state S4 FeatureName.main.S2.C1 false false
```

```
transition t2 t2:[inState(FeatureName.main.S2.C2.S6)]/ FeatureName.main.S2.C1.S3 FeatureName.main.S2.C1.S4
state S5 FeatureName.main.S2.C2 true false
state S6 FeatureName.main.S2.C2 false false
transition t3 t3:E2+(o)/ FeatureName.main.S2.C2.S5 FeatureName.main.S2.C2.S6
transition t4 t4:[a=='val']/ FeatureName.main.S2.C2.S6 FeatureName.main.S2.C2.S5
```

Item to be Declared	Declaration Format
Feature	feature ⟨feature name⟩
Macro	macro ⟨input sequence⟩⟨replacement output sequence⟩
State	state ⟨state name⟩⟨parent region⟩⟨first state in region?⟩⟨composite state?⟩
Transition	transition ⟨transition name⟩⟨expression⟩⟨source state⟩⟨destination state⟩

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

4.2.1 Control Flow Graph

Control flow graph (CFG) is a directed graph, usually seen in compiler analysis to represent all execution paths through a program during its execution [29]. This representation is also useful in model slicing. As introduced in Chapter 2, some model slicers [18] convert a model into CFGs before slicing. The main benefit of converting an input model into CFGs is to simplify the representation of the model so that it becomes easy for further processing, particularly for the dependences computations. FormlSlicer will do the same conversion.

A CFG only consists of nodes and edges. As shown in Figure 4.2, there are two types of nodes—TNode and SNode–which all belong to the generic type—Node. Each Node has an ID, a set of outgoing nodes, and a set of incoming nodes.

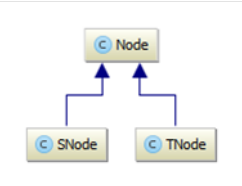


Figure 4.2: The Class Hierarchy of Node, TNode and SNode in CFG

TNode

FormlSlicer converts each transition into a TNode. A TNode contains the set of monitored variables and controlled variables of its corresponding transition. Section 4.2.2 describes how a transition is converted into a TNode. The set of incoming nodes and the set of outgoing nodes of a transition are both singleton; they are IDs of the source and destination state of the transition, respectively.

SNode

FormlSlicer converts each state into an SNode. SNode is simpler than TNode. It does not contain more information other than the generic information (its own ID and sets of outgoing and incoming nodes' IDs) inherited from Node.

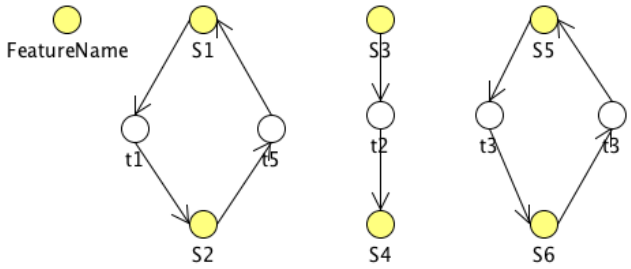


Figure 4.3: Several CFGs that are Converted from the FORML FOSM. Yellow Nodes are SNodes.

After the conversion, each FOSM in the input model becomes one or more CFGs. The FOSM example in Figure 3.1 becomes the CFGs in Figure 4.3. We can see that there is one distinctly connected CFG for each sub-machine in this example. But if there is a transition that crosses the hierarchy border, it will be converted into a TNode that connects two distinct CFGs together. When the CFGs for the input model are ready, it is time to move on to the next preprocessing step—dependences computation.

4.2.2 Converting a Transition into a TNode

We can recall from Section 2.2.3 that data dependence in an SBM captures the notion that one transition defines a value to a variable and another transition may potentially use this value. Similarly, the transitions in the FORML state machines have data dependence among themselves. When FormlSlicer converts a transition into a TNode, the information needed to detect data dependence must be preserved.

Section 3.2 describes that a transition label in FORML may contains WCE, guard and WCA. The WCA performed by a 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))*). It is sufficient to match actions and trigger events involving similar message type; in the above example, we can see that there is a dependence relationship between t_1 and t_2 and we need to match t_1 and t_2 with respect to the same type of message.

To facilitate detection of data dependence among transitions, we need to simplify WCE, guard and WCA expressions in transitions to include just the information needed. FormlSlicer simplifies them into two groups of variables:

- *Monitored Variables*, which represent phenomena that are sensed by or inputs to the system;
- *Controlled Variables*, which represent phenomena that are controlled or affected by system outputs [30].

In the previous example, FormlSlicer makes "*+C*" to become t_1's controlled variable and t_2's monitored variable, so that it is easy to match t_1 and t_2 automatically using the computation algorithm of data dependence.

In Table 4.2, the first two columns show the types and formats of WCE in a FORML transition. The types include: 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 how FormlSlicer transforms each of them into a string value that represents a monitored variable.

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 WCE

Table 4.3 shows how FormlSlicer transforms different types of WCA. 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 how FormlSlicer transforms each of them into a string value representing a controlled variable. Note that in an assignment action, the arguments of the function need to be

[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 are going to over-approximate the influence of this action by saying that all the objects belonging to that category of message are destroyed.

identified as monitored variables, because the assignment action of a transition use the values of the function's arguments that may be defined in the assignment action of another transition. 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 WCA

The guard of a transition use variables that may be defined in the 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 dependence between t_1 and t_2 with respect to the variable *status*. To detect data dependence, FormlSlicer just needs to know what variables are involved in the guard expression (e.g., the variable *status*); it does not concern the guard expression itself. Based on this idea, FormlSlicer simplifies a guard into one or more monitored variables.

The 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 only true 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 Guard

As a result, when FormlSlicer converts the transition into a TNode, the WCE, guard and WCA of a transition in FORML are transformed into monitored variables and controlled variables that are stored in the TNode.

4.3 Preprocessing: Dependences Computation

FormlSlicer starts from an empty slice set and gradually adds model components into it. The dependences among nodes (including SNodes and TNodes) are very important information for FormlSlicer to determine which nodes should be selected into the slice set.

In the previous section, we have discussed that the FOSMs in the input model have been converted into CFGs. CFG is a lightweight graph structure suitable for the dependence computation, because dependence computation concerns only the connectivity relationship among nodes. This section introduces three types of dependences that are computed by FormlSlicer.

4.3.1 Hierarchy Dependence

The states in a FORML model are organized in a hierarchy. A state may be a composite state which contains one or multiple regions; each region contains a sub-machine which consists of more states, called the *child states* of the composite state. On the other hand, the composite state is called the *parent state* of the *child states*. As an example, Figure 4.4 shows such a state hierarchy; state $n1$ resides within $region1$ of state $n2$ and thus $n1$ is $n2$'s child state and $n2$ is $n1$'s parent state. The first state immediately after the pseudo state in the region is called the *default initial child state*[3] of its parent state.

Hierarchy dependence (HD) is a dependence to reflect such a state hierarchy relationship among nodes. We say that $n1$ is **hierarchy dependent** on $n2$, denoted as $n1 \xrightarrow{hd} n2$.

FormlSlicer computes the HD while performing the CFG conversion. It keeps two tables to record the results:

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

HDtable2 a 2D-table mapping each SNode to its default initial child states[4]

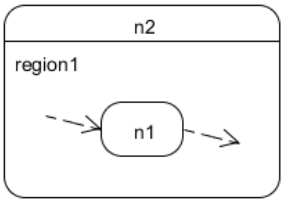


Figure 4.4: Hierarchy Dependence

[3]The arrow from a pseudo state to its default initial child state (i.e., the state pointed by the arrow from the pseudo state) cannot be labeled. If user needs a transition from the pseudo state to its default initial state, user can make the original pseudo state as a new default initial state and create a new pseudo state to point to it.

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

4.3.2 Data Dependence

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

TNode A can be data dependent on TNode B with respect to a certain variable, if TNode A monitors the variable and TNode B controls the variable, and there are no other TNodes between TNode A and TNode B which interfere the controlling effect of TNode B.

To put it formally, we say t_k is **data dependent** on t_1 with respect to v, denoted as $t_k \xrightarrow{dd}_v t_1$, iff there exists a variable $v \in mv(t_k) \cap cv(t_1)$ and there exists a path $[t_1 \cdots t_k](k \geq 1)$ such that $v \notin cv(t_j)$ for all $1 < j < k$. Here, t represents a TNode in a CFG produced by FormlSlicer, v represents a system-controlled variable, $mv(t)$ and $cv(t)$ are monitored and controlled variables contained in t respectively.

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 4.5 has illustrated this concept.

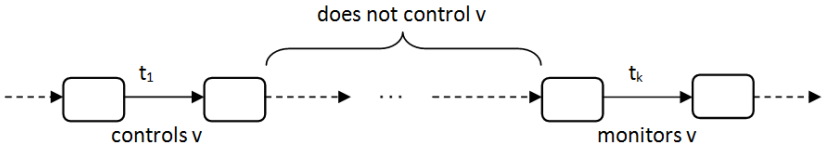


Figure 4.5: Data Dependence between t_k and t_1 w.r.t. v

DD is the most important dependence among the three. The concepts of monitored and controlled variables are specifically created for it.

InState Dependence In a FORML transition label, there is a special type of guard condition called "InState". In the example shown in Figure 4.6, transition T2 contains the guard condition of "InState(DummyState)", then T2 will be triggered when the current state configuration of the model contains the state of *DummyState*. For consistency, FormlSlicer treats InState Dependence as a sub-type of Data Dependence. In this example, FormlSlicer considers T2 as data dependent on all the incoming transitions to *DummyState*, e.g., T1, with respect to the variable "DummyState". FormlSlicer will stores this information as a data dependence entry in the computation results of DD.

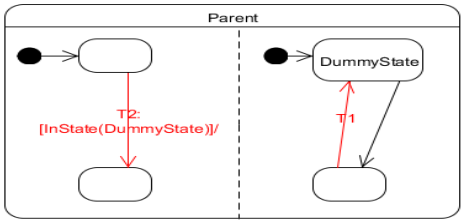


Figure 4.6: InState Dependence as a Variation of Data Dependence

Algorithm 4.1 shows how FormlSlicer computes data dependence. Compared to existing algorithms, ours is dealing with a more complex model with state hierarchy structure. The computation results are stored in a table:

DDtable a 1D-table mapping each variable to a pair of TNodes such that the left one is dependent on the right one

Algorithm 4.1 traces from a TNode with a controlled variable v to another reachable TNode which monitors v, and thereby establishes data dependence between the two TNodes. In each loop of the *foreach* statement at Line 1, the algorithm checks whether a node has controlled variables. If there are some controlled variables, it loops for each variable at Line 4 to search for nodes that are data dependent with respect to this particular variable. The algorithm uses a queue *qt* to perform the searching. At each iteration of the *while* loop at Line 8, it polls one node from *qt* and checks whether that node is monitoring the variable. If so, data dependence is established between the starting node and the polled node, as shown at Line 21.

The *foreach* statement at Line 24 is checking whether *currNode* is controlling the variable v. If so, it means that the definition-clear path between *node1* and any other nodes reachable from *node1* beyond *currNode* is cut off, and so there is no need to continue the search beyond *currNode*.

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 := [the outgoing node of node1]  // Initialize qt
 7         set dependentOn := an empty pair consisting of two Nodes
 8         while qt is not empty do
 9             set currNode := qt.poll();
10             if currNode == node1 then continue;
11             if currNode instanceof SNode then
12                 if visitedSNodes contains currNode then continue;
13                 if HDtable2 contains currNode then
14                     ADD the all the default initial child SNodes of currNode to qt
15                 end
16                 ADD all the outgoing nodes of currNode to qt
17                 ADD currNode to visitedSNodes
18             else
19                 if dd contains v→(currNode,node1) then continue;
20                 if currNode.monitoredVars contains v then
21                     ADD (currNode,node1) to dependentOn;
22                 end
23                 set isContVRset := false
24                 foreach c in currNode.monitoredVars do
25                     if c == v then
26                         set isContVReset := true;
27                         break;
28                     end
29                 end
30                 if isContVRset := false then  ADD the outgoing node of currNode to qt
31             end
32         end
33         if DDtable contains Key v then  MERGE dependentOn to DDtable[v]
34         else  DDtable[v] := dependentOn;
35     end
36 end
```

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 on whether one node can decide the execution of another node. Generally, a branching node[5] is a decision point on whether the nodes along one of its branch can be passed through, and thus the branching node decides the execution of those nodes along one of its branch.

Without using control dependence, the sliced model will lose the useful information on how a path branches to reach an important node, and therefore become more imprecise. By using control dependence, if a node is included in the sliced model, the other node which acts as its decision point will be included in the sliced model too. In this way, the decision point serves like an "anchor" for the path flowing through it. The control flow structure is therefore preserved in the sliced model, and model comprehension is facilitated in the resultant slice.

In the scenario when the default initial state is a branching state, using control dependence to add the default initial state into the sliced model is crucial. In FORML, the pseudo state points to exactly one default initial state; it is incorrect to have multiple default initial states within one sub-machine. If a default initial state branches and it is not selected into the sliced model, we will run into trouble in determining the new default initial state in the sliced model. This issue will become more evident when FormlSlicer reaches the last step of model slicing process, which will be elaborated in Section 4.4.3.2.

Therefore, control dependence is an important dependence in FormlSlicer's slicing task.

As elaborated in Section 2.2.3.1, researchers have proposed many different definitions on control dependence in both program slicing and model slicing. At present, there is no standard, 100% correct algorithm to compute control dependence for slicing on SBMs.

FormlSlicer simply adopts one of these definitions—the NTSCD as defined by Ranganath et al. [21]. A precise definition of control dependence is presented as below:

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,*

1. *for all maximal paths from n_k, n_j always occurs and it occurs before any occurrence of n_i;*

[5] A branching node in CFG has more than one outgoing paths, i.e., its outdegree is greater than 1. It is always an SNode, which is equivalent to a state in FORML with multiple outgoing transitions.

2. *there exists a maximal path from n_l on which either n_j does not occur, or n_j is strictly preceded by n_i.*

In other words, when there are at least two branches of paths from n_i, we can definitely pass through n_j by taking one of the branch, and can possibly avoid n_j by taking another branch. When this is the situation, n_j is control dependent on n_i. This is illustrated in Figure 4.7, the key idea behind this definition is that reaching again a start node is analogous to reaching the end of path.

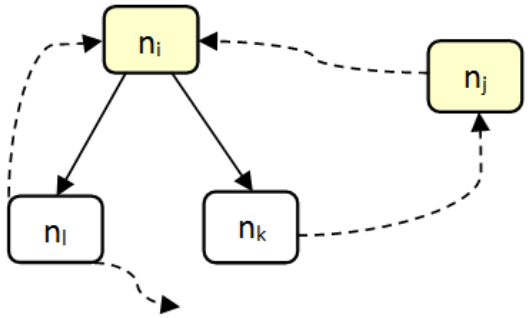


Figure 4.7: An Illustration of Non-termination Sensitive Control Dependence: n_j is control dependent on n_i

4.3.3.2 CD Algorithm: Paths Representation for a Node

Ranganath et al. [21] has presented a dynamic algorithm of computing NTSCD. Its main idea is to start from each branching node and search for any other nodes along the path that can be control dependent on that branching node. The algorithm represents sets of CFG paths symbolically and propagates these symbolic values to collect the effects of particular control flow choices at program points 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[6].

FormlSlicer computes control dependency by firstly finding CFG paths from each branching node to all other nodes, and then for all possible paths from node n_i to node n_j, performs

[6]There are a few reasons why we do not directly use Ranganath et al.'s algorithm: (1) The paper only shows the skeleton of the algorithm and does not explain certain details, such as why cardinality of sets of paths and branching node's outdegree can be used in determining control dependence; (2) the CFG used in their algorithm is slightly different from ours. However, the main idea remains the same.

a special operation called *reduction* to check whether n_i cannot avoid passing through n_j, so as to determine the control dependence between n_j and n_i.

We use an array p, which is an array to record the paths for each node along the traversal. The paths for node i, as stored in p[i], represent the different paths that can be traversed from a specific branching node to node i.

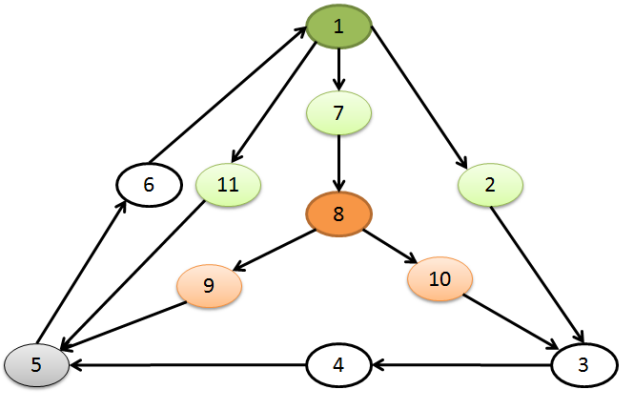


Figure 4.8: A CFG Example to Illustrate the Paths Representation of Node 5 from Node 1;
Node 8 has Two Outgoing Nodes (9 and 10), Highlighted in Orange Color;
Node 1 has Three Outgoing Nodes (2, 7 and 10), Highlighted in Green Color

The paths representation of a node can be explained in Figure 4.8. In this CFG, we want to represent the different paths from Node 1 to Node 5; Node 1 is the branching node. We observe that there are four different possible paths to reach Node 5 from Node 1:

1. the Path of "1→11→5";
2. the Path of "1→7→8→9→5";
3. the Path of "1→7→8→10→3→4→5";
4. the Path of "1→2→3→4→5";

The paths representation of Node 5 from Node 1 is shown in Figure 4.9. The paths are separated by semicolon. Each path consists of one or more sub-paths. Each subpath consists of one source index and one target index.

Path 1 is represented as "1:11". Along the traversal from Node 1 to Node 5, there is only one decision point at Node 1 and the path has chosen the branch of Node 11 from Node 1.

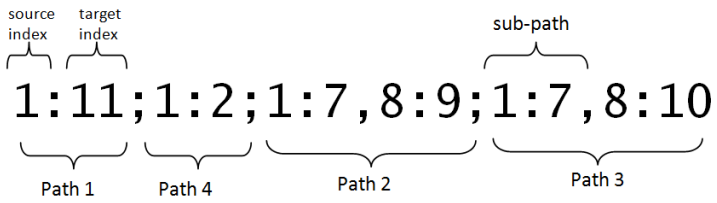


Figure 4.9: The Representation of Paths of Node 5 from Branching Node 1 in the CFG in Figure 4.8

Path 2 is represented as "1:7,8:9". There are two sub-paths here: "1:7" and "8:9". This means that along the traversal from Node 1 to Node 5, there are two branching nodes. The first branching node is Node 1 and the path takes the branch of Node 7. The second branching node is Node 8 and the path takes the branch of Node 9 from Node 8.

Path 3 is represented 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 takes the branch of Node 10 from Node 8. Although Node 3 and 4 are along this path, they are neither branching nodes nor the next nodes of any branching nodes; at either of these two nodes, the path has only one possible way to go. Therefore, Node 3 and 4 do not carry useful information in distinguishing the current path from other paths, and we do not record them in the path.

Path 4 is represented as "1:2". At Node 1, the path takes the branch of Node 2.

After collecting all the paths from Node 1 to Node 5. We observe that no matter which path Node 1 takes, it always reaches Node 5. In other words, **Node 1 cannot avoid passing through Node 5**. This does not satisfy the definition of CD. Therefore, Node 5 is not control dependent on Node 1.

4.3.3.3 CD Algorithm: Reduction of the Paths Representation to Detect CD

FormlSlicer can automatically detect the fact that Node 5 is not control dependent on Node 1 by reducing the paths representation. Based on the fact that 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". This implies that at Node 8, no matter we choose the branch of "8:9" or the branch of "8:10", we always reaches Node 5.

Now the representation of paths becomes "1:11;1:2;1:7". Because Node 1 has three outgoing nodes (2, 7 and 11), a second round of reduction takes place and "1:11;1:2;1:7" is reduced to "".

As a result, the paths representation of Node 5 from Node 1 is empty.

After walking through the example, we can generalize the rule for paths reduction. Given that Node i has some outgoing nodes, $1, 2, \ldots, k$, and the sorted paths representation of Node j from Node i contains the substring $Xi:1; Xi:2; \ldots; Xi:k$ (X is the common prefix for paths), we can reduce the paths by replacing the substring to be X.

The paths representations of other nodes from Node 1 cannot be reduced to empty. For example, the paths representation of Node 3 from Node 1 is "1:2;1:7,8:10". Node 3 is said to be control dependent on Node 1, because its paths representation implies that there are some possible ways from Node 1 that can pass through Node 3 and some other possible ways from Node 1 that can avoid Node 3; in other words, Node 1 controls the execution of Node 3.

4.3.3.4 CD Algorithm: Pseudo-code and Explanations

This section presents FormlSlicer's algorithm in computing the control dependence. The main algorithm is shown in Algorithm 4.2. The computation results are stored in a table:

CDset a set consisting of many pairs of nodes (n1, n2) where n2 is control dependent on n1

In Algorithm 4.2, p is an array of String, with each element storing the paths representation of each node in CFG from a specific branching node.

The algorithm examines all the nodes in the *foreach* statement at Line 3 and checks whether the node is a branching node at Line 4. Only branching nodes can possibly control other nodes' execution.

Next, from Line 5 to Line 10, the algorithm initializes the paths representation of the neighbors of the branching node *node1*. These neighbor nodes are added into *workset*.

In each iteration of the *while* loop at Line 11, *currNode* is polled from the *workset* and analyzed in three cases. We reuse the same CFG from Figure 4.8 to explain.

1. If *currNode* is a branching node, as detected at Line 13, then the paths representation of each outgoing node of *currNode* needs to be extended.

   - An example is shown in Figure 4.10a. The paths representation of Node 10 copies the paths representation of Node 8 and appends a subpath "8:10", to show that the path has branched at Node 8. We say that the paths representation of Node 10 is extended from that of Node 8.

Algorithm 4.2: Main Algorithm of Control Dependence Computation

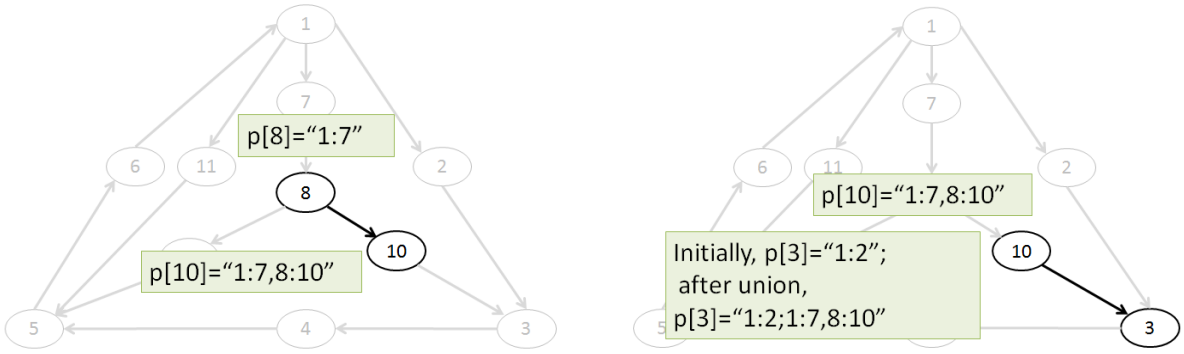
```
   Input: allNodes
   Output: CDset
 1 set CDset := ∅
 2 set p := array[String]
 3 foreach node1 in allNodes do
 4 |   if node1 has 1 outgoing node then continue;
 5 |   set workset := unique queue
 6 |   reset all fields in p to be null
 7 |   foreach node2 (with ID n2idx) in node1.outgoingNodes do
 8 |   |   p[n2idx] := "n1idx:n2idx"
 9 |   |   ADD node2 into workset
10 |   end
11 |   while workset is not empty do
12 |   |   set currNode := workset.poll(); currIndex := currNode.ID;
13 |   |   if currNode.outgoingNodes.size > 1 then
14 |   |   |   foreach n3 (with ID n3idx) in the currNode.outgoingNodes do
15 |   |   |   |   if n3 == node1 then continue;
16 |   |   |   |   p[n3idx] = ExtendPath(currIndex, "currIndex:n3idx");
17 |   |   |   |   if NOT HasReductionOccursBefore (n3idx) AND IsReachableFromPartialPaths (n3idx) then
18 |   |   |   |   |   ADD n3 to workset
19 |   |   |   |   end
20 |   |   |   end
21 |   |   else if currNode.outgoingNodes.size == 1 then
22 |   |   |   set n3 (with ID n3index) := currNode.outgoingNodes[0]
23 |   |   |   if n3 == node1 then continue;
24 |   |   |   if NOT HasReductionOccursBefore (n3index) AND UnionPathHappens(n3index, n2index) then
25 |   |   |   |   ADD n3 to workset
26 |   |   |   end
27 |   |   end
28 |   end
29 |   for j from 0 to last index in p do
30 |   |   if IsReachableFromPartialPaths (j) then  ADD the pair of (j, n1idx) into CDset
31 |   end
32 end
```

- The function *Extend*, as listed in Algorithm C.5, extends the paths representation of each outgoing node of a branching node.

2. If *currNode* is not a branching node, as detected at Line 21, then the paths representation of *currNodes*'s only outgoing node is *union-ed* with that of *currNode*.

   - An example is shown in Figure 4.10b. The 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 *UnionPathHappens*, as listed in Algorithm C.4, unions the paths representation of the outgoing node of a non-branching node.

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



(a) Extend the Path Representation when *currNode* is a Branching Node

(b) Union the Path Representation when *currNode* is a Non-Branching Node

Figure 4.10: Two Cases in Propagating the Paths Representation from *currNode* to its Neighbour in Algorithm 4.2

As mentioned in Section 4.3.3.3, when the paths representation of a node n is not empty, it implies that from the branching node, it is possible to take a path to pass through n and take another path to avoid passing through n. In this case, the branching node controls the execution of n. Function *IsReachableFromPartialPaths* is monitoring this scenario to determine the control dependence relationship between a given node and the branching node.

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. In the table, p[i] refers to the paths representation of the node with an ID of i.

Function Signature	Goal of Function	Algorithm
HasReductionOccursBefore (targetIndex)	It determines whether p[*targetIndex*] has been reduced to empty.	C.1
IsReachableFromPartialPaths (targetIndex)	It determines whether p[*targetIndex*] contains some non-reducible paths, i.e., whether the node with an ID of *targetIndex* is control dependent on the branching node.	C.2

ReducePaths (originalPath)	It performs reduction on the input paths representation.	C.3
SortPaths (paths)	It performs an insertion sort on *paths*, firstly based on units of paths, secondly based on units of sub-paths. Insertion Sort is more efficient because the paths are likely to be in ascending order.	Omitted
UnionPathHappens (targetNodeIndex, prevNodeIndex)	It adds p[*prevNodeIndex*] to p[*targetNodeIndex*]. If nothing is changed, the function returns false. Otherwise, it returns true.	C.4
ExtendPath (srcIndex, newSubPath)	It adds the new subpath to each path in p[*srcIndex*] and returns the new p[*srcIndex*].	C.5

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

4.3.4 Summary

In summary, we have computed all three dependences from the CFGs and obtained the following results:

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

HDtable2 a 2D-table mapping each SNode to its start child state[7]

DDtable a 1D-table mapping each variable to a pair of TNodes such that the left one is dependent on the right one

CDset a set consisting of many pairs of nodes (n1, n2) where n2 is control dependent on n1

[7] A composite state can contain multiple regions and therefore have multiple start child states

These results serve as dictionaries for the slicing processes to look up. They will not be modified after the preprocessing task.

We are now ready to move on to the slicing task. FormlSlicer will fork off n processes for n features, each considers one feature as the FOI. In the next section, we will present the workflow of each slicing process.

4.4 Multi-Stage Model Slicing Process

This section presents the workflow of each slicing process in the slicing task. Because the workflow is divided into several stages, we name it as **Multi-Stage Model Slicing Process**.

FormlSlicer's slicing strategy is to start with an empty sliced model in the ROS. At each step in the Multi-Stage Model Slicing Process, certain model elements (either an SNode or a TNode) in the ROS are put into the sliced model. Along the process, the sliced model grows gradually. The FOI will remain unchanged throughout the process.

Figure 4.11 shows a side-by-side comparison between the original model and the sliced model. Some FOSMs in ROS are entirely absent in the sliced model; some are partially absent.

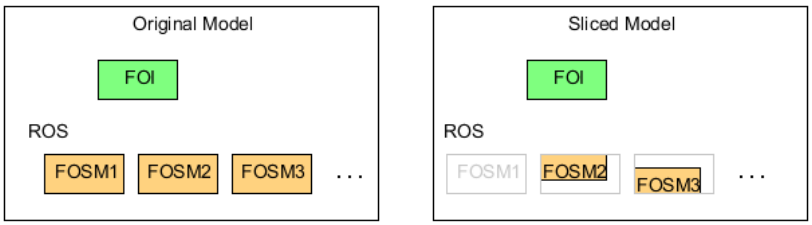


Figure 4.11: A Comparison between Original Model and Sliced Model

Table 4.6 lists the three stages in Multi-Stage Model Slicing Process and explains their distinct purposes.

Stage Name	Purpose of Stage
Initiation Stage	This stage selects the initial set of TNodes into the sliced model.
General Iterative Slicing Stage	This stage uses the dependences computed from the preprocessing task to add more SNodes and TNodes based on the initial sliced model.
Model Enrichment Stage	This stage ensures that all the nodes in the sliced model form a well-formed state machine.

Table 4.6: Distinct Purpose of Each Stage in Multi-Stage Model Slicing Process

We will use a slicing example to illustrate each step of the Multi-Stage Model Slicing

Process. The example model consists of only two FOSMs; one of which is the FOI and the other one is an FOSM in ROS.

The Multi-Stage Model Slicing Process is performed on the CFG structure. Recall from Section 4.2.1 that a state in the input model is converted into an SNode and a transition is converted into a TNode in a CFG, it is easy to match the CFGs and the equivalent FOSM. In the following sections, we will present both the CFGs and the equivalent FOSM side-by-side to show the effects of slicing at each step. Elements that are newly added into the sliced model are highlighted in red color; existing elements in the sliced model are highlighted in black color; 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*; on the other hand, an element is *out-of-slice* if it is not in the sliced model.

4.4.1 Initiation Stage

This stage selects the initial set of TNodes into the sliced model in ROS, based on what the FOI monitors.

4.4.1.1 Variable Extraction Step

Only the monitored variables of the FOI will be used as a slicing criterion to select the initial set of transitions in other FOSMs in the ROS. We do not care the controlled variables of the FOI. This is because **FormlSlicer only concerns how the other FOSMs influence the FOI, but not how the FOI influence the other FOSMs.**

(a) The Control Flow Graph of Feature of Interest **(b)** The FOSM of Feature of Interest

Figure 4.12: A Simple Example of Feature of Interest

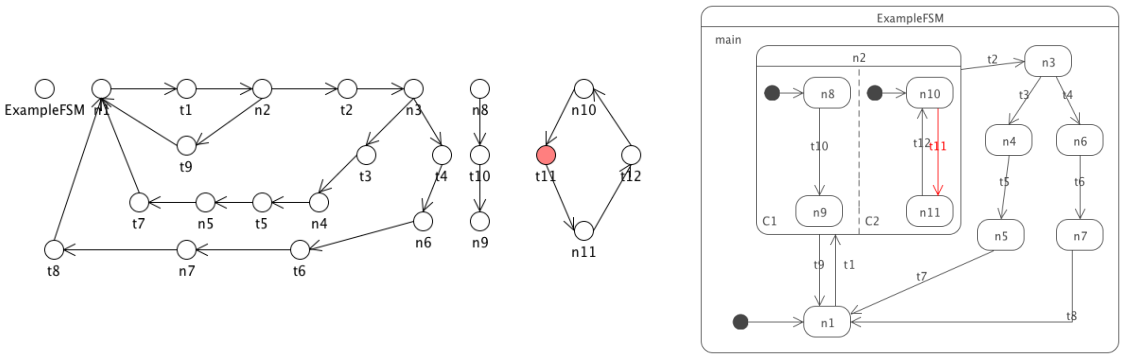
Based on this idea, *Variable Extractor* is a module in FormlSlicer to extract a collection of monitored variables from the FOI.

Figure 4.12 shows a simple example of the FOI. It monitors one variable, $v1$. *Variable Extractor* takes in the CFGs of this FOI (shown in Figure 4.12a) and extracts all 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 this example, $V_{Rv} = [v1]$.

4.4.1.2 Initial Transition Selection Step

Initially, we have an empty sliced model in the ROS. During the Initial Transition Selection Step, FormlSlicer selects all the TNodes in the ROS that control any relevant variable, and add them into the sliced model.

A simple example of an FOSM in the ROS is shown in Figure 4.13.



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

(b) The Example FOSM in the ROS

Figure 4.13: The Example FOSM in the ROS after Initial Transition Selection Step

In our example, the transition *t11* in Figure 4.13b has a label of "t11:[v2==‘val2’] /a1:v1:=‘val’;". According to how FormlSlicer transforms a transition label into monitored and controlled variables as discussed in Section 4.2.2, the TNode of *t11* has one monitored variable $v2$ and one controlled variable $v1$.

Because $v1$ is a relevant variable (collected by *Variable Extractor* in the previous step), *t11* is added into the sliced model. The monitored variables of *t11*, e.g., $v2$, need to be added to the set of relevant variables. Now, $V_{Rv} = [v1, v2]$.

4.4.2 General Iterative Slicing Stage

As elaborated in Section 4.3, we have obtained the dependences from the preprocessing task. The results are stored at *HDtable1*, *HDtable2*, *DDtable* and *CDset*. These are useful information for the General Iterative Slicing Stage in adding more SNodes and TNodes based on the initial sliced model.

The General Iterative Slicing Stage consists of four steps (Figure 4.14):

DD Step
Add more TNodes that are transitively data dependent on the part-of-slice TNodes w.r.t. any relevant variable;

Transition-to-State Step
Add more SNodes based on the part-of-slice TNodes;

Replacing Cross-Hierarchy Transition
Replace all out-of-slice cross-hierarchy transitions with "true" transitions and add them into the sliced model;

CD-HD Step
Add more SNodes that are transitively control dependent or hierarchy dependent on the part-of-slice SNodes.

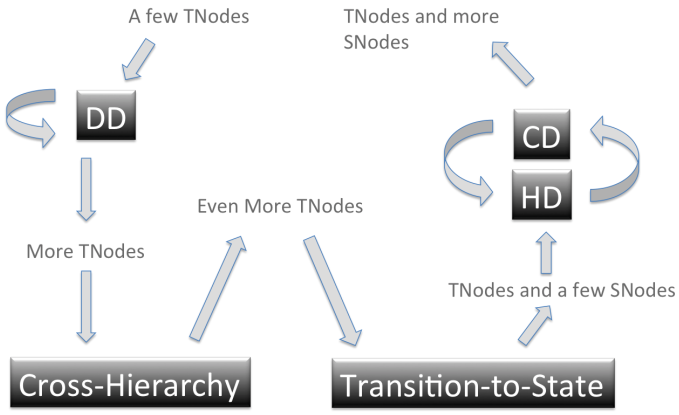


Figure 4.14: Four Steps in General Iterative Slicing Stage

Both the DD Step and CD-HD Step perform the node adding operation **iteratively**, in order to add nodes that are **transitively** dependent on any part-of-slice nodes. This is why the word "iterative" is in the name of this stage.

4.4.2.1 DD Step

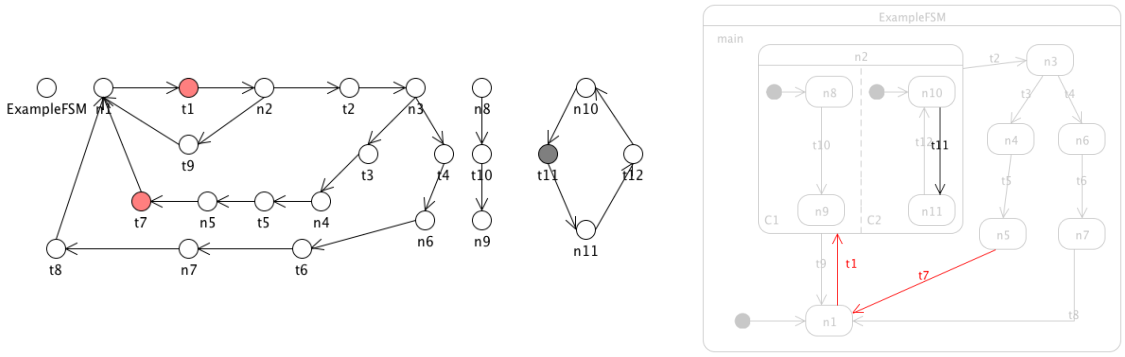
DD Step uses *DDtable* to add more TNodes that are transitively data dependent on the part-of-slice TNodes.

DD Step is a vital step in almost all SBM slicers that use dependence-based analysis. After all, slicing is a static analysis technique based on define-use relationships among elements. An SBM slicer needs to iteratively find more transitions based on define-use relationships of variables. This define-use relationship is the data dependence among nodes.

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) enlarges the size of V_{Rv}. It terminates when no more change to the V_{Rv} is possible.

Algorithm 4.3 shows how DD Step works. Its inputs include *relevantVariables* and *sliceSet*, which represent V_{Rv} and the partial sliced model respectively. It iterates repeatedly, as shown at Line 2. At Line 5, it looks up *DDtable* and checks if any TNode in the sliced model is data dependent on another TNode not in sliced model (*anotherTNode*) with respect to a relevant variable. If so, *anotherTNode* will be added into the sliced model, as shown at Line 6. At Line 7, the monitored variables of *anotherTNode* are added into *relevantVariables*.

At the end of each iteration, the algorithm checks whether there have been any new variables added to the relevant variables at Line 11. If there are any changes, the algorithm has to continue searching for more TNodes.



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

(b) The Example FOSM in the ROS

Figure 4.15: The Example FOSM in the ROS after DD Step

Algorithm 4.3: DD Step in General Iterative Slicing Stage

Input: sliceSet, relevantVariables, DDtable
Output: SliceSet, relevantVariables

```
1  set listOfRelevantVariables := {};
2  repeat
3      set prevSizeRelVar := relevantVariables.size();
4      foreach v in relevantVariables do
5          if DDtable contains v⇒(tnode, anotherTNode) AND sliceSet contains tnode then
6              ADD anotherTNode to sliceSet;
7              ADD anotherTNode.monitoredVariables to relevantVariables;
8          end
9      end
10     set currSizeRelVar := relevantVariables.size();
11 until currSizeRelVar == prevSizeRelVar
```

We need to repeatedly add more TNodes in order to add more TNodes on which the FOI is transitively data dependent. Re-consider the example in Figure 4.13. We know that $V_{Rv} = [v1, v2]$ and $t11$ is in the sliced model, from the previous step. Now, consider the transition *t1* with 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*) which controls $v3$, that transition affects *t1*, and indirectly affects *t11*, which again indirectly affects the FOI. Therefore, we must include $v3$ into V_{Rv} and perform the node adding operation in another iteration.

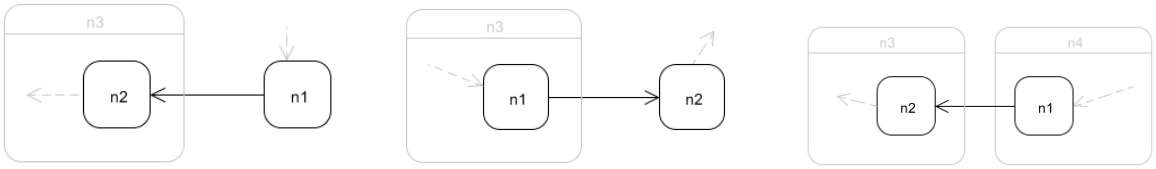
The sliced model now contains TNodes *t1* and *t7*, as shown in Figure 4.15.

4.4.2.2 Replacing Cross-Hierarchy Transition

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

A cross-hierarchy transition may transit from the outside of a hierarchy boundary to its inside; in this case, the destination state's rank is higher than the source state's rank

(Figure 4.16a). A cross-hierarchy transition may transit from the inside of a hierarchy boundary to its outside; in this case, the destination state's rank is lower than the source state's rank (Figure 4.16b). A cross-hierarchy transition may even transit from the inside of a hierarchy boundary, to the inside of another hierarchy boundary; in this case, the destination state's rank may be higher than, lower than or same as the source state's rank (Figure 4.16c).



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

Figure 4.16: Examples of Cross-Hierarchy Transitions

The complexity comes from the composite states which hierarchy boundaries are involved in a cross-hierarchy transition t_{ch} (e.g., *n3* or *n4* in Figure 4.16). Consider this composite state n_{cps}; note that n_{cps} is neither the source state nor the destination state of t_{ch}, but it is an ancestor state of either of them. If n_{cps} is selected into the slice due to some other reasons (e.g., some other nodes are control dependent on it), and FormlSlicer does not select t_{ch} into the slice, the state configuration in the sliced model cannot correctly simulate the state configuration in the original model, because n_{cps} will not be entered in the sliced model while it is entered in the original model, and will not be exited in the sliced model while it is exited in the original model.

This causes many potential problems. Assume that this composite state n_{cps} is a source state of another important transition t_x, then any appropriate events can trigger t_x to leave n_{cps}. The correct triggering of t_x is therefore affected by a cross-hierarchy transition t_{ch} which either enters or exits n_{cps}. If we do not select t_{ch} into the slice, n_{cps} will not be entered or exited in a correct manner in the sliced model, that will cause the execution of t_x incorrect, and consequently any important actions brought by t_x will be incorrect. This causes the sliced model to be incorrect.

Some may ask why we cannot simply detect that important composite state n_{cps} and create a path from a starting part-of-slice state to n_{cps}. The caveat of doing so is that it cannot accommodate many corner cases. For example, in the example of Figure 4.16a, if there is another transition t_y from *n2* to another descendant state of *n3*, then t_y cannot be

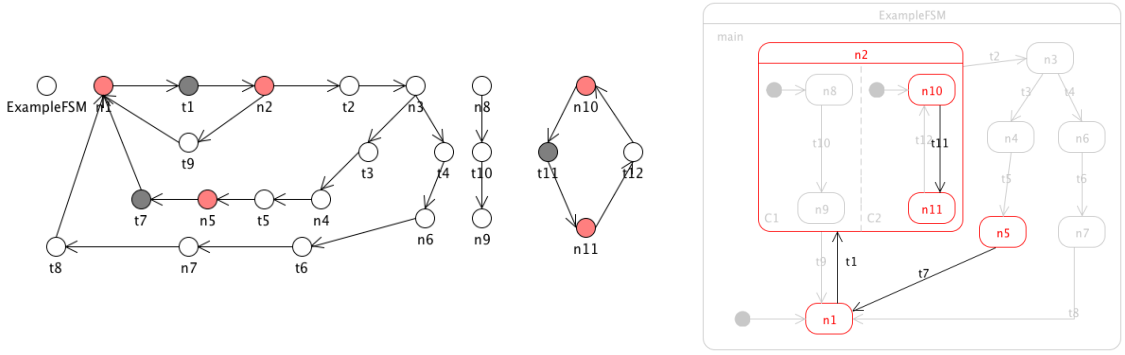
simulated in the sliced model because there is no way to transit from *n3* to its descendant state using a transition. The example can be made more complex. We will then need a way to fix all possible corner cases; this will make the algorithm long and inelegant and yet it is still hard to guarantee that the sliced model is 100% correct.

Based on the above analysis, FormlSlicer has this step—"Replacing Cross-Hierarchy Transition"—after the DD Step in the Multi-Stage Model Slicing Process. The step looks for any out-of-slice cross-hierarchy transitions, and for each of them, replaces the cross-hierarchy transition with a transition carrying only "true" in its guard condition, called **a "true" transition**. The "true" transition is added into the sliced model.

The reason of using a "true" transition is that 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. We can therefore safely ignore their monitored or controlled values.

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 is a turning point to bring in the SNodes (equivalent to states).



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

(b) The Example FOSM in the ROS

Figure 4.17: The Example FOSM in the ROS after Transition-to-State Step

For each part-of-slice transition, this step makes its source state and destination state become part-of-slice. In other words, the SNodes that correspond to these source and

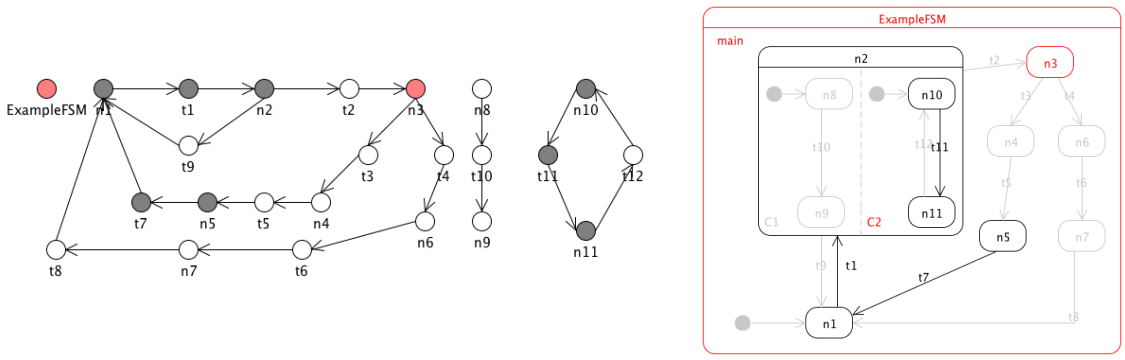
destination states are added into the sliced model. Figure 4.17 shows the effect of this step.

Although this step is simple, it is very important. Firstly, it enriches the fragmented sliced model so that it becomes one step closer to well-formedness, as it is not logical to have a transition without either source or destination state. Secondly, it introduces the initial set of SNodes into the sliced model, which becomes the starting point for the iterative CD-HD Step.

4.4.2.4 CD-HD Step

We have explained the importance of control dependence and hierarchy dependence in Section 4.3.3.1 and Section 4.3.1 respectively. The computation results, *HDtable1* and *CDset*, are used in the CD-HD Step to add more SNodes that are transitively control or hierarchy dependent on the part-of-slice TNodes.

Algorithm 4.4 shows how CD-HD Step works. Similar to the algorithm for DD Step, it has a loop that will terminate only when no more changes occur, as shown at Line 12. In each iteration, there are two lookups. At Line 4, it looks up the hierarchy dependence and adds an SNode into the slice if its child SNodes are part-of-slice. At Line 8, the algorithm looks up the control dependence and adds an SNode which controls the execution of other part-of-slice nodes.



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

(b) The Example FOSM in the ROS

Figure 4.18: The Example FOSM in the ROS after CD-HD Step

The reason for such a repetition is that an SNode can be transitively control dependent or hierarchy dependent on another SNode. For example, a composite state can contain

many descendants in different ranks of state hierarchy. Once a descendant state is selected into the slice, the repetition in CD-HD Step will ensure that all the states from the descendant state to the root state along the state hierarchy are all added into the slice.

Algorithm 4.4: CD-HD Step in General Iterative Slicing Stage

```
   Input: sliceSet, HDtable1, CDset
   Output: SliceSet
 1 set changesOccur
 2 repeat
 3 |   reset changesOccur := false;
 4 |   if HDtable1 contains node⇒parentSNode AND sliceSet contains node then
 5 |   |   ADD parentSNode to sliceSet;
 6 |   |   reset changesOccur := true
 7 |   end
 8 |   if CDset contains node⇒controllingSNode AND sliceSet contains node then
 9 |   |   ADD controllingSNode to sliceSet;
10 |   |   reset changesOccur := true
11 |   end
12 until changesOccur == false
```

Such a repetition of lookups needs to include both control dependence and hierarchy dependence together, because both dependence are based on SNodes.

Figure 4.18 shows the same slicing example after the CD-HD Step. The state *n3* has been added into the sliced model because *n5* is control dependent on it. The state *ExampleFSM* has been added into the slice because many part-of-slice nodes, such as *n1*, are hierarchy dependent on it.

4.4.3 Model Enrichment Stage

Model Enrichment Stage aims to make the FOSMs in the sliced model become well-formed state machines, and as a consequence, the sliced model also becomes a well-formed big state machine. Informally, an FOSM is a well-formed FOSM when all the states are reachable from the pseudo state[8] and all transitions can be triggered when appropriate.

The basic idea in this stage is to try to bring the states far away from one another to come "closer", so that they do not remain disconnected. There are two steps in this stage:

[8] A pseudo state is denoted as a black solid circle with an outgoing, non-labeling transition pointing to the default initial state of a state machine, as introduced in Section 3.2.

Step I: State Merging Step
picks any two suitable states and merges them together;

Step II: True Transitions Creation
makes all part-of-slice states reachable from the pseudo state.

4.4.3.1 Step I: State Merging

The State Merging step brings two states "closer" to each other by merging them. This does not only make the FOSM one step closer to becoming a well-formed FOSM, but also increases the degree of reduction in slicing.

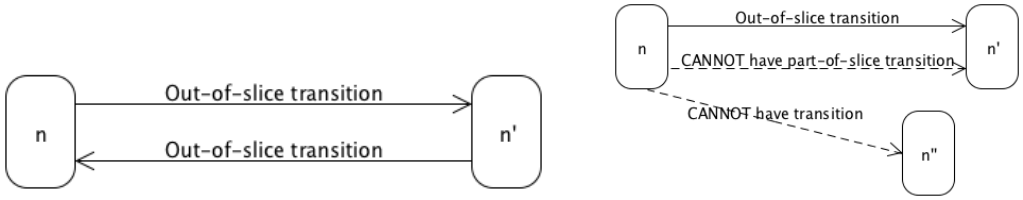
FormlSlicer uses Korel et al.'s two state merging rules that satisfy the *traversability* property[9] [5] :

Rule 1 If there exist out-of-slice transitions from state n to n' and also from state n' to n, 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 a 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 $n'' \neq n'$.

Figure 4.19 illustrates these two rules.



(a) State Merging Rule 1 **(b)** State Merging Rule 2

Figure 4.19: Illustration of State Merging Rules in Step I of Model Enrichment Stage

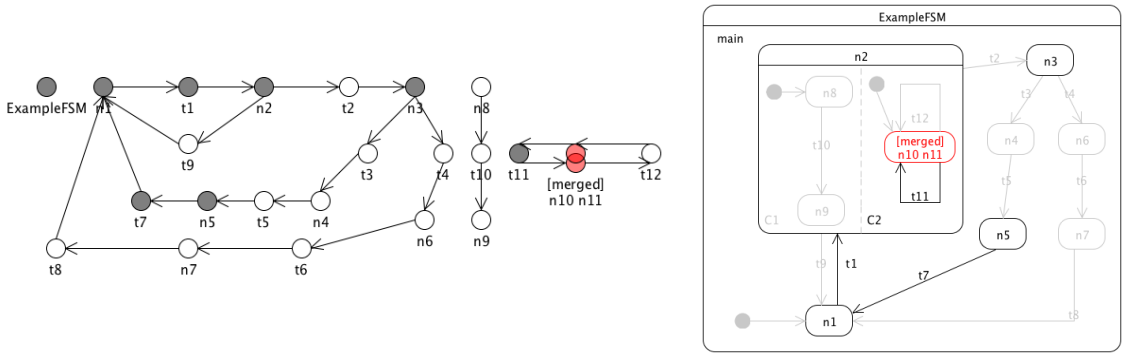
[9] See Section 2.2.5 about traversability property on a slice.

Intuitively, these two rules can be easily justified. In Rule 1, since the transitions between the two states are out-of-slice, they are not important; otherwise, they have already become part-of-slice during DD Step. In Rule 2, from n the machine does not have anywhere else to move except n'. Therefore, it is safe to merge n and n'.

FormlSlicer adjusts the two rules for its own use.

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

2. It will perform state merging only when these two states have the same parent state. Otherwise, the sliced model will become incorrect.



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

(b) The Example FOSM in the ROS

Figure 4.20: The Example FOSM in the ROS after Stage Merging Step

Figure 4.20 shows the same slicing example after 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 Step II: True Transitions Creation

This is the last step for the entire Multi-Stage Model Slicing Process. The True Transitions Creation step makes all part-of-slice states to become reachable from the pseudo state. It starts from the beginning of the FOSM and "probes" to connect the part-of-slice states together.

The step makes use of the concept of *next part-of-slice state*. All the part-of-slice states that are reachable from a state n via an out-of-slice path are called the *next part-of-slice states* of n. Informally, a state n' is the next part-of-slice state of state n if n' is part-of-slice and there is a path that starts from n and reaches n' and that all the states and transitions along this path are out-of-slice and all the states are at the same rank of state hierarchy. A state n may have more than one part-of-slice states because a path from n may branch. We call all of them as the *next part-of-slice state set* of n.

There are two sub-steps in Step II:

Sub-step 1: Search for New Default Initial States We know that each sub-machine has one default initial state. It is the state pointed by a transition without label from the pseudo state. If the default initial state is out-of-slice, this sub-step 1 searches for the new default initial state for the sub-machine.

This sub-step benefits greatly from the use of control dependence. Section 4.3.3.1 has explained this scenario. When a default initial state is a branching state, it will be added into the sliced model because of control dependence. This prevents FormlSlicer to run into the trouble of creating multiple default initial states in the sliced model when the original default initial state is out-of-slice.

As a result of this sub-step, for each sub-machine in the original model, its default initial state in the sliced model is:

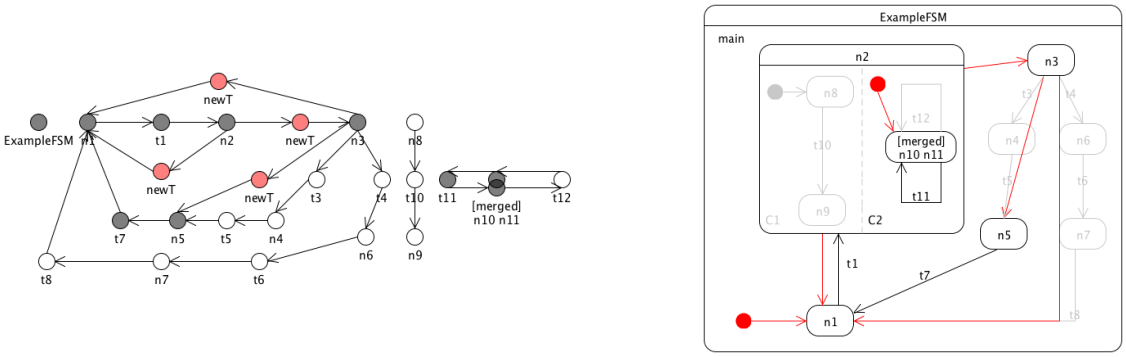
1. the same one in the original model;
2. the next-part-of-slice state of the original default initial state;
3. absent, because the whole sub-machine is out-of-slice.

Sub-step 2: Search for Next Part-of-slice State Consider a path starting from a part-of-slice state n to another part-of-slice state n', and all the other states and transitions along this path are out-of-slice because they are not selected during all the previous steps in Multi-Stage Model Slicing Process. The original model can take this path to move from n to n'. But in the sliced model, n and n' are disconnected.

We need to connect n and n' together in the sliced model. This connection will be a "true" transition[10].

[10]Recall from Section 4.4.2.2 that a transition carrying only "true" in its guard condition is called a "true" transition.

Sub-step 2 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.



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

(b) The Example FOSM in the ROS

Figure 4.21: The Example FOSM in the ROS after True Transitions Creation Step

Figure 4.21 shows the slicing example after this step. We can see that *n3* is now connected to its next part-of-slice state *n1*. 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.

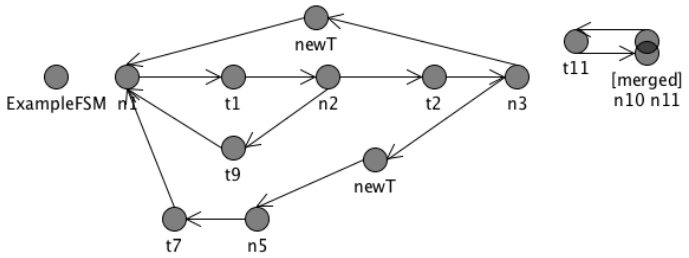
4.4.4 Summary and Postprocessing

We have presented all the steps for the Multi-Stage Model Slicing Process. Figure 4.22a shows the resultant CFGs for the example feature. It looks much smaller than the original CFGs in Figure 4.13a.

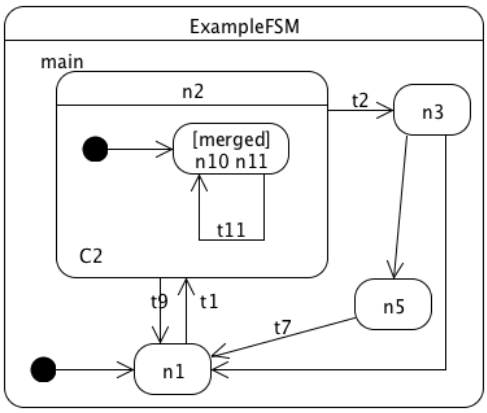
As a postprocessing step, FormlSlicer converts the resultant CFGs into an FOSM, as shown in Figure 4.22b.

The resultant FOSM and the FOI (which remains unchanged before and after slicing) form the sliced model. The *ModelWriter* module in FormlSlicer writes the sliced model to a text file, in the format as specified in Table 4.1.

As mentioned in Section 4.1, FormlSlicer will perform the Multi-Stage Model Slicing Process for multiple times, each time with a different feature as the FOI.



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



(b) The Example FOSM in the ROS

Figure 4.22: The Example FOSM in the ROS after All Steps in Multi-Stage Model Slicing

4.4.5 Testing the Multi-Stage Model Slicing Process

We create a model in order to test the steps in the Multi-Stage Model Slicing Process. Figure B.1 shows the original model. It consists of four FOSMs—*E1* (Figure B.2), *E2* (Figure B.3), *E3* (Figure B.4) and *E4* (Figure B.5). The sliced model with respect to *E1* is shown in Figure B.6.

In Section 4.4, we have explained the majority of steps in the Multi-Stage Model Slicing Process using a slicing example. The slicing example we use is *E2*. The resultant sliced FOSM is shown in Figure B.7. However, there are a few corner cases that are not covered

in the examples; so we create *E3* and *E4*.

Here are the steps in the Multi-Stage Model Slicing Process that are not covered in *E2*:

Replacing Cross-Hierarchy Transition

This is a step in the General Iterative 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* (Figure B.5): *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 (Figure B.9).

State Transition Rule 1

Section 4.4.3.1 presents two state merging rules. In Rule 1, if there exist out-of-slice transitions from state n to n' and also from state n' to n, these two states are merged into one state "n,n'". This is shown in the FOSM *E3* (Figure B.4). The transitions *t3* and *t4* are both out-of-slice and transiting between states *n2* and *n3*; therefore *n2* and *n3* are merged (Figure B.8).

Sub-step 1: Search for New Default Initial States

Section 4.4.3.2 presents the "True Transitions Creation" step in the Model Enrichment 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* is not added into the sliced model, making the state *nk* to be the next part-of-slice state for the pseudo state in the region of *withinactive*. In the sliced model, *nk* becomes the new start state in the sub-machine within the region *withinactive*.

As a summary, FormlSlicer has implemented correctly on all the steps in the Multi-Stage Model Slicing Process.

Chapter 5

Correctness of FormlSlicer

This chapter presents a correctness proof to show that the sliced model produced by the Multi-Stage Model Slicing Process (described in Chapter 4) can simulate the original model.

5.1 Overview

5.1.1 Purpose of the Proof

As the sliced model is used to replace the original model for safety property checking, the non-negotiable requirement for the sliced model is to guarantee that

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

whereby FOI is the feature of interest (FOI), ROS is the rest of the system (ROS) executing with the FOI (i.e., all the feature-oriented state machines except the FOI state machine), $ROS_{\mathcal{L}}$ is the ROS after slicing, $M_{ROS+FOI}$ is the original model, $M_{ROS_{\mathcal{L}}+FOI}$ is the sliced model and φ is the safety property for FOI.

This is equivalent to saying that **the execution traces in FOI in the original model is a subset of the execution traces in FOI in the sliced model**. In such case, 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. It is acceptable if there are some execution traces in the sliced model that are impossible to occur in the original model[1].

[1]This is the basic correctness property of a sliced model. See Section 2.2.5 on discussions in literature about correctness of SBM slicing.

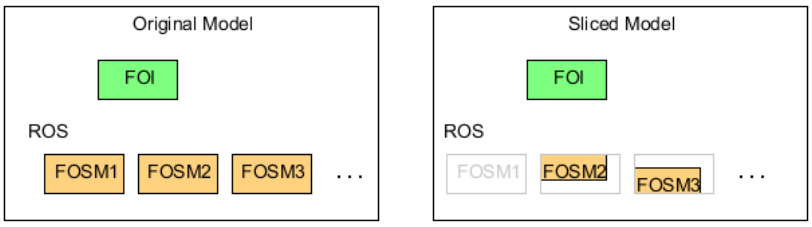


Figure 5.1: The Slicing Environment with Original Model and Sliced Model

Figure 5.1 shows a side-by-side comparison between the original model and sliced model. Some FOSMs in ROS are entirely absent in the sliced model; some are partially absent.

5.1.2 Intuition of the Proof

We can visualize an execution trace in the original model as a long sequence of execution steps:

$$e_0, e_1, e_2, \ldots, e_k$$

Due to feature interactions between FOI and other FOSMs in the ROS, there are certain execution steps that occur in other FOSMs to support the execution within FOI. For example, the initial execution step in FOI may be triggered only when a particular system-controlled variable v is equal to a certain value *'val'*, as illustrated in Figure 5.2; in this case, the execution step in $FOSM1$ that assigns v to be *'val'* must be executed before the initial execution step in FOI, so as to trigger the latter.

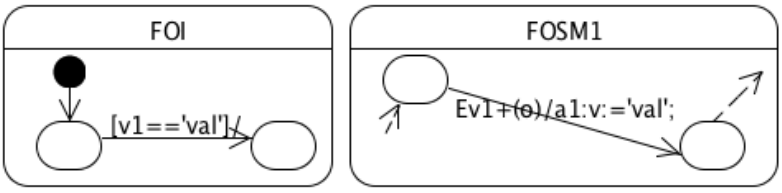


Figure 5.2: A Simple FOI Executing with Another FOSM in the Rest of System

In order to show that the set of execution traces in FOI in the original model is a subset of the execution traces in FOI in the sliced model, we want to prove that any given

execution trace in original model can be *simulated* by an execution trace in the sliced model.

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 execution step in the sliced model's execution trace.

The intuition of our proof is to show that the snapshot between two execution steps in original model can be projected to a corresponding snapshot in sliced model, and that such a projection is maintained before and after each step in original model's execution trace.

5.1.3 Proof Outline

The proof is using induction to show that the execution trace in the original model can be simulated by at least one execution trace in the sliced model. Section 5.2 defines terminology that is useful for the proof. This section introduces some concepts, such as state configuration and interpretation. Section 5.3 describes the state transition rule that is standard to a hierarchical concurrent state machine. Section 5.4 writes the Multi-Stage Model Slicing Process using the semantics defined in Section 5.2. Section 5.5 shows the proof. It starts with the concept of relevant variables, and describes the projection of snapshot, transition and execution step from the original model to the sliced model.

5.2 Semantics

5.2.1 Variables, States, Regions, Transitions and Model

In the previous chapters, we have informally introduced and used the concepts of variable, state, region, transition and model. This section will define their respective notations in the proof.

Variables

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].

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

States

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 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 pseudo state is represented as a black solid circle $\bullet$ in an FOSM; it is a notation to point to the default initial state.

A state can be either a basic state (i.e., state without child regions and states), or a composite state (i.e., state that contains child regions and states). A composite state p can have many child states $n_1, \ldots, n_k$, denoted as a set $ChildStates(p) = [n_1, \ldots, n_k]$; also, a state's parent state is denoted as $ParentState(n_1) = p$.

Regions

We use r to denote an orthogonal region inside a composite state[3]. The composite state n that contains this region is denoted as $ParentStateOfRegion(r)$; and $ChildRegions(n) = [r]$[4]. On the other hand, if a state n is in the sub-state machine enclosed by an orthogonal region r (i.e., n's rank in state hierarchy is one level higher than the rank of $ParentStateOfRegion(r)$), we use $ParentRegion(n) = r$ to denote this relationship. Section 3.2 has shown a FORML example to illustrate this relationship.

We also need a notation to indicate that two orthogonal regions are sibling with each other when these two regions are both contained within the same composite state n and they are at the same rank of state hierarchy:

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 $r \neq r'$.*

Transitions

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'.*

[3]See Section 3.2 for explanations on concepts like "orthogonal regions" and "composite state" in FORML

[4]There may be more than one orthogonal regions in a composite state; so here we use a set of regions, $[r]$.

Moreover, $ss(t)$ and $ds(t)$ refer to the source state and destination state of t, respectively. In Definition 2, $ss(t) = n$ and $ds(t) = n'$. The source state and destination state of t do **not** need to be at the same rank of state hierarchy, but there is a restriction: if a state is a child state of a composite state that has multiple orthogonal regions, we do not allow any transitions crossing hierarchy boundary from or to this state[5].

Model

As introduced in Section 3.2, FormlSlicer treats the Behavior Model in FORML as a big state machine. This big state machine only has a composite state containing many orthogonal regions. Each orthogonal region contains one sub-state machine, which is an FOSM. In some literature, this is called an 150% model [28].

In this proof, We use M to denote a model. As M contains k FOSMs, $[F_1, \ldots, F_k]$, we write it as:

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

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, including composite states and basic states. It contains a finite set of orthogonal regions, R_F. The state and region form a containment relation in hierarchy (each composite state $n \in S_F$ contains one or more regions; each region contains a sub-state machine). S_F^I is the set of all 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 $ds(t) \in S_F$. There is a set of variables, V_F, which is controlled or monitored by F.

We know that a basic state machine without hierarchy and concurrency constructs can be in only one state at a time; the state it is in at any given time is called the *current state*. However, because of hierarchy and concurrency, an FOSM can be in multiple states at a time. Therefore, we use N to refer to the **state configuration** of the FOSM.

This is not saying that the machine can be in any arbitrary combination of states at a time. Rather, N is the set of current states $N \subseteq S_F$ such that if any state n is in N,

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

then so are all of n's ancestors, and if any composite state n is in N, then for each r in $ChildRegions(n)$, there must be one state contained in r that are in N too. (A state machine's current state configuration is the set of current states in a hierarchy tree where a current composite state contains one current substate per orthogonal region. [31])

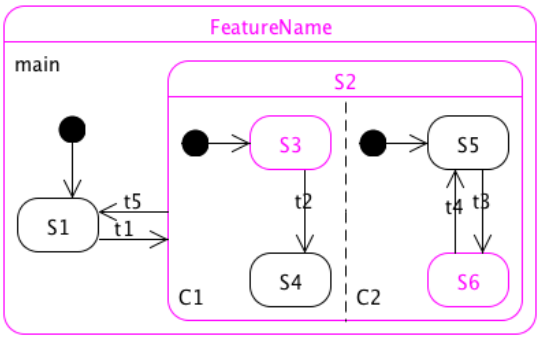


Figure 5.3: The FORML Example from Figure 3.1 with its Current States Highlighted in Magenta

Figure 5.3 shows an example of an FOSM with current states in $FeatureName, S2, S3, S6$ simultaneously. The state configuration is $N = [FeatureName, S2, S3, S6]$. Note that the root state is always in the state configuration.

Since a model M is a big state machine, the definition of state configuration also applies to M. Similarly, it 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 $S_M^I \subseteq S_F$. For any FOSM F contained within one orthogonal region of M, $S_F \subset S_M \wedge R_F \subset R_M \wedge T_F \subset T_M \wedge V_F \subset V_M$.

We use σ to denote an **interpretation** which maps variables to their values; thus, $\sigma(v)$ represent the interpretation of the variable v in the environment. The domain of σ is the set of all variables $V_{sys} \cup V_{env}$ in the slicing environment.

5.2.3 Execution Step

Informally, a **snapshot** is an observable point in an model's execution [32]; it refers to the status of an model between execution steps.

Definition 3. *We define the snapshot of a model as a combination of the model's state configuration, N, and the interpretation of variables, σ, at that particular point in time; we denote the snapshot as (N, σ).*

An execution step changes the snapshot of the model. Through an execution step, the model effectively exits the set of current states and enters the set of destination states of the transitions in the execution step[6]; meanwhile, the values of some variables may be changed.

Here we define an execution step formally.

Definition 4. *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 actions through the execution or environmental changes.*

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 for all $1 \leq i \leq k$, we write $t_i \subset e$ to denote that t_i is one of the many 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**.

In general, we want to precisely describe what combination of transitions can occur concurrently in a single execution step. Recall Definition 1 on sibling regions and the various accessor functions in Section 5.2.1.

Definition 5. *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 $ss(t_j) \xrightarrow{t_j} ds(t_j)$ where $ss(t_j) \in N$ and $ds(t_j) \in N'$ for all $1 \leq j \leq k$;*
- *$ParentRegion(ss(t_j)) = ParentRegion(ds(t_j))$ 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 $ParentRegion(ss(t_j)) \parallel ParentRegion(p)$, for any $1 \leq i \leq k \wedge 1 \leq j \leq k \wedge i \neq j$;*

[6] An model starts executing from an initial snapshot, (N^I, σ^I).

Figure 5.4 illustrates an example of an execution step which consists of two concurrent transitions t_m, t_n. One may observe that the two transitions t_m, t_n are "contained" within their respective orthogonal region only. This is consistent as FormlSlicer's restriction on transitions crossing hierarchy boundary on sibling regions, as discussed in Section 3.3.

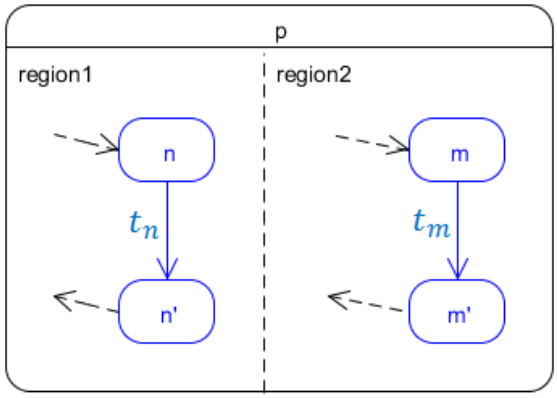


Figure 5.4: Concurrency in Orthogonal Regions as an Execution Step (Only Blue Colored Components are Relevant in the Execution)

Slice Set This symbol $\mathcal{L}$ is used as a subscript to annotate a model component in the sliced model. A state n in the sliced model is denoted as $s \in S_{\mathcal{L}}$. A transition t in the sliced model is denoted as $t \in T_{\mathcal{L}}$. Thus, an original FOSM F would become $F_{\mathcal{L}}$ after slicing. 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 $\sigma_{\mathcal{L}}$.

Monitored and Controlled Variables The set of monitored variables and controlled variables of an execution e are denoted as $mv(e)$ and $cv(e)$ respectively.

5.2.4 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 6. *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 7. *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 8. *We say $n \in S_M$ (or $t \in T_M$) is **control dependent** on $n' \in S_M$, denoted as or $n \xrightarrow{cd} n'$ (or $t \xrightarrow{cd} n'$), iff n' has at least two outgoing edges, t_k and t_l,*

1. *for all maximal paths from t_k, n (or t) always occurs and it occurs before any occurrence of n';*
2. *there exists a maximal path from t_l on which either n (or t) does not occur, or n (or t) is strictly preceded by n'.*

Transitivity on dependences is also considered. If $n_1 \xrightarrow{hd} n_2$ and $n_2 \xrightarrow{hd} n_3$, then $n_1 \xRightarrow{hd} n_3$; this means that the state n_3 is the grandparent state of n_1. Similarly, if $n_1 \xrightarrow{dd} n_2$ and $n_2 \xrightarrow{dd} n_3$, then $n_1 \xRightarrow{dd} n_3$. The transitivity applies to control dependence too.

5.3 State Transition Rule

The state transition rule defines how the current state configuration N evolves to the next state configuration N' in the original and the sliced model.

Intuitively, we know that through an execution step t, the current state configuration exits the source state of t and enters the destination state. This forms the base rationale of our state transition rule[7].

First of all, 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.

[7]In hierarchical systems without concurrency, the state transition rule for one transition t is non-trivial: the set of entered states includes not only $ds(t)$, but also all of the $ds(t)$s ancestor states plus the default states of $ds(t)$ and of its entered descendants [32]. By adding in concurrency, it becomes more complicated. Due to the complexity of such a state machine, we will not formalize the state transition rule strictly.

The least common ancestor between two states is the state of highest rank that is an ancestor state of both states[33, 11].

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

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

S_M refers to the set of all states in model M.

- $N \subseteq S_M$ and $N' \subseteq S_M$ are the current and next state configuration of the model;
- $exited(t)$ is the set of all states including:
  - the source state itself ($ss(t)$),
  - the ancestor states of $ss(t)$ up along the tree of state hierarchy before reaching the least common ancestor with $ds(t)$ ($AncestorTillLCA(ss(t), ds(t))$),
  - the descendant states of $ss(t)$ ($Descendants(ss(t))$).
- $entered(t)$ is the set of all states including:
  - the destination state itself ($ds(t)$),
  - the ancestor states of $ds(t)$ up along the tree of state hierarchy before reaching the least common ancestor with $ss(t)$ ($AncestorTillLCA(ds(t), ss(t))$),
  - the recursively identified default initial states of $ds(t)$ and its entered descendant states[8] ($InitDesc(ds(t))$).

A more detailed version of the state transition rule will be:

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

5.4 FormlSlicer's Multi-Stage Model Slicing

Section 4.4 has elaborated on how the FormlSlicer performs a Multi-Stage Model Slicing Process on an FOSM with respect to the FOI. This section writes the process using the terminologies defined in Section 5.2.

[8] 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 the default initial child state within the default initial state is also entered. This is defined recursively.

5.4.1 Definitions

Firstly, we need to formalize some definitions.

The concept of the next part-of-slice state set has been informally introduced in Section 4.4.3.2. Here we define it formally.

Definition 9. *The **next part-of-slice state set** of state n, denoted as $\widetilde{npos}(n)$, is the set of states such that for each $n' \in \widetilde{npos}(n)$:*

- $n' \in S_{\mathcal{L}}$;
- $\exists$ *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 9. In this case, there is exactly one out-of-slice transition between the state n and its next-part-of-slice state n'.

The concept of relevant variables has been informally introduced in Section 4.4.1 and Section 4.4.2.1 where it is used to initiate the initial selection of nodes into the sliced model. We need to formally define this concept.

Definition 10. *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' such that $t' \xrightarrow{dd} t$ and $v \in mv(t')$.*

Note that T_{FOI} represent $\bigcup_{\forall t \in T_{FOI}} t$ in Definition 10.

In other words, a variable is a relevant variable if it directly or indirectly controls the FOI. It appears in at least one data dependence entry in the DD table and there is at least one transition in the sliced model which uses it.

5.4.2 Multi-Stage Model Slicing Process

We denote T_{ROS} to represent $\bigcup_{\forall f \in ROS. \forall t \in T_f} t$ and S_{ROS} to represent $\bigcup_{\forall f \in ROS. \forall n \in S_f} n$.

At each step, certain model elements from the original model are added into the sliced model.

1. Initiation Stage

(a) Variable Extraction Step
Let

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

(b) Initial Transition Selection Step
$\forall v \in Rv.(\exists t \in T_{ROS}.v \in cv(t)) \Rightarrow (t \in T_{\mathcal{L}} \wedge mv(t) \subset Rv)$.

2. General Iterative Slicing Stage

(a) DD Step
$\forall t, t' \in T_{ROS}.(t \xrightarrow{dd} t' \wedge t \in T_{\mathcal{L}}) \Rightarrow (t' \in T_{\mathcal{L}} \wedge mv(t') \subset Rv)$.

(b) Replacing Cross-Hierarchy Transition
$\forall t \in T_{ROS}.(t \notin T_{\mathcal{L}} \wedge ParentState(ss(t)) \neq ParentState(ds(t)))$, create true transition $ss(t) \xrightarrow{t_{true}} ds(t)$ and add t_{true} to $T_{\mathcal{L}}$.

(c) Transition-to-State Step
$\forall t \in T_{ROS}.t \in T_{\mathcal{L}} \Rightarrow (ss(t) \in S_{\mathcal{L}} \wedge ds(t) \in S_{\mathcal{L}})$.

(d) CD-HD Step
$\forall n, n' \in S_{ROS}.(n \xrightarrow{cd} \xrightarrow{hd} n' \wedge n \in S_{\mathcal{L}}) \Rightarrow n' \in S_{\mathcal{L}}$.

3. Model Enrichment Stage

(a) Step I: State Merging Step

i. Rule 1
Consider two states $n, n' \in N_{ROS}.n \in S_{\mathcal{L}}$. Let $T_{n,n'}$ to be the set of all transitions $t \in T_{ROS}$ such that $ss(t) = n$ and $ds(t) = n'$, and $T_{n',n}$ to be the set of all transitions $t \in T_{ROS}$ such that $ss(t) = n'$ and $ds(t) = n$. If $T_{n,n'} \neq \emptyset \wedge T_{n',n} \neq \emptyset \wedge (\forall t \in T_{n,n'} \cup T_{n',n} \Rightarrow t \notin T_{\mathcal{L}}))$, then n and n' can be merged together.

ii. Rule 2
Consider two states $n, n' \in N_{ROS}.n \in S_{\mathcal{L}}$. Let $T_{n,n'}$ to be the set of all transitions $t \in T_{ROS}$ such that $ss(t) = n$ and $ds(t) = n'$. Let T_n be the set of all transitions $t \in T_{ROS}$ such that $ss(t) = n$. If $T_{n,n'} \cap T_{\mathcal{L}} = \emptyset \wedge T_{n,n'} = T_n$, then n and n' can be merged together.

(b) Step II: True Transitions Creation
$\forall n \in S_{\mathcal{L}}.(\forall n' \in \widetilde{npos}(n)$, create true transition $n \xrightarrow{t_{true}} n'$ and add t_{true} to $T_{\mathcal{L}}$.

5.5 Proof

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

In Subsection 5.1.2, we present that the intuition is to show that the snapshot in original model can be projected to a corresponding snapshot in sliced model, and that such a projection is maintained before and after each step in original model's execution trace.

Recall from Definition 3 that a snapshot consists of two components: the state configuration and the interpretation of variables. Similarly, the projection of snapshot from the original model to the sliced model also include two notions:

1. the value of any relevant variable is the same in both the original model and the sliced model;
2. The state configuration of the original model has certain relation with that of the sliced model.

For the first notion, we only consider the system-controlled variables. Because the environment-controlled variables are influenced by the external environment, we cannot and need not to prove their values if any of them are monitored in a transition. For example, it is meaningless to prove that a variable like "actual temperature" has a value of "larger than 37℃".

For the second notion, based on the fact that true transitions are added into $T_{\mathcal{L}}$ connect two part-of-slice states[9], we can imagine that sometimes the set of current states in sliced model is larger than the set of current states in original model. This is acceptable because we allow extra execution traces in the sliced model. However, the state configuration in sliced model is not strictly a superset of that in original model, because those states that are not added into the sliced model cannot possibly appear in the state configuration of the sliced model. Based on these two observations, we write that the state configuration N in original model has the following relation with the state configuration $N_{\mathcal{L}}$ in sliced model, if that sliced model simulates the original model:

$$N \cap \mathcal{L} = N_{\mathcal{L}}$$

[9] See "Step II: True Transitions Creation" in Section 4.4.3.2

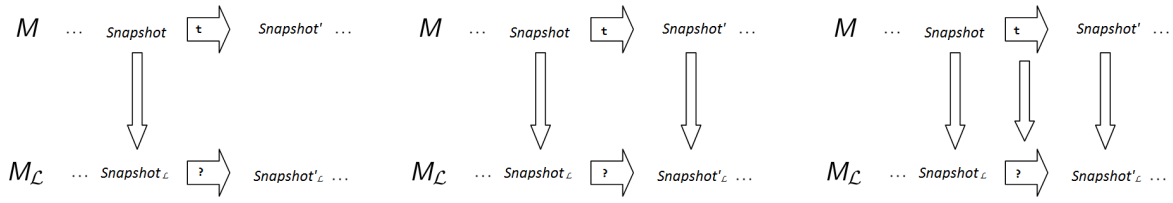
Definition 11. *We define that a snapshot in the original model* (N, σ) *is* ***is projected to*** *another snapshot in the sliced model,* $(N_{\mathcal{L}}, \sigma_{\mathcal{L}})$*, when*

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

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

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

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 5.5c), such that if the snapshot in the original model is projected to the snapshot in the sliced model before the transition (Figure 5.5a), 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 5.5b).

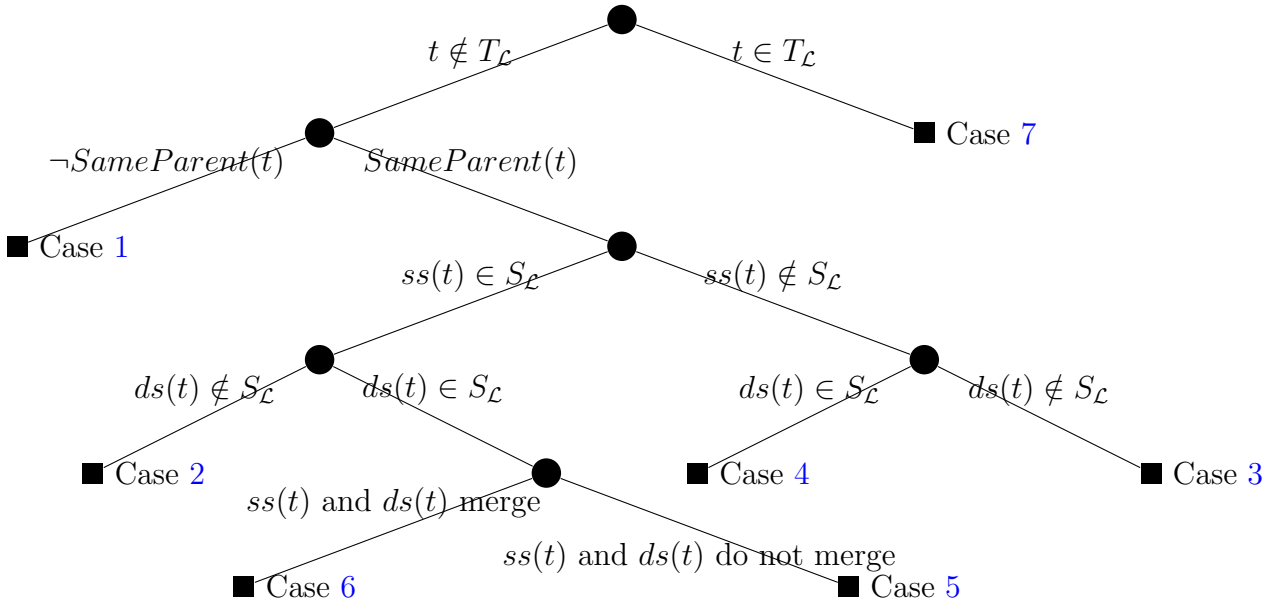


(a) Before Transition **(b)** After Transition **(c)** Projection of a Transition

Figure 5.5: Projection of a Transition from the Original Model to the Sliced Model

This can be proved by case-based analysis. For brevity, we denote $SameParent(t)$ to represent the condition of $(ParentState(ss(t)) = ParentState(ds(t)))$.

We divide the situation into 7 different cases as shown in the decision tree below:



Lemma 1. *Consider a transition t in the original model M, $M \vdash t : (N, \sigma) \Rightarrow (N', \sigma')$[10] The projection function of t to its counterpart $(t_{\mathcal{L}} \vee \epsilon)$ (consider $t_{\mathcal{L}}$ to be $M_{\mathcal{L}} \vdash t_{\mathcal{L}} : (N_{\mathcal{L}}, \sigma_{\mathcal{L}}) \Rightarrow (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$) in the sliced model is:*

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

such that given $P((N, \sigma)) = ((N_{\mathcal{L}}, \sigma_{\mathcal{L}}))$, then:

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

Proof.

[10] Recall from Section 3.2 that a model M is a big state machine consisting of k FOSMs, each as a sub-state machine in one of its k orthogonal regions. Also, recall from Section 5.2.3 that we write $e = t$ when the execution step consists of only one transition t.

Given Conditions Because $P((N,\sigma)) = ((N_{\mathcal{L}}, \sigma_{\mathcal{L}}))$, we know that:

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

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

Also, the state transition rule is:

$$\begin{aligned} N' =& (N - ss(t) - AncestorTillLCA(ss(t), ds(t)) - Descendants(ss(t))) \\ & \cup ds(t) \cup AncestorTillLCA(ds(t), ss(t)) \cup InitDesc(ds(t)) \end{aligned} \tag{10}$$

The Case of Cross-Hierarchy Transition This proves for **Case** (1).

When $(t \notin T_{\mathcal{L}}) \wedge (\neg SameParent(t))$, the step of Replacing Cross-Hierarchy Transition in General Iterative Slicing Stage will replace t with a "true" transition $t_{true} \in T_{\mathcal{L}}$, which preserves the original $ss(t)$ and $ds(t)$. Then because of the Transition-to-State Step,

$$ss(t), ds(t) \in \mathcal{L} \tag{11}$$

Because of (11) and the CD-HD Step in General Iterative Slicing Stage, we get:

$$AncestorTillLCA(ss(t), ds(t)) \subset S_{\mathcal{L}}, \quad AncestorTillLCA(ds(t), ss(t)) \subset S_{\mathcal{L}}. \tag{12}$$

We can also observe that:

$$Descendants(ss(t)) \cap S_{\mathcal{L}} = Descendants_{\mathcal{L}}(ss(t)) \tag{13}$$

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

Given the condition (8), (10), (11), (12), (13) and (14), we can deduce that:

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

Next, we want to prove that $\sigma'(v) = \sigma'_{\mathcal{L}}(v) \forall v \in Rv$. In order to do so, we can firstly prove $\sigma(v) = \sigma'(v) \forall v \in Rv$ by assuming in contradiction that there is a variable $v \in Rv$ that has its value changed through t, making $\sigma(v) \neq \sigma'(v)$. Then $v \in cv(t)$. Based on Definition 10, there must be another transition $t_x \in T_{\mathcal{L}}$ such that $v \in mv(t_x)$ and a sequence of execution steps exist between t_x and t. According to the DD Step in General Iterative Slicing Stage, as $t_x \xrightarrow{dd}_v t, t_x \in T_{\mathcal{L}}$, so $t \in T_{\mathcal{L}}$ which contradicts with the given condition $t \notin T_{\mathcal{L}}$. Therefore:

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

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

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

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

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

Given both (15) and (18), we can deduce that $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$ by projecting the transition t in the original model to t_{true} in the sliced model.

The Case of Having Only Source State in Slice This proves for **Case** (2).

When $ss(t) \in S_{\mathcal{L}} \wedge ds(t) \notin S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SameParent(t)$, the state configuration in the sliced model does not change when the state configuration in original model changes from N to N' through the transition t. We observe that:

$$SameParent(t) \Rightarrow (AncestorTillLCA(ss(t), ds(t)) = AncestorTillLCA(ds(t), ss(t)) = \emptyset) \tag{19}$$

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

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

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

Because of (21), (8) and the state transition rule (10), and the fact that $exited(t)$ will not affect the subset relation between state configuration in the original model and $N_{\mathcal{L}}$, we have

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

Next, we want to prove that $\sigma'(v) = \sigma_{\mathcal{L}}(v) \forall v \in Rv$. In order to do so, we can firstly prove $\sigma(v) = \sigma'(v) \forall v \in Rv$; this proof will be exactly same as that in the Case 1 because of the DD Step. Given condition (9), we can therefore deduce that

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

Given both (22) and (23), we can deduce that $P((N', \sigma')) = (N_{\mathcal{L}}, \sigma_{\mathcal{L}})$ by projecting the transition t in the original model to epsilon ϵ in the sliced model.

The Case of Having Nothing in Slice This proves for **Case** (3).

When $ss(t) \notin S_{\mathcal{L}} \wedge ds(t) \notin S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SameParent(t)$, the state configuration in the sliced model does not change when the state configuration in original model changes from N to N' through the transition t. This case is same as Case (2) except that $ss(t)$ is not in the slice. Because of the fact that $exited(t)$ in state transition rule (10) will not affect the subset relation between state configuration in the original model and $N_{\mathcal{L}}$, this difference does not matter. Therefore, the proof about state configuration will be exactly the same as in the Case (2). Therefore:

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

Next, we want to prove that $\sigma'(v) = \sigma_{\mathcal{L}}(v) \forall v \in Rv$. In order to do so, we can firstly prove $\sigma(v) = \sigma'(v) \forall v \in Rv$; this proof will be exactly same as that in the Case 1 because of the DD Step. Given condition (9), we can therefore deduce that

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

Given both (24) and (25), we can deduce that $P((N', \sigma')) = (N_{\mathcal{L}}, \sigma_{\mathcal{L}})$ by projecting the transition t in the original model to epsilon ϵ in the sliced model.

The Case of Having Only Destination State in Slice This proves for **Case** (4).

When $ss(t) \notin S_{\mathcal{L}} \wedge ds(t) \in S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SameParent(t)$, the step of Step II: True Transitions Creation in Model Enrichment Stage has searched from another state $n \in \mathcal{L}$ to reach $ds(t) \in \widetilde{npos}(n)$ and creates a "true" transition $n \xrightarrow{t_{true}} ds(t)$. The step has ensure that all the states along the path $[n_1, \ldots, n_k]$ ($n_1 = n$, $n_k = ds(t)$) are at the same rank of state hierarchy, and therefore in the state transition rule (10) we know that:

$$AncestorTillLCA(n, ds(t)) = AncestorTillLCA(n_i, n_j) = \emptyset . \forall (i, j \in [1 \cdots k]) \wedge (i \neq j) \tag{26}$$

$$(ds(t) \in \mathcal{L}) \Rightarrow (InitDesc(ds(t)) \cap \mathcal{L} = InitDesc_{\mathcal{L}}(ds(t))) \tag{27}$$

In the sliced model, the "true" transition has changed the state configuration such that $ds(t)$, $InitDesc_{\mathcal{L}}(ds(t))$ will be entered and n and $Descendants(n)$ will be exited (because of (26), we will ignore their ancestor states in state transition rule (10)).

From the analysis result of Case (2) and Case (3), we know that the subset relation of state configurations between sliced model and original model is maintained when the state configuration in the original model changes through a sequence of transitions while

the state configuration in the sliced model remains unchanged. Therefore, condition (8) holds in this case.

Because of (26), (27), and (8), we can get that

$$N' \cap \mathcal{L} \subseteq N'_{\mathcal{L}}. \tag{28}$$

Next, we want to prove that $\sigma'(v) = \sigma'_{\mathcal{L}}(v) \forall v \in Rv$. In order to do so, we can firstly prove $\sigma(v) = \sigma'(v) \forall v \in Rv$; this proof will be exactly same as that in the Case 1 because of the DD Step. Together with condition (9), we get

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

Given both (28) and (29), we can deduce that $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$ by projecting the transition t in the original model to t_{true} in the sliced model.

The Case of Having Source and Destination States in Slice This proves for **Case** (5).

When $ss(t) \in S_{\mathcal{L}} \wedge ds(t) \in S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SameParent(t)$, the Step II: True Transitions Creation in Model Enrichment Stage will add in a "true" transition t_{true} to the slice connecting $ss(t)$ and $ds(t)$. Thus we know that the state configuration in the sliced model will change through t_{true} in the same fashion as that in the original model through t. Given condition (8), we thus know that:

$$N' \cap \mathcal{L} \subseteq N'_{\mathcal{L}}. \tag{30}$$

Next, we want to prove $\sigma'(v) = \sigma'_{\mathcal{L}}(v) \forall v \in Rv$. Like the proof for all above cases, the fact that $t \notin T_{\mathcal{L}}$ is that it does not control any relevant variables and thus is not selected into the slice during the DD Step. Similarly, we get:

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

Given both (30) and (31), we can deduce that $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$ by projecting the transition t in the original model to t_{true} in the sliced model.

The Case of Having a Merged State in Slice This proves for **Case** (6).

When $n_{merged} = (ss(t) \vee ds(t)) \in S_{\mathcal{L}} \wedge t \notin T_{\mathcal{L}} \wedge SameParent(t)$, which satisfy one of the two state merging rules, are merged together to become a merged state n_{merged} because of the State Merging Step in Model Enrichment Stage. We know that:

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

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

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

The state configuration in sliced model remains unchanged as $N_{\mathcal{L}}$.

Because of the CD-HD Step in General Iterative Slicing Stage, the parent node of $ss(t), ds(t), n_{merged}$[11], p must have been added to the slice set. Because of (33) and the CD-HD Step,

$$p \in N_{\mathcal{L}}, \forall p.(n_{merged} \xrightarrow{hd} p) \tag{35}$$

$$\text{Because (35), } AncestorTillLCA(ds(t), ss(t)) \subseteq N_{\mathcal{L}} \tag{36}$$

Because during the State Merging Step in Model Enrichment Stage, the merged state will contain all child regions of the original states if they are composite states. Therefore:

$$InitDesc(ds(t)) \subseteq N_{\mathcal{L}} \tag{37}$$

We apply the state transition rule in original model from Equation (10) and finds that all components of $entered(t)$ are all subsets of $N_{\mathcal{L}}$ as explained in (34), (37) and (36). Given that (8), we can deduce that $N' \cap \mathcal{L}$ remains the same subset relation with $N_{\mathcal{L}}$, as

$$N' \cap \mathcal{L} \subseteq N_{\mathcal{L}} \tag{38}$$

Next, we want to prove that $\sigma'(v) = \sigma'_{\mathcal{L}}(v) \forall v \in Rv$. When $t \notin T_{\mathcal{L}}$, the proof will be same as the cases in (2), (3), (4), (5) because of the DD Step. Therefore:

$$\sigma'(v) = \sigma'_{\mathcal{L}}(v) \forall v \in Rv \tag{39}$$

Given both (38) and (39), we can deduce that $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$ by projecting the transition t in the original model to ϵ in the sliced model.

[11] The two SNodes, $ss(t)$ and $ds(t)$, are merged only when they are at the same rank of state hierarchy; their merged node is therefore at the same rank of hierarchy. See Section 4.4.3.1 for further details.

The Case when the Transition is Part-of-Slice This proves for **Case** (7).

When $t \in T_{\mathcal{L}}$, according to the Transition-to-State Step in General Iterative Slicing Stage, $ss(t) \in S_{\mathcal{L}}, ds(t) \in S_{\mathcal{L}}$. The state configuration in original model changes in the same fashion as that in the sliced model. Because $t \in T_{\mathcal{L}}$, the value of any relevant variable in the sliced model will change in the same way as that in the original model. Therefore, $P((N', \sigma')) = (N'_{\mathcal{L}}, \sigma'_{\mathcal{L}})$ by projecting the transition t in the original model to t in the sliced model.

■

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

As mentioned in Section 5.2.3, each execution step involves a set of transitions which occur concurrently, denoted as:

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

In addition, we have proved that $P(t) = (\epsilon \vee t_{\mathcal{L}})$ in Lemma 1.

Moreover, according to Definition 5, each concurrent transition of an execution step e occurs in a distinct orthogonal region; they do not interfere each other.

Based all the above evidences, we can compose the projections of the transitions in e together to form $e_{\mathcal{L}}$:

$$P(e) = P\begin{pmatrix} t_1 \\ \vdots \\ t_k \end{pmatrix} = \begin{pmatrix} P(t_1) \\ \vdots \\ P(t_k) \end{pmatrix} = e_{\mathcal{L}} \tag{40}$$

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

In other words, an execution step e in the original model is projected to an executions step $e_{\mathcal{L}}$ in the sliced model. This means that if the snapshot in the original model is projected to the snapshot in the sliced model before e, then the snapshot in the original model is projected to the snapshot in the sliced model after e.

5.5.4 Simulation

The initial snapshot of the original model will be (N^I, σ^I). N^I will be the set of root state and all the default initial states of relevant descendant states of the root state, identified recursively. $\sigma^I(v)$ for all $v \in Rv$ will be at default or uninitialized value of each variable.

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[12]. Therefore, $N^I \cap \mathcal{L} \subseteq N^I_{\mathcal{L}}$.

In addition, we also know that the values of all relevant variables in the sliced model are at their default or uninitialized values, because no transitions have occurred yet when the model is in its initial state configuration. Therefore, $\sigma^I(v) = \sigma^I_{\mathcal{L}}(v) \forall v \in Rv$.

Because $N^I \cap \mathcal{L} \subseteq N^I_{\mathcal{L}}$ and $\sigma^I(v) = \sigma^I_{\mathcal{L}}(v) \forall v \in Rv$, we can conclude that the initial snapshot in the original model can be projected to the initial snapshot in the sliced model. This forms the **base case** of the simulation.

The **inductive case** is that if the snapshot in the original model is projected to the snapshot in the sliced model before e, then the snapshot in the original model is projected to the snapshot in the sliced model after e. This is already proved in Section 5.5.3.

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. We can therefore conclude that:

Theorem 1. *By simulating an original model inside a sliced model produced by FormlSlicer, an execution step in original model can always be projected into one execution step in sliced model.*

This completes our correctness proof for FormlSlicer.

As a summary, this chapter proves that the sliced model produced by FormlSlicer is correct. However, correctness is not a sufficient criterion for a good model slicer. In the next chapter, we will present the empirical evaluation on FormlSlicer to show that the sliced model is not only correct, but also useful.

[12] Recall from the sub-step 1 of "True Transitions Creation" in Section 4.4.3.2 that sometimes when the original default initial state is missing in the sliced model, FormlSlicer will search for the next part-of-slice state to be a new default initial state.

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 FormlSlicer, we must use an SPL model that represents requirements in a real-world domain; otherwise, it is not convincing to demonstrate the reduction effects of FormlSlicer. Because of this, the slicing example we use in Section 4.4 cannot be used in empirical evaluation.

It is not a trivial task to create such an SPL model in FORML with sufficient content. There are only two FORML models from Shaker's thesis (the original work in FORML [11])—the telephony case study and the automotive case study. The telephony case study involves communications among different products; that means 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 a partial feature model of *Autosoft*. A feature model is a tree that depicts the constraints among features in an SPL: a feature is dependent on another feature if the former is a child node of the latter

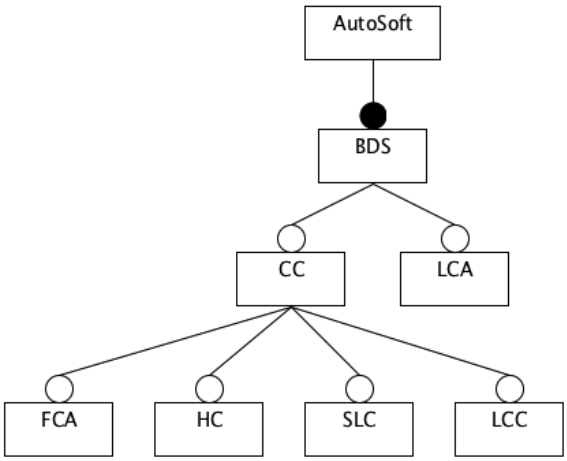


Figure 6.1: A Feature Model that Constraints the Relationships between Features in *Autosoft*

[34]. In FORML, the feature modules that are dependent on others are expressed in terms of 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].

Firstly, we create five FOSMs:

- Adaptive cruise control (ACC) (Figure A.2), which is composed from features BDS, CC and 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, these FOSMs have overlapping model elements among themselves. For example, each of all the above FOSMs has a transition "*t1: IgniteOn+(o) / a1: car:ignition:=on;*" from the state *off* to the state *on*.

In order to add some varieties to the model, we create another FOSM which represents an independent feature from the rest of features:

- Air quality system (AQS) (Figure A.7).

The feature of AQS is derived from one of Dietrich's mode-based state machines that model the behavioral requirements of production-grade automotive features [35]. We simplify the original feature in Dietrich's work by removing all the undefined functions and all variables that interact with other features related to air conditioning. In fact, the complexity of AQS feature does not matter; the more important fact is that AQS is independent from other five features and therefore will be "sliced away" when any of the other five features is chosen as the FOI.

Altogether, these six 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 for *Autosoft*. 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 75 states, 54 variables, 45 regions and 90 transitions. As FormlSlicer has multiple slicing processes[2], each considering a different feature as the FOI, we therefore have 6 different sliced models. Each sliced model consists of the original FOI and the ROS with a reduced size.

From Table 6.1, we can see that each sliced model has a reduced size as compared to the original model. On average, a sliced model has 45.8% of states, 51.2% of variables, 57.8% of regions and 37.6% of transitions of the original model. This is a significant reduction on the model size.

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

Model	States		Variables		Regions		Transitions	
	#	S/O	#	S/O	#	S/O	#	S/O
Original Model	75		54		45		90	
Sliced Model w.r.t. ACC	40	53.3%	33	61.1%	31	68.9%	41	45.6%
Sliced Model w.r.t. LCA	21	28.0%	16	29.6%	18	40.0%	13	14.4%
Sliced Model w.r.t. FCA	45	60.0%	38	70.4%	34	75.6%	45	50.0%
Sliced Model w.r.t. LCC	48	64.0%	37	68.5%	37	82.2%	50	55.6%
Sliced Model w.r.t. SLC	41	54.7%	35	64.8%	31	68.9%	43	47.8%
Sliced Model w.r.t. AQS	11	14.7%	7	13.0%	5	11.1%	11	12.2%
AVG		45.8%		51.2%		57.8%		37.6%

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

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

The first 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.11 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 that in the number of regions. This will be left for future work.

The second 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 is very small. The reason is that the feature of AQS concerns only the air pollution level within the car, whilst the other five features concern the vehicle's motion; in other words, it does not interact with all the other five features. Thus, the sliced model with respect to AQS contains only the FOI, which is the feature of AQS itself; all the other FOSMs are not present in the sliced model.

Another less extreme example is the feature of LCA. It concerns the vehicle's steering direction (e.g., monitoring the variable *car.steerDirection*), whilst the majority of features in the model (ACC, FCA and SLC) concerns the vehicle's moving speed and acceleration. Unlike AQS which has zero interactions with other features, LCA still shares some basic behaviors about the vehicle's steering directions with other features. Therefore, the sliced model with respect to LCA still contains a small portion of other FOSMs.

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 proposed slicing techniques, the various dependences' definitions and discussions on slicing correctness that are related to SBM slicing.

A Detailed Slicing Workflow
Chapter 3 and Chapter 4 propose a workflow of slicing on the behavior model in FORML. The workflow includes two tasks: the preprocessing task and the slicing task. The preprocessing task converts the original FORML semantics to a simpler intermediate representation—CFG—and computes three types of dependences within CFG. The slicing task is to fork off multiple processes, each working on a different feature as the FOI; this is the Multi-Stage Model Slicing Process.

Correctness
Chapter 5 presents the proof to show the correctness of the proposed slicing workflow. It proves that an execution trace in the original model can be simulated by at least

one execution trace in the sliced model, which is produced by the proposed slicing workflow.

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

As a nutshell, this thesis proposes a unique slicing workflow on a hierarchical and concurrent state-based model for feature-oriented SPL requirements, 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.

1. Compared to other existing SBM slicing approaches, the target model of this slicing workflow is much more complex. We devise different ways to tackle the different challenges of the model.
    - The challenge of having hierarchical construct is solved by using hierarchy dependence (HD). In HD, each state is mapped to its parent state. Therefore, the difficult task of slicing a hierarchical state machine is decomposed into many simple tasks of slicing on flat state machines.
    - The challenge of having concurrent construct is also solved by using HD. 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 "inState" expression in guard condition is solved by making it as a subtype of data dependence (DD).
    - The challenge of having cross-hierarchy transitions is solved by preserving these transitions. Because the Multi-Stage Model Slicing Process is designed not to change the state hierarchy, these transitions can be correctly preserved in the sliced model.
2. We design a unique workflow by coalescing different slicing techniques from multiple literature studies, including:

- Korel et al.'s state merging rules [5],
- Various existing algorithms in data dependence [5, 19],
- Ranganath et al.'s definition and algorithm in control dependence [21],
- Ojala's use of CFG to perform slicing on an SBM [18],
- and Kamischke et al.'s use of a model enrichment step in the slicing algorithm which incrementally adds the model elements [17].

3. We make automation of slicing easier by transforming each transition label in FORML—which contains complex information on WCE, guard and WCA—into two groups of variables—monitored variables and controlled variables, and store them in a CFG node.

4. We conduct a proof to show that the workflow we have proposed can produce correct sliced models.

5. Lastly, we implement a tool to show that this slicing workflow is feasible.

7.1.2 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 (Chapter 6). 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[1]. 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 only use the state merging rules that are proved to preserve the precision of the original model in Korel et al.'s model slicing technique [5].

[1]See Chapter 2.2.5 for more details on how researchers define the correctness property of a model slicer.

7.2 Future Work

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

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

There are mainly two differences between FormlSlicer's input model and the FORML model:

- FormlSlicer's input model does not contain presence conditions;
- FormlSlicer's input model does not contain the world model.

In Section 3.3, we discuss that the model used by FormlSlicer does not contain presence conditions and therefore it is a 150% model which represents a product with all the features. Because this model does not consider the relationship among features (e.g., whether two features are mutually exclusive), it may represent an invalid product. A real SPL model must contain presence conditions. A presence condition is a boolean variable that is named after its corresponding feature; it is usually placed in the guard of a transition to indicate that this transition can only be triggered when the particular feature is present in the product [11].

To consider presence conditions, the workflow in FormlSlicer needs to be improved. There are many possible ways to tackle this. One possible way is to invent a new dependence relationship to represent the relationship among features; in this case, when a transition's guard expression contains a presence condition of feature F, we establish this transition and F so that whenever the transition is selected into the slice set, then F is selected into the slice set. Another possible way is to add a new transition that monitors the presence condition and moves from a new default initial state to the composite state representing an FOSM; then we replace all presence conditions to become *inState* expression (e.g. a presence condition F is transformed into *inState(F)* and F is the composite state representing the FOSM of feature F). At this moment, we do not implement a way to tackle the presence conditions and this will be left for future work.

We also mentioned in Section 3.3 that FormlSlicer does not perform slicing on the world model. Because the theme of this thesis is focused on 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.

Besides the two above-mentioned differences between FormlSlicer's input model and the FORML model, there are also some minor differences. 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 in a SPL; 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 [11] is not suitable for our empirical evaluations. This will be left for future work.

7.2.2 Improving the Multi-Stage Model Slicing Process

Currently, the Multi-Stage Model Slicing Process we propose is still conservative.

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.11), 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[2]. 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.

7.2.3 Customization of Slicing in FormlSlicer

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

[2]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.

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.

7.2.4 Making the Correctness Proof More Rigorous

Currently, many definitions in the correctness proof are not formalized strictly because of their complexities. Some examples are shown below.

- The state transition rule (Section 5.3) is not formalized because of its complexity. In this thesis, we use six components to represent the state transition rule, including $ss(t)$, $AncestorTillLCA(ss(t), ds(t))$, $Descendants(ss(t))$, $ds(t)$, $AncestorTillLCA(ds(t), ss(t))$ and $InitDesc(ds(t))$. Only $ss(t)$ and $ds(t)$ are clearly defined as the source state and destination state of the transition t. The other four components are described using plain English.
- The state machine used in FOSM is not formalized. In Section 5.2.2, we introduce this state machine in a paragraph. But it is not trivial to formalize it as a tuple, because this state machine contains many model components and each component is in a relationship with one another (e.g., state and region forming a containment relation).

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 six FOSMs—ACC (Figure A.2), FCA (Figure A.3), LCA (Figure A.4), LCC (Figure A.5), SLC (Figure A.6) and AQS (Figure A.7).

Figure A.8 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.9, Figure A.10, Figure A.11 and Figure A.12. The FOSM of feature AQS is not present in the sliced model.

Figure A.13 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.14, Figure A.15, Figure A.16 and Figure A.17. The FOSM of feature AQS is not present in the sliced model.

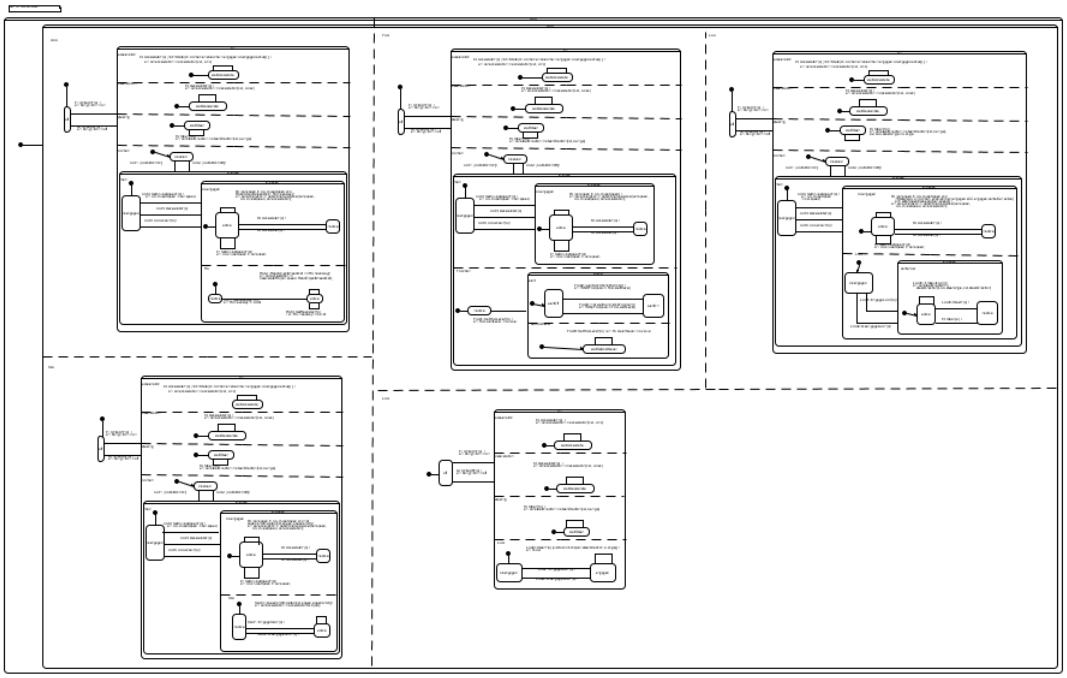


Figure A.1: The Original Automotive Model

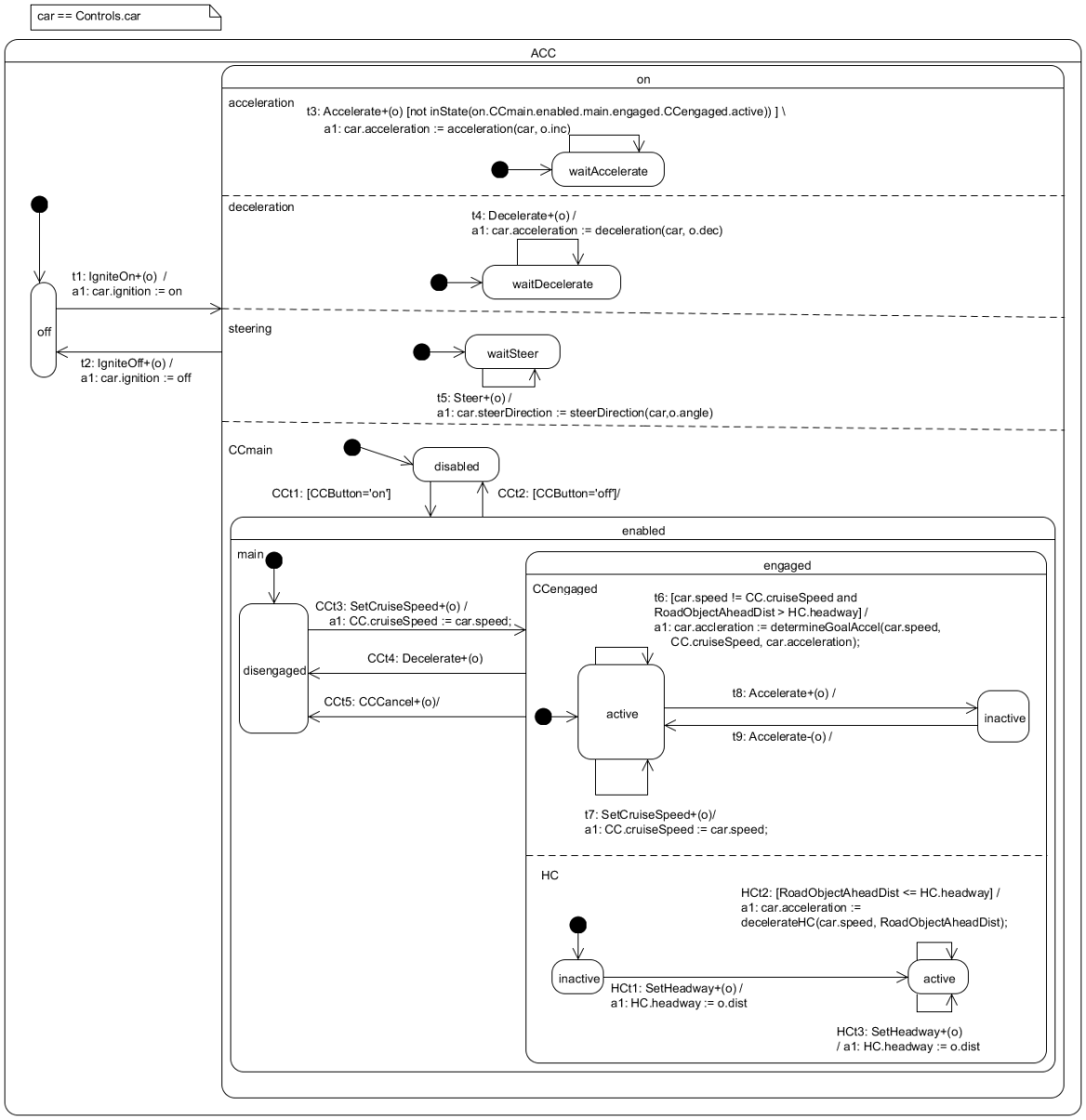


Figure A.2: Original ACC feature of the Automotive Model

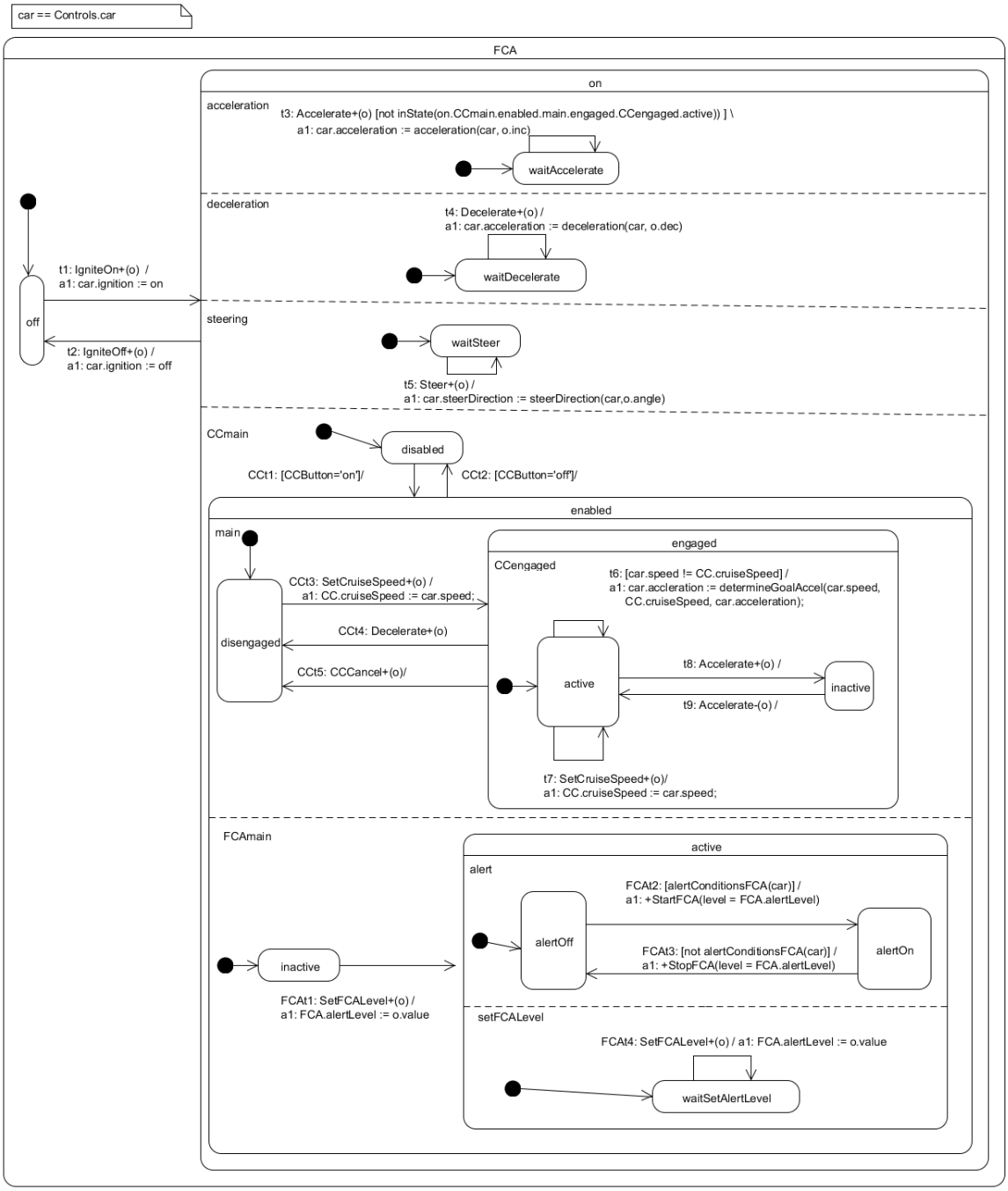


Figure A.3: Original FCA feature of the Automotive Model

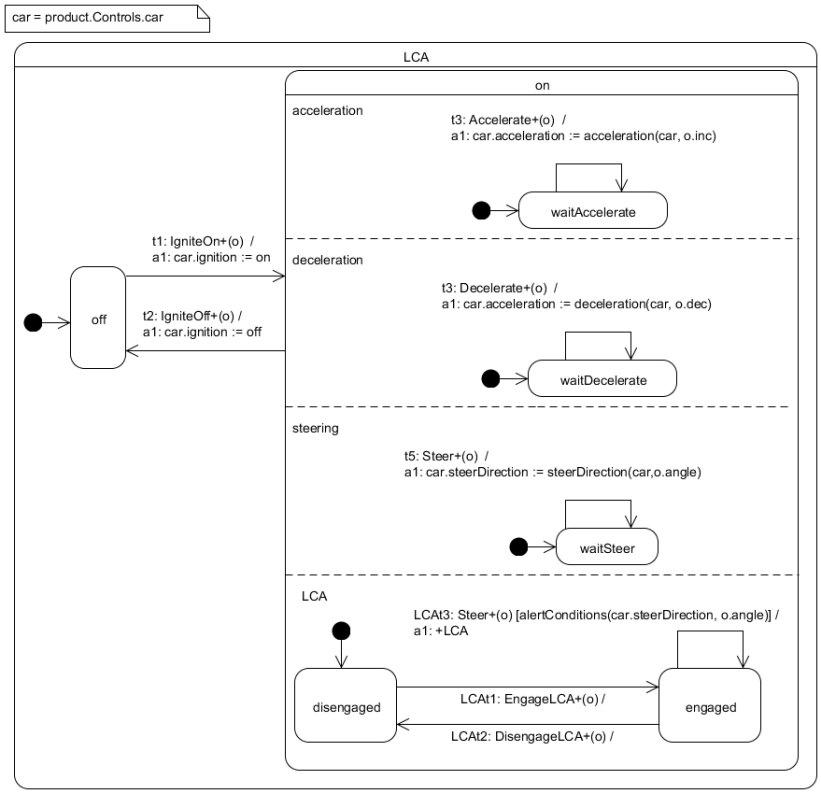


Figure A.4: Original LCA feature of the Automotive Model

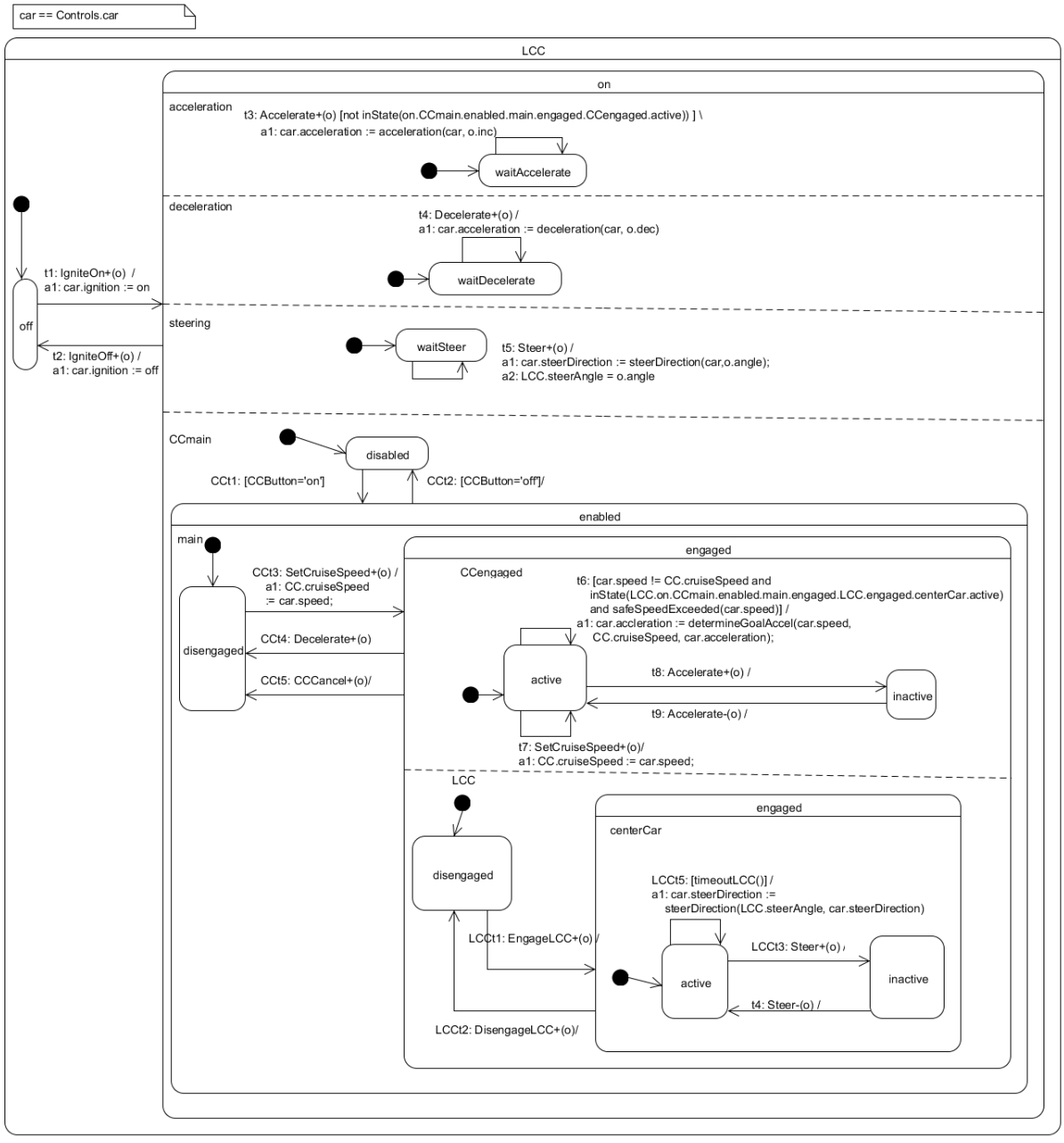


Figure A.5: Original LCC feature of the Automotive Model

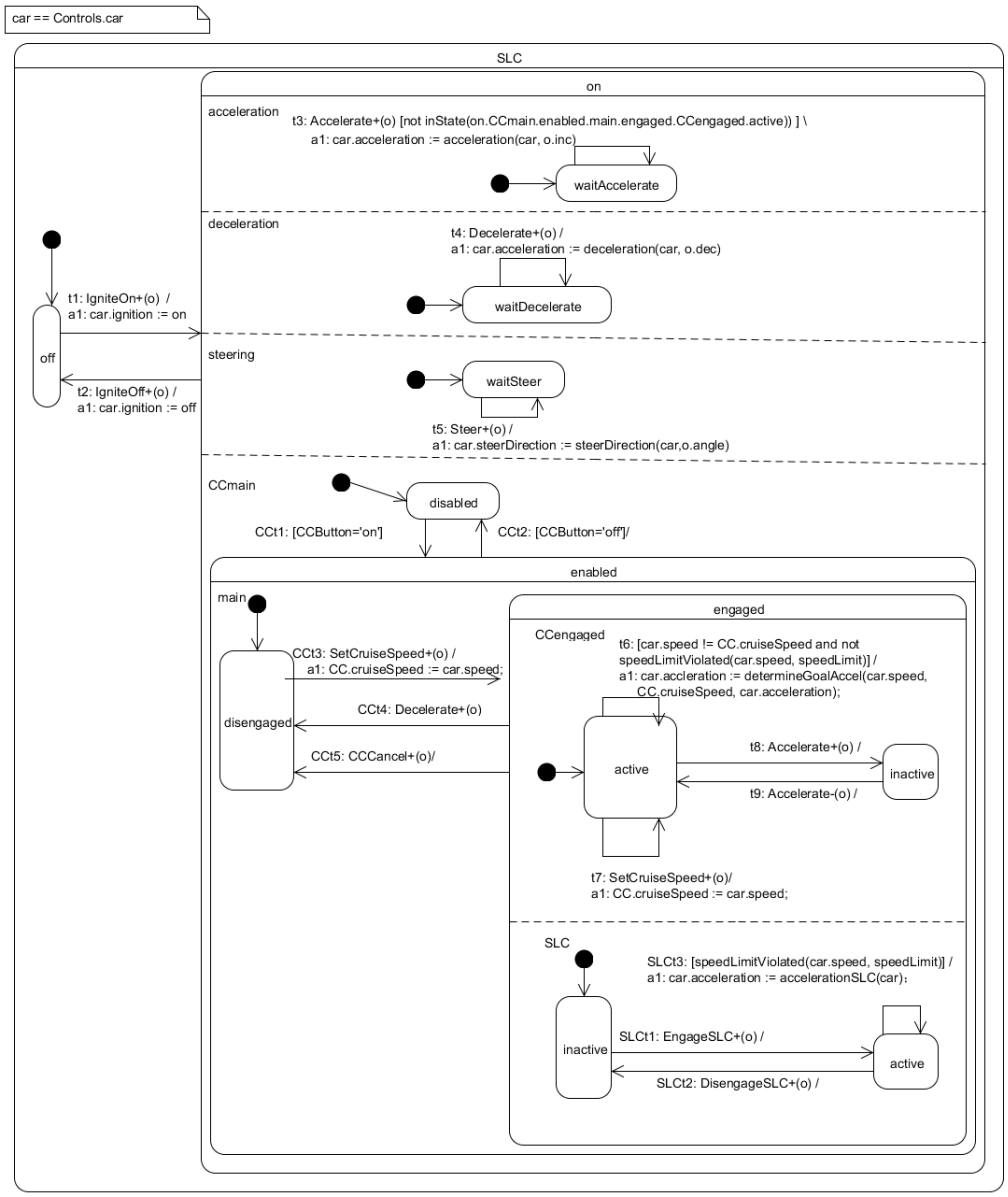


Figure A.6: Original SLC feature of the Automotive Model

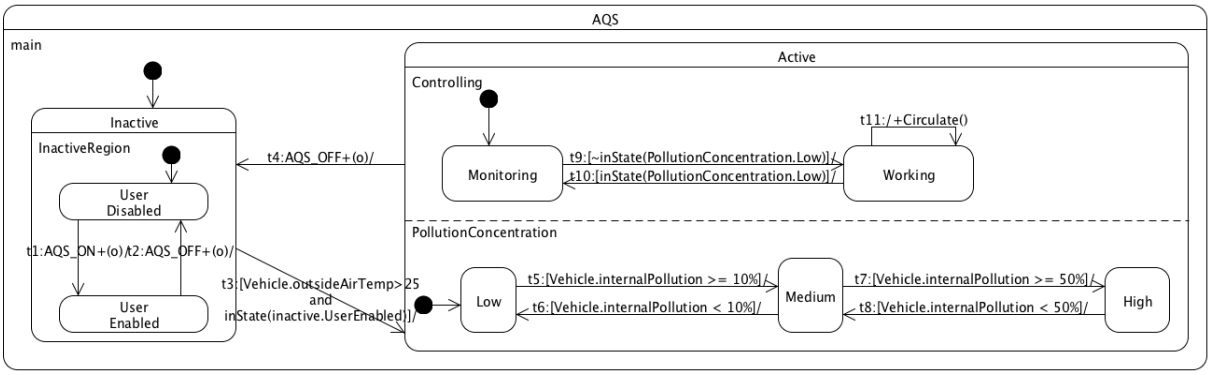


Figure A.7: Original AQS feature of the Automotive Model

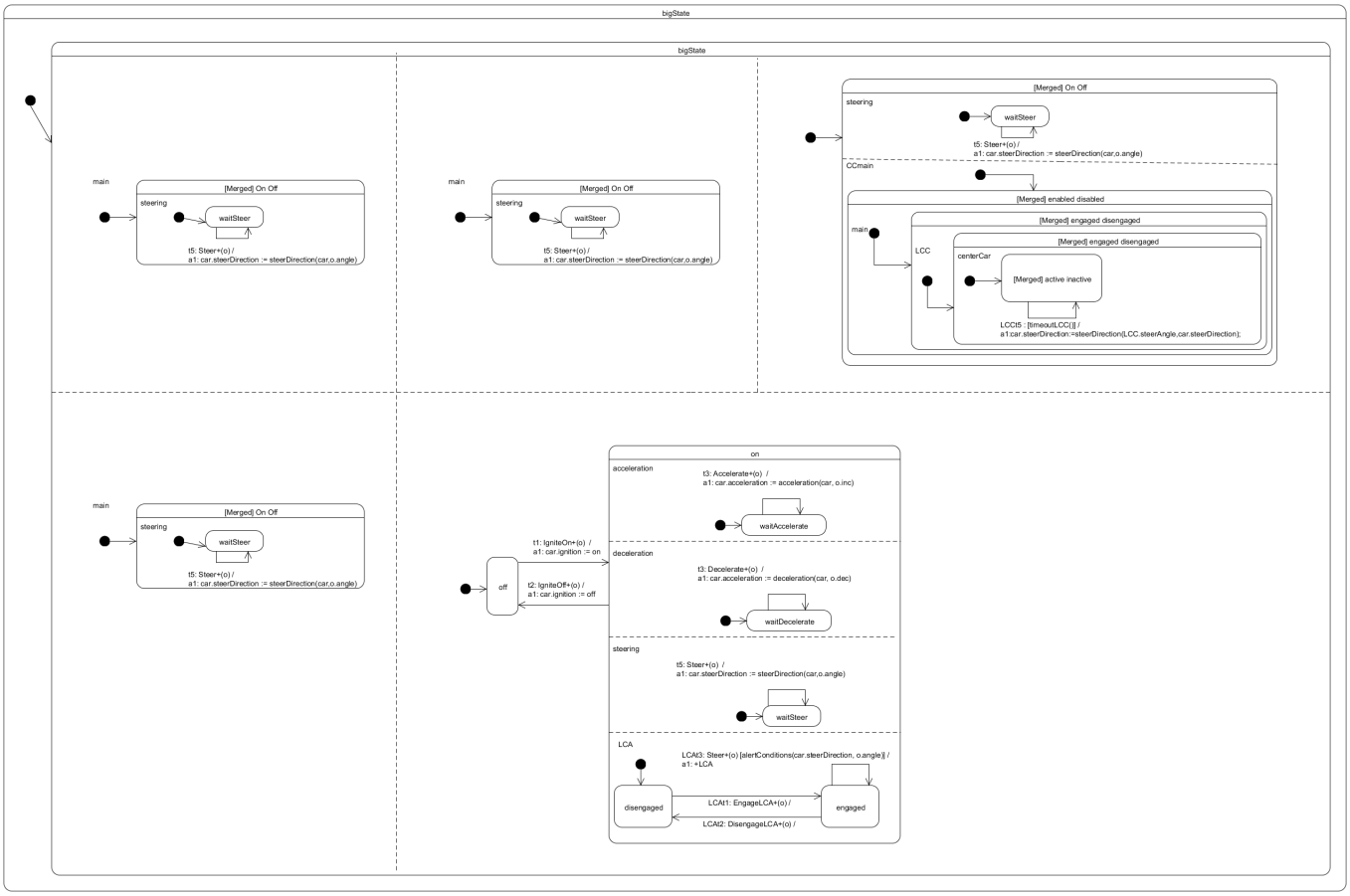


Figure A.8: The Sliced Model w.r.t. LCA

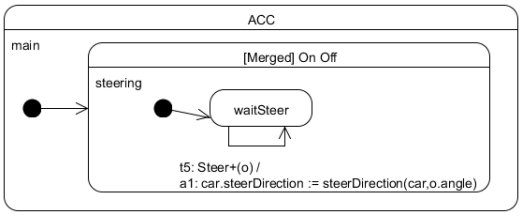


Figure A.9: Sliced ACC feature w.r.t. LCA

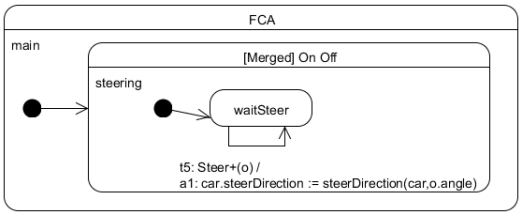


Figure A.10: Sliced FCA feature w.r.t. LCA

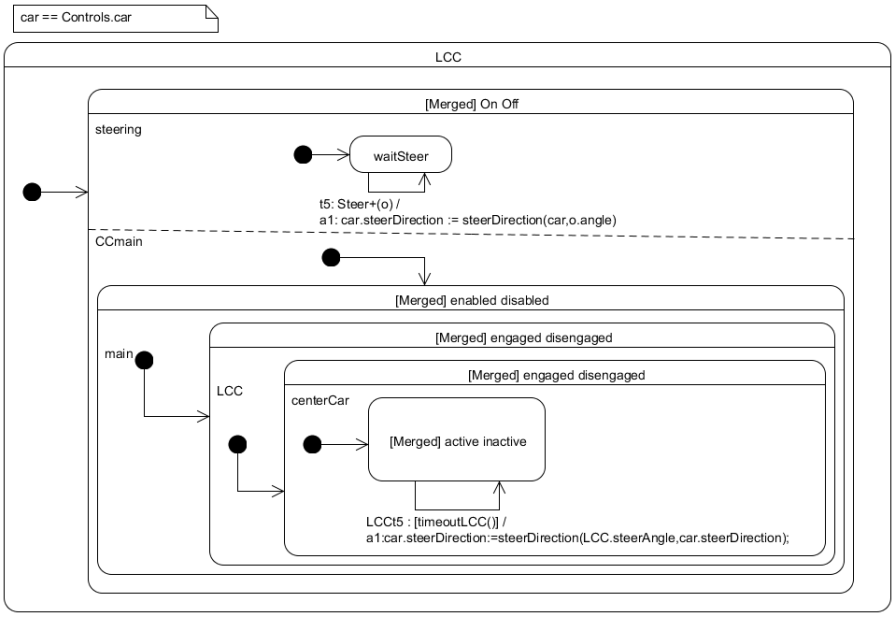


Figure A.11: Sliced LCC feature w.r.t. LCA

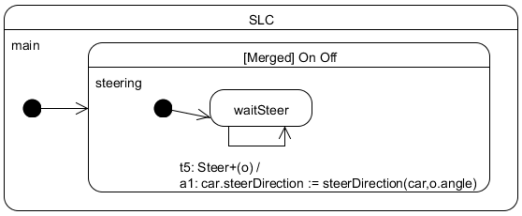


Figure A.12: Sliced LCC feature w.r.t. LCA

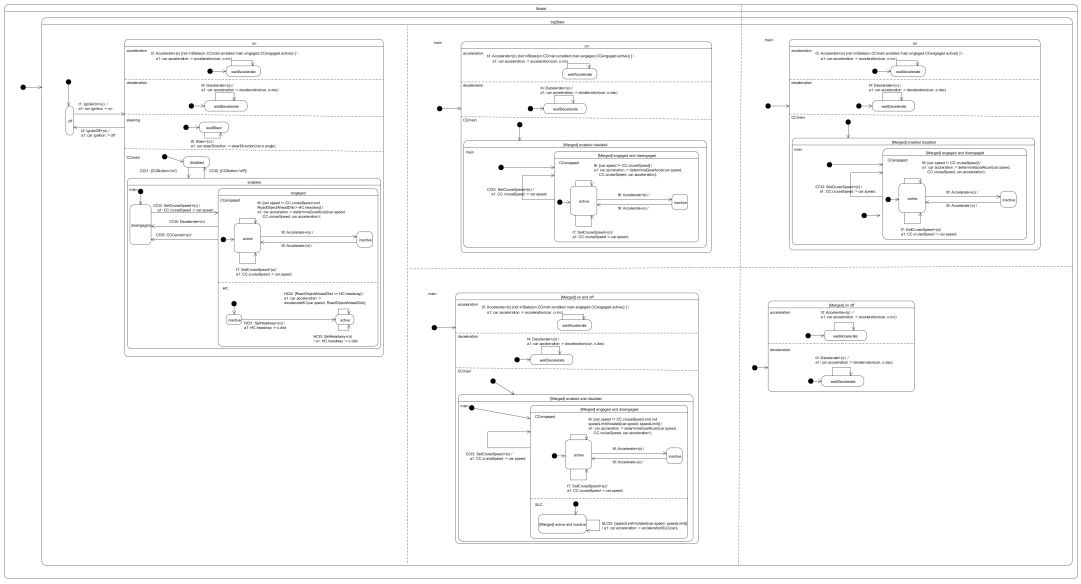


Figure A.13: The Sliced Model w.r.t. ACC

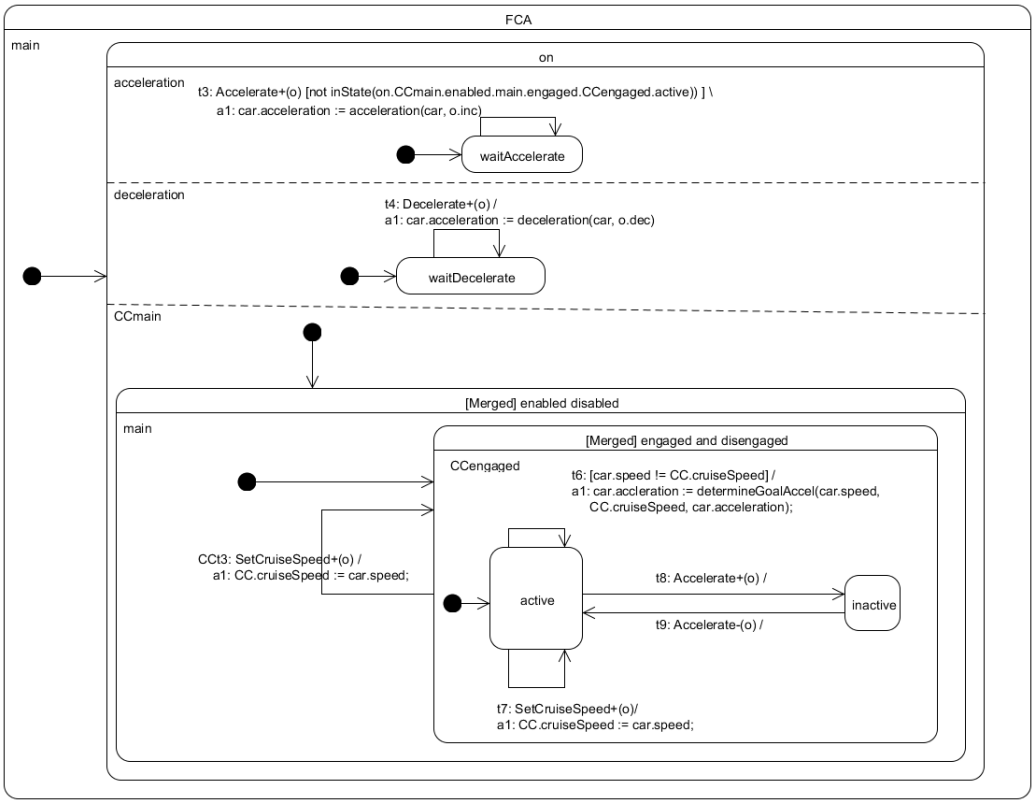


Figure A.14: Sliced FCA feature w.r.t. ACC

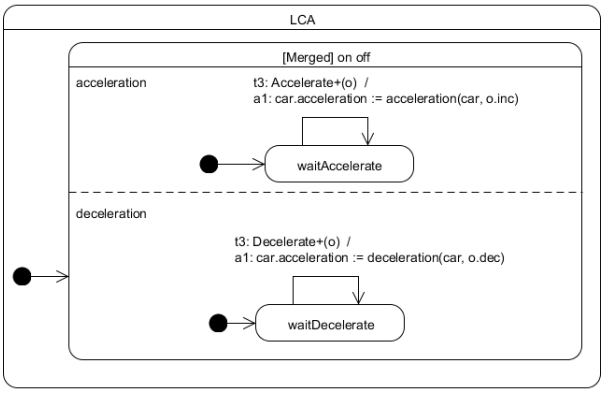


Figure A.15: Sliced LCA feature w.r.t. ACC

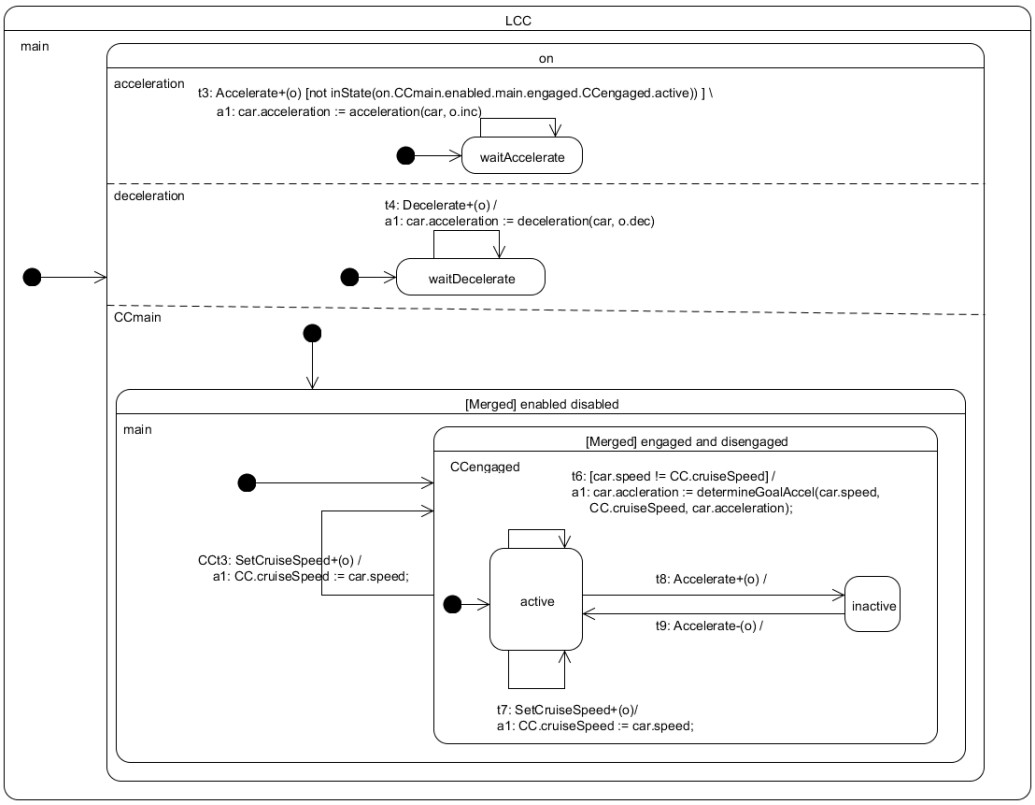


Figure A.16: Sliced LCC feature w.r.t. ACC

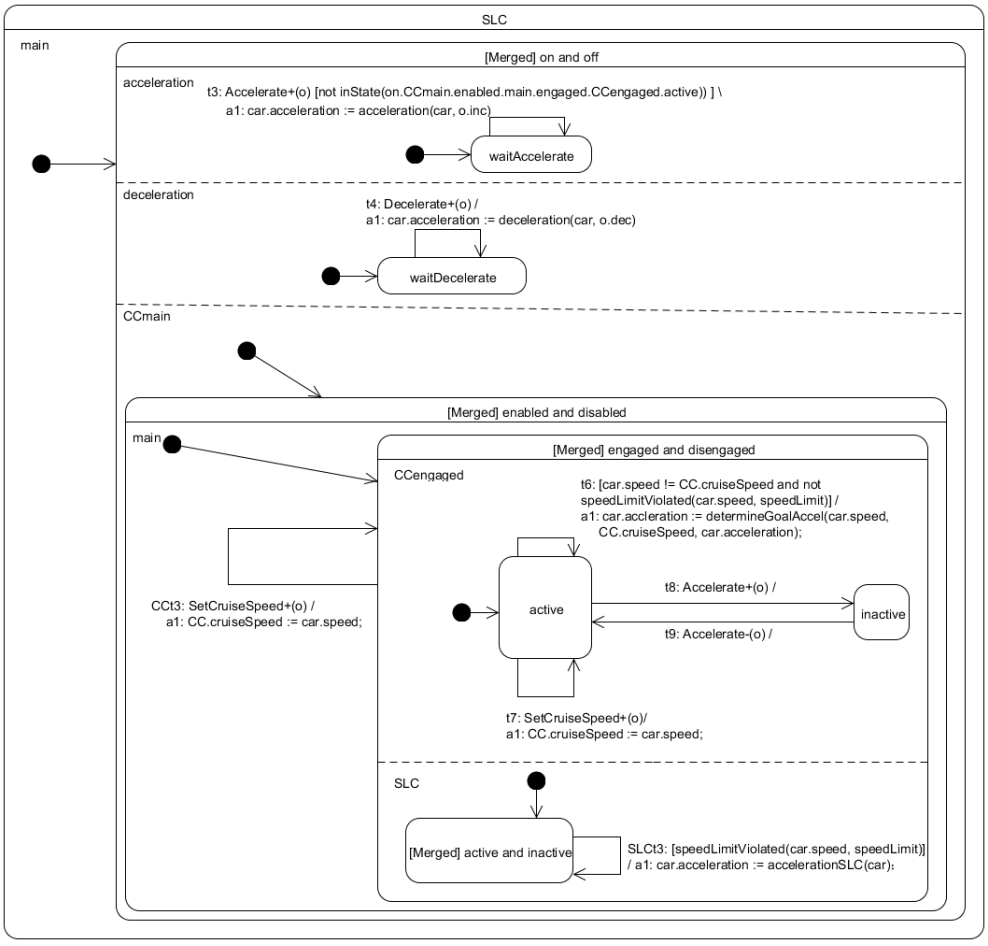


Figure A.17: Sliced SLC feature w.r.t. ACC

Appendix B

Stress Test

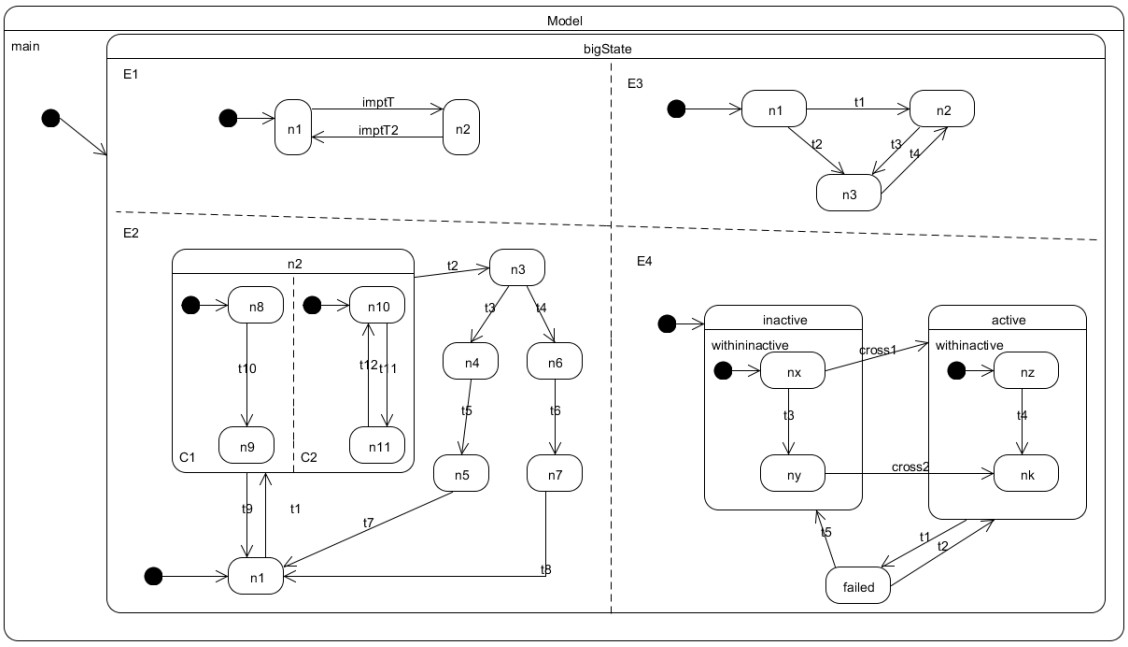


Figure B.1: The Original Model

The original model, as shown in Figure B.1, consists of four FOSMs—*E1* (Figure B.2), *E2* (Figure B.3), *E3* (Figure B.4) and *E4* (Figure B.5).

Figure B.6 shows the sliced model with respect to *E1*.

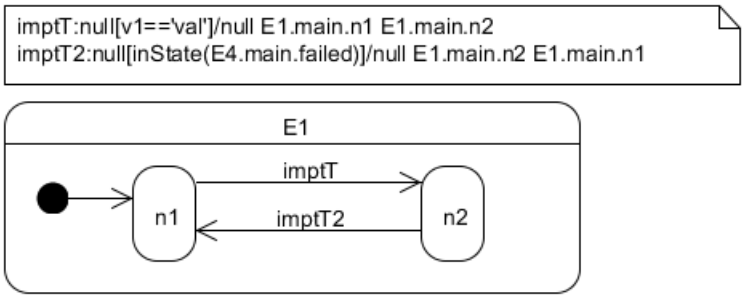


Figure B.2: Original *E1*

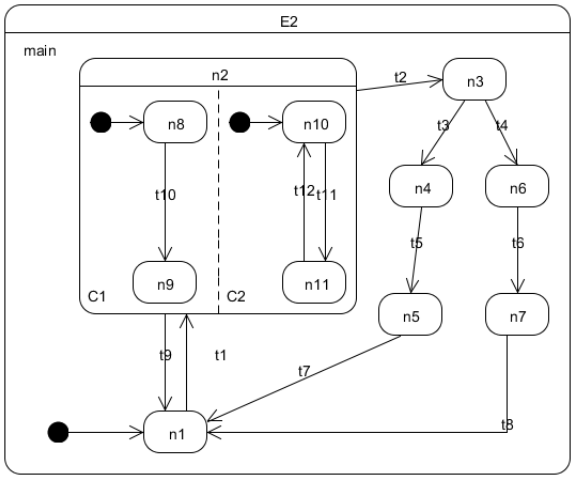


Figure B.3: Original *E2*

The sliced model consists of three FOSMs in the ROS, as shown in Figure B.7, Figure B.8 and Figure B.9.

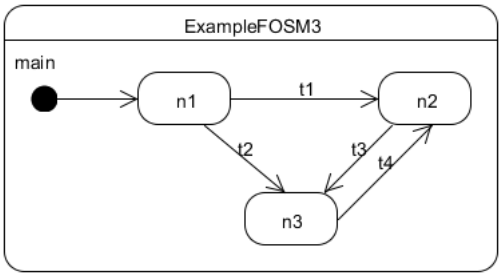


Figure B.4: Original *E3*

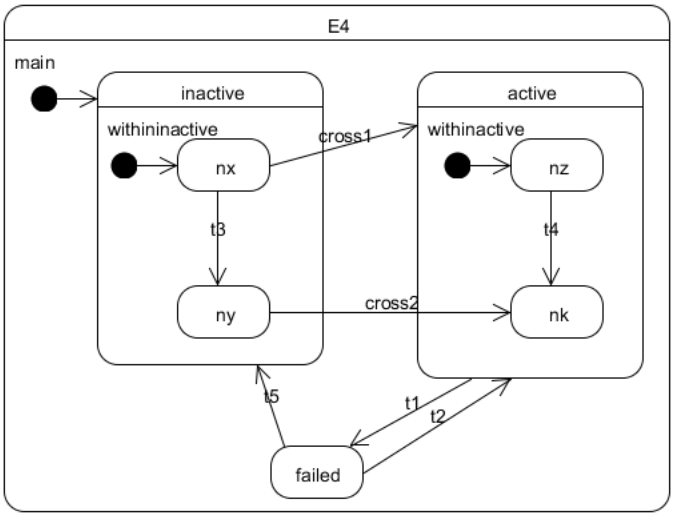


Figure B.5: Original *E4*

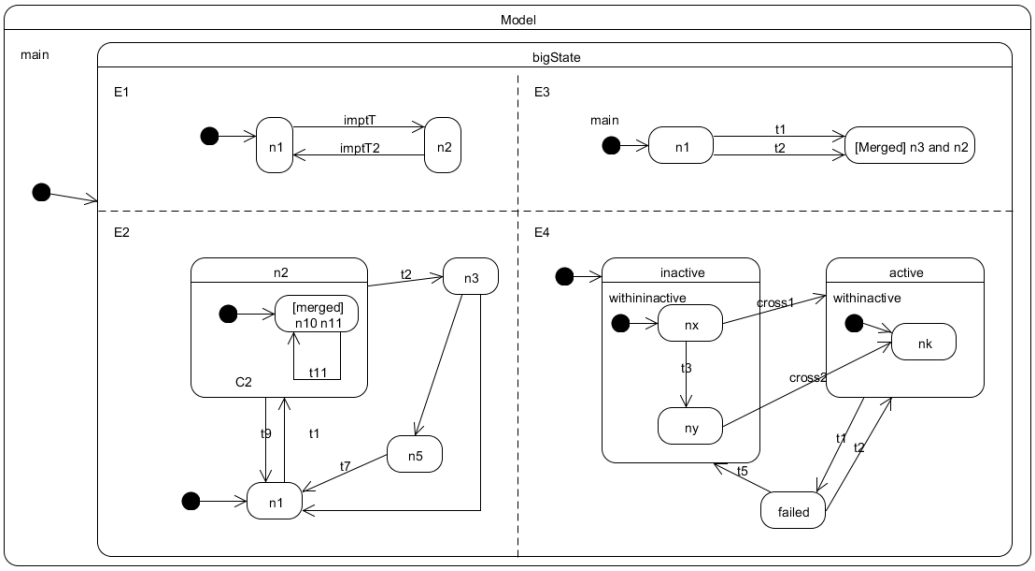


Figure B.6: The Sliced Model w.r.t *E1*

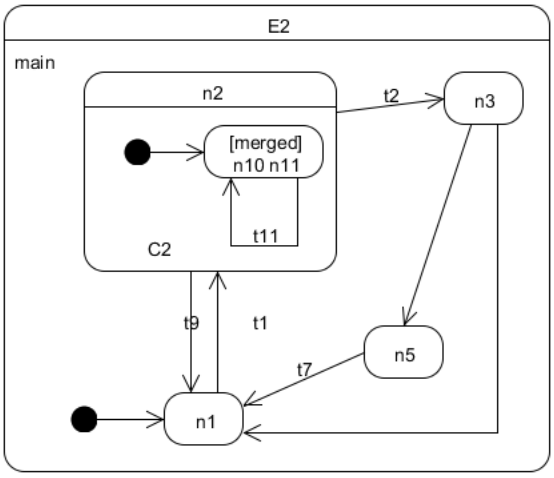


Figure B.7: Sliced *E2* w.r.t. *E1*

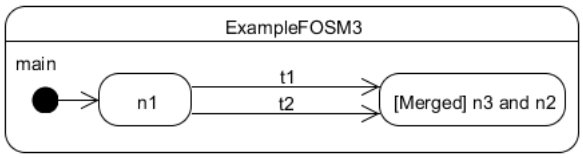


Figure B.8: Sliced *E3* w.r.t. *E1*

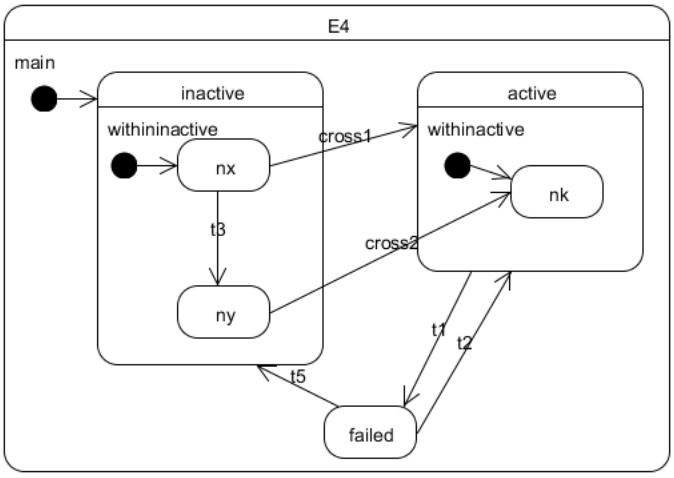


Figure B.9: Sliced *E4* w.r.t. *E1*

Appendix C

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 C.1: Supporting Function for CD Algorithm: HasReductionOccursBefore

```
Function HasReductionOccursBefore (targetIndex) :
    if p[targetIndex]=="!" then return true
    else return false
end
```

Algorithm C.2: Supporting Function for CD Algorithm: IsReachableFromPartialPaths

```
Function IsReachableFromPartialPaths (targetIndex) :
    if p[targetIndex] == null OR p[targetIndex] == "!" then return false;
    set originalPath := p[targetIndex];
    set paths := ReducePaths (originalPath);
    set resultString := paths.join(";");
    if originalPath != null AND resultString == null then
        set p[targetIndex] := "!";
        return false
    end
    set p[targetIndex] := resultString
    return true
end
```

Algorithm C.3: Supporting Function for CD Algorithm: ReducePaths

```
Function ReducePaths (originalPath) :
    set paths := array.split(originalPath, ";");
    if paths.length ≤ 1 then return paths;
    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+1 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
    return paths
end
```

Algorithm C.4: Supporting Function for CD Algorithm: UnionPathHappens

```
Function UnionPathHappens (targetNodeIndex, prevNodeIndex) :
    set oldTargetNodeIndex := p[targetNodeIndex];
    if hasReductionOccursBefore (prevNodeIndex) == true then
        set p[targetNodeIndex] := "!";
        return true
    end
    if p[prevNodeIndex] != null then
        if p[targetNodeIndex] != null then
            set prevNodeIndexSet := array.split( p[prevNodeIndex], ";");
            set targetNodeIndexSet := array.split( p[targetNodeIndex], ";");
            foreach prevP in prevNodeIndexSet do
                set duplicate := false;
                foreach targetP in targetNodeIndexSet do
                    if prevP starts with targetP then
                        reset duplicate := true;
                        break
                    end
                end
                if duplicate == false then APPEND prevP to p[targetNodeIndex];;
            end
        else
            set ptargetNodeIndex := p[prevNodeIndex];
        end
    else
        set p[targetNodeIndex] := null;
        return true
    end
    if p[targetNodeIndex] == oldTargetNodeIndex then
        return false
    else
        return true
    end
end
```

Algorithm C.5: Supporting Function for CD Algorithm: ExtendPath

```
Function ExtendPath (srcIndex, newSubPath) :
    if p[srcIndex] == null then return newSubPath;
    if hasReductionOccursBefore (srcIndex) == true then return "!";
    set srcIndexArr := array.split( p[srcIndex], ";")
    foreach i from 1 to srcIndexArr.size do
        APPEND newSubPath to srcIndexArr[i];
    end
    return 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] Torben Amtoft, Kelly Androutsopoulos, and David Clark. Correctness of slicing finite state machines. *RN*, 13:22, 2013.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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

[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] 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.

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

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

[17] 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.

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

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

[20] 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.

[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, 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] David Dietrich. *A Mode-Based Pattern for Feature Requirements, and a Generic Feature Interface*. PhD thesis, University of Waterloo, 2013.

[36] 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.