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

by

Xiaoni Lai

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

Waterloo, Ontario, Canada, 2014

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

TODO: modifyModel slicing is an essential process to support compositional verification of feature-oriented requirements models. In order to detect unintended feature interactions, the composed Software Product Line (SPL) model is sliced against all monitored variables of the feature of interest, before passing to a model checker. This makes model checking perform more times, each time with a much smaller input, and thereby encourages concurrency while maintaining constraints raised from feature interactions. Here I would present you the FORML Slicer—a model slicer that serves the purpose of optimizing model checking on feature interactions. It incorporates the highly expressive nature of Feature-oriented Requirements Modelling Language (FORML), calculates four types of dependencies within the corresponding Control Flow Graph, and uses a unique slicing approach to output a sliced model, in which executability and well-formedness is maintained.

Acknowledgements

Table of Contents

List of Tables

List of Figures

List of Algorithms

Glossary

ACC Adaptive Cruise Control. x, xi, 75, 80–83

BDS Basic Driving Service. 9

CC Cruise Control. 9

CD Control Dependency. 5, 24, 25

CFG Control Flow Graph. 4, 5, 15, 17, 18, 20, 32, 40

DD Data Dependency. 5, 10, 12, 13, 17, 21, 22

EFSM Extended Finite State Machine. 7

FCA Forward Collision Alert. x, xi, 76, 80, 81

FOI Feature of Interest. x, 3–6, 11, 12, 14, 15, 22, 32–34, 46, 59, 70, 71

FORML Feature-oriented Requirements Modelling Language. iii, 4–6, 8, 9, 11, 12, 15, 17, 19, 20, 34, 41, 49, 69

FOSD Feature-oriented Software Development. 1, 2, 8

FOSM Feature-oriented state machine. ix, 4–6, 8–12, 15, 17–20, 22, 33, 34, 46, 47, 49, 51, 59, 69, 70

HC Headway Control. 2

HD Hierarchy Dependency. 5, 17, 21

LCA Lane Change Alert. x, xi, 77, 80–82

LCC Lane Centring Control. x, xi, 78, 80–82

ROS Rest of System. 3, 5, 6, 12, 14, 15, 33, 34, 46, 47

SLC Speed Limit Control. x, xi, 2, 79, 83

SNode A Node in CFG representing a state in FORML model. 19, 20, 32, 33, 40

SPL Software Product Line. iii, 1–5, 8

TNode A Node in CFG representing a transition in FORML model. 19, 20, 32, 33, 40

WCA World Change Action. 10–13, 19, 53

WCE World Change Event. 10–12, 19, 53

Chapter 1

Introduction

1.1 Background

The background section presents the motivation of feature-oriented model slicing—the core of this research project. It briefly explains the importance to detect unintended feature interactions that exist in most Software Product Lines. Then it presents several strategies using model checking for product-line analysis; this includes the strategy of compositional verification, in which model slicing plays an essential role.

1.1.1 Feature Interactions in Software Product Line Requirements

Many organizations develop a family of similar software systems in the same domain, called a Software Product Line (SPL). For example, an automotive manufacturer will design a series of new vehicle models that are close variants of one another. These software systems have different combinations of a fixed set of features in the SPL. 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 share most of the other features, such as the Traction Control feature.

In light of this, the Feature-oriented Software Development (FOSD) paradigm is usually adopted in SPL requirements. This paradigm 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 [1].

One well-known challenge of the FOSD paradigm in SPL requirements is managing feature interactions. In software requirements, features are usually modelled 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 onto a system, different features can influence one another in determining the overall properties and behaviours of their combinations [2]. In a single system, such feature interactions can still be pre-determined during modelling stage. But in an SPL, different system variants can be equipped with different subsets of all features, resulting in a possibly exponential number of feature combinations; therefore, detecting feature interactions in an SPL is a complicated task.

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 behaviour of the system and might pose potential hazards to the system users. Consider the Headway Control (HC) feature and Speed Limit Control (SLC) feature, if the vehicle approaches an obstacle at a speed faster than speed limit, both features will simultaneously send messages to the actuators responsible for controlling vehicle acceleration. If their messages are different, the behaviour of the vehicle is undefined and acceleration may be set to an unpredictable value [3].

To understand feature interactions, we can visualize the SPL as a set of features with a shared pool of variables in the environment. Each feature is monitoring a subset of variables (*i.e.*, monitored variables), so that the values of these variables can affect the feature behaviour. Each feature may also control a subset of variables (*i.e.*, controlled variables), so that the values of these variables are modified according to the feature behaviour. In the previous example of feature interaction, we can say that the acceleration is the variable simultaneously controlled by both the HC feature and SLC feature. The concepts of controlled and monitored variables will be further elaborated in Section 3.2.

1.1.2 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 understanding and managing feature interactions. Formal method techniques like model checking have been proposed to detect feature interactions in

product lines. In the past, several strategies for product-line analysis have been proposed, in particular, family-based, product-based and feature-based strategies [4]. Family-based strategy exploit commonality among products in an SPL and can deliver sound and complete analysis results, but is often computationally infeasible as the whole SPL is too huge to be analysed in one go. Product-based strategy performs model checking on individual products in an SPL and thus faces severe scalability problems for larger sets of products. Feature-based strategy ignores interactions across feature boundaries; it is fast, 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 minimal Rest of System (ROS) (*i.e.* fragments of other features that are relevant to the feature). In each iteration of analysis, the Feature of Interest (FOI) (*i.e.* the feature which is being verified in that iteration) and minimal ROS are fed into 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: linear number of models to check, parallel verification possible; also, feature interactions are taken into considerations and this can yield more complete verification result.

Now, how can we create fragments of a feature in ROS with respect to the FOI?

Model Slicing used in Compositional Verification

Model slicing evolves from program slicing, which, informally defined, is a source code analysis technique used to identify a minimal sub-program based on a user-specified slicing criterion. That sub-program, called a "slice", is an independent program that represent faithfully the original program within the domain of slicing criterion [5]. Model slicing is similar to program slicing; but it applies on model, instead of code.

The concept of slicing is highly applicable to our task of creating fragments of a feature in ROS, because we need to guarantee that the fragments faithfully simulate the subset of behaviour of the original feature that can influence the FOI—which can be deemed as the slicing criterion. Only when this condition is met, the feature interactions between the FOI and the features in ROS can be included in model verification, and thus the combined verification results from individual feature verification can be complete.

The "model" used in model slicing can refer to a wide range of distinct UML representations, ranging from class diagram (describing static structure) to state-based diagrams (describing behaviour). In this research project, we have chosen a modelling notation that can

express the behaviour of features in a SPL, that is the Feature-oriented Requirements Modelling Language (FORML) [6]. FORML notation is based on standard software-engineering notations (e.g., UML class and state-machine models, feature models), and therefore many existing slicing practices on class diagrams and state-based diagrams can be applied in this work.

1.2 Thesis Overview

This thesis introduces a model slicing tool that aims to slice an SPL model consisting of many features, with respect to one feature (*i.e.* the FOI) at a time. The tool is called **FormlSlicer** because it is slicing on a model represented using FORML modelling language.

FormlSlicer undergoes multiple stages to produce the feature fragments (*i.e.* the slice). In a preprocessing stage, it needs to parse and process the whole original model, transforms the transitions and states that exist within many Feature-oriented state machines into a Control Flow Graph (CFG) so that it can compute the dependency relationship among the nodes easily. These dependencies serve like pre-computed "dictionary" that can be repeatedly looked up by the slicing processes. Afterwards, the main slicing processes are initiated, each process treats one feature as the FOI, and slices the rest of Feature-oriented state machine (FOSM) against the FOI. Since the slicing process starts from emptiness and gradually include components from the FOSM into slice, the resulting sliced FOSM might be a disconnected state-based machine; thus the slicing processes will undergo an "enrichment" stage to make the sliced FOSM an executable and complete state-based machine.

1.2.1 Thesis Statement

A Feature-oriented state machine (FOSM) that model the behaviour of a feature in a Software Product Line (SPL) can be sliced with respect to one Feature of Interest (FOI), by following the below multiple stages:

- preprocessing stage, that parses and converts the FOSM into Control Flow Graph (CFG) followed by dependencies computations,
- slicing stage, that identifies components of CFG that should be included to be part-of-slice,

- and finally enrichment stage, that transforms the fragmented CFG into a complete FOSM again;

this is to produce sliced features in the Rest of System (ROS):

- that are executable on their own,
- that maintain the constraints raised from feature interactions with the FOI,
- that are as small as possible,
- and therefore can serve as a minimal context for the FOI that will be verified by model checker against a set of related safety properties;

this feature-based slicing and verification process is repeated for all features, so as to support compositional verification of the whole SPL.

1.3 Chapter Summary

The thesis explains all the work devoted in creating FormlSlicer.

Chapter 2 shows the literature survey on a wide range of existing model slicing techniques.

Chapter 3 explains the semantics of Feature-oriented Requirements Modelling Language (FORML), which is the modelling language used in this work to express the feature modules in an SPL, as well as other important concepts used extensively throughout the work, like how monitored variables and controlled variables are extracted from the feature modules in FORML.

Chapter 4 is the main chapter in the thesis. It elaborates the internals of FormlSlicer. The chapter is divided into two major sections.

- Section **??** explains how the feature modules in FORML are parsed and converted into CFGs, which is an easier structure for computation of different types of dependencies. This section explains three different types of dependencies—Hierarchy Dependency (HD), Control Dependency (CD) and Data Dependency (DD)—that are important in FormlSlicer. The section describes the algorithms on how these dependencies are computed. The computed dependencies are shared by all slicing processes, which execution is described in the next Section.

- Section 4.4 describes what each slicing process does in order to slice the FOSMs in the ROS with respect to a specific FOSM (*i.e.* the FOI). This section explains how the Variable Extractor in FormlSlicer does to select the initial nodes from the FOSMs to become part-of-slice. Then it elaborates the details of multiple stages of the slicing process, including the General Iterative Slicing Stage that utilizes the dependencies to select more nodes to become part-of-slice, and the final Model Enrichment Stage that de-fragment all resulting FOSMs to make them complete state-based machines that are executable.

Chapter 5 is the theoretical evaluation of FormlSlicer. It takes on an abstract approach to prove the correctness of FormlSlicer. By using mathematical induction, it shows that on each step of execution, the sliced model is always in sync with the original model, and therefore proves how the sliced model simulates the original model.

Chapter 6 is the empirical evaluation of FormlSlicer. It derives the FORML models in Automotive case study from [7] and [8] and uses them as slicing targets. FormlSlicer produces satisfactory slicing results.

Chapter 7 concludes the thesis and shows the contributions and challenges of this research work.

Chapter 2

Related Work

2.1 Program Slicing

2.2 Slicing on Finite State Machine

This is a major difference between our proof and the existing correctness proof on Extended Finite State Machine (EFSM).

2.2.1 Control Dependence for Extended Finite State Machines

Chapter 3

Preliminaries

This chapter will present the preliminary works before the design and implementation of FormlSlicer. First of all, a good understanding of the modelling language upon which FormlSlicer is based is important; Section 3.1 introduces the semantics of Feature-oriented Requirements Modelling Language (FORML) extracted from a PhD thesis [8] and explains how we apply this language into the design. Section 3.2 illustrates a big picture of the slicing task; this inevitably brings in the concepts of variables controlled and monitored by Feature-oriented state machine (FOSM) in a FORML model because they play an important role in the slicing.

3.1 Adaptation of FORML semantics to the Use of FormlSlicer

FORML is a modelling language designed to specify feature-oriented models of Software Product Line (SPL) requirements. It is based on the Feature-oriented Software Development (FOSD) and accompanied by a taxonomy for explicitly modelling intended feature interactions [6]. A model expressed in this language consist of two major portions: a World Model, which is a description of the world in terms of a set of concepts and the relationships between them, and a Behaviour Model, which is structured in terms of many feature modules. Each feature module specifies the behaviour 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 [8].

There are many reasons why we choose [6]. The most important ones are:

- FORML is a UML-like language: the World Model and Feature Modules are based on UML class diagrams and UML state machines, respectively. This makes the design of the related slicing tool much easier, because we can utilize the research works that have been carried out by others in studying slicing on class diagrams and state-based machines.

- It is a good practice in software engineering research community to adopt and extend state-of-the-art analysis tools [9]. FORML is initially designed as a precise modelling language within NECSIS *Theme 3*[1] and there have been a few tools developed within the research group to manipulate the language, including FORML Feature Composer and a collection of transformation tools from FORML to SMV [9]—a modelling language used in model checking by the NuSMV model checker [11]. It is beneficial to the community in further extending this line of works by creating a model slicer to support the compositional model verification.

Here we focus on the feature module in the Behaviour Model of FORML. If that feature is an independent feature, such as the Basic Driving Service (BDS) feature which controls basic acceleration, deceleration and steering functionalities of a car, the feature module is represented as a complete state machine. If it is an enhancement or modification of another feature, the feature module is represented as a fragment; for example, the Cruise Control (CC) feature is built on top of BDS and thus its corresponding feature module is just a region that is to be embedded within one specific state of BDS. However, we can utilize the FORML Feature Composer to compose the fragmented feature module and its base feature modules so as to create one complete state machine.

In FormlSlicer, only complete state machines are taken as input. If the original feature module is a fragment, it needs to be composed with all its base feature modules to become one valid input model to FormlSlicer. Otherwise, an independent feature module like the BDS can be treated as a valid input model by itself. We call such a complete state machine as a Feature-oriented state machine (FOSM) to indicate that it is a composed feature in the format of a state-based machine.

[1]The Network for the Engineering of Complex Software-Intensive Systems for Automotive Systems (NECSIS) is a research network to tackle the obstacles and develop new Model Drive Engineering capabilities that lead to the development of the next generation of MDE methods and tools. This project, Feature Oriented Modelling and Analysis, groups activities within the NECSIS *Theme 3*: Uncertainty, Adaptability, and Variability [10].

A typical example of the FOSM is in Figure 3.1. The feature itself is a composite state with only one Region-the *main* region. Inside this region, there are two states, *S1* and *S2*. *S1* is a simple state. *S2* is a composite state which consist of two orthogonal regions, *C1* and *C2*. A state is drawn using solid rectangle whilst a region is drawn using dotted rectangle. The state and region form a containment relation in hierarchy (each composite state contains one or more regions; each region contains a substate machine). So you can see that such a mutual relationship between state and region is recursive and it can go infinitely deep. The black solid circle, called *pseudo state*, is the default beginning of the sub state machine inside its corresponding region. In FormlSlicer, we regard the first state pointed by the transition from pseudo state as the default start state, like *S1*, *S3* and *S5* in this figure.

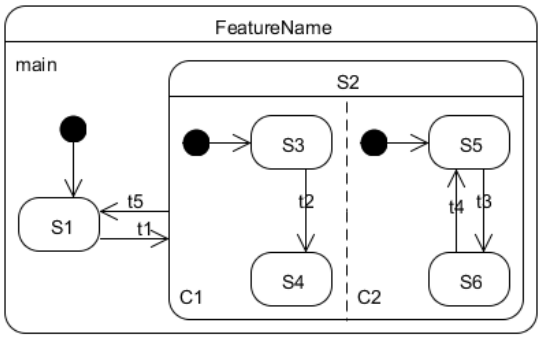


Figure 3.1: The Structure of FOSM in FormlSlicer

Throughout the thesis, we are going to use the following definitions adapted from [8] to access information about state hierarchy:

Definition. *The root of the state hierarchy represents the state machine, which is a composite state itself. The ancestors of a state x are all of the nodes along the path from the root node to x. The descendants of a node x are all of the states in the subtrees of x. The rank of a node 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 node that has both $x1$ and $x2$ as descendants.*

The transition in an FOSM 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, or called World Change Event (WCE), *gc* is an optional guard condition, and $al_1...al_n$ are labels that specify a set of concurrent actions, or called World Change Action (WCA), [8]. The Subsection 3.2 will explain how FormlSlicer extract information from the transition that are useful for computing the Data Dependency (DD).

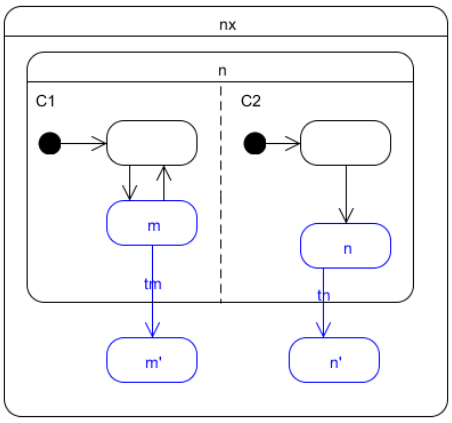


Figure 3.2: Concurrent Transitions crossing Hierarchy Border are not allowed in FormlSlicer

In FORML, transitions crossing two parallel orthogonal regions (*i.e.* different regions that are contained by the same composite state) are not allowed. 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 accessing a shared memory, it is illogical that a running thread suddenly reaches the middle of execution of another thread. Similarly, executions in different orthogonal regions can affect one another by generating/receiving the same event (*i.e.* one transition generates the event in WCA and another transition listens to that event in WCE), but not transiting from one state to another state crossing two parallel regions.

Moreover, if a state is within an orthogonal region which has another parallel orthogonal region (*i.e.* it is part of the concurrency), we do not allow any transitions crossing hierarchy border from or to this state. This kind of transition, as illustrated in Figure 3.2, is not mentioned in [8]. It may be allowed in some very rare real-life cases. However, in order not to complicate FormlSlicer, we decide not to consider this kind of transition.

3.2 The Big Picture of Slicing Environment

Figure 3.3 shows the big picture of environment during one iteration of slicing process. Inside the Behaviour Model, there are a set of FOSM with one of them being Feature of Interest (FOI). FOI is the feature which we are concerned about; all other features need to be sliced to retain the minimal components that are relevant to FOI; these sliced

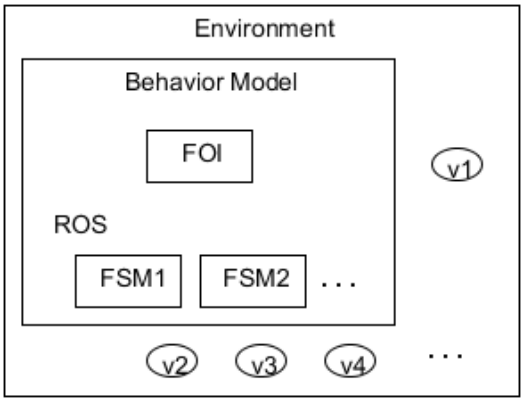


Figure 3.3: The Big Picture of Slicing Environment in FormlSlicer

components form the minimal context for FOI. We say that all these other features form the Rest of System (ROS). Take note that the slicing environment looks similar in other iterations except that a different FOSM is chosen as FOI.

Outside all the features, there are a collection of variables that are influencing the behaviours of one or more features. Some of them are environment-controlled variables, *i.e.*, their values can only be changed by the external environment. We use V_{env} to refer this set of variables. 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 direct set the actual speed; instead, they can only monitor it. Another group of variables are system-controlled because their values can be influenced by one or more FOSM. We use V_{sys} to represent them. For example, the acceleration of a car is a system-controlled variable.

If a variable is influencing the behaviour of the FOSM, we say it is the monitored variable of that FOSM. If a variable is influenced by the behaviour of the FOSM, we say it is the controlled variable of that FOSM. Based on the definitions, we know that any $v \in V_{env}$ can only be monitored variables.

Same definitions applies to one specific transition as well. From the WCE, we can extract the monitored variables of that transition, as shown in Table 3.1. In FORML, some message objects can be generated by WCA. In order to match a transition that consist of a WCE of *C+(o)* and another transition that consist of a WCA of *+C(o)*, FormlSlicer converts this pair of event/action to the same form as *+C* (See DD in Section **??**).

WCE Type	WCE Format	Monitored Variable
Message Object C Appear	$C+(o)$	$+C$
Message Object C Disappear	$C-(o)$	$-C$
Attribute in Message Object C Change Value	$C.a\ (o)$	$C.a$

Table 3.1: How FormlSlicer extracts Monitored Variables from WCE

Table 3.2 shows how FormlSlicer deals with different types of WCA. 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 all the objects belonging to that category of message are destroyed.

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 3.2: How FormlSlicer extracts Monitored/Controlled Variables from WCA

In an assignment action, FormlSlicer can extract the monitored variables from the arguments of the function and the controlled variables from the variable being assigned by that function. It is possible that there is only one literal value on the right hand side of this assignment; in this case, there is no monitored variables attached.

The Guard expression can be one of the three different types, as shown in Table 3.3. In particular, the "inState" expression is an interesting structure. It means that this guard condition is only true when the execution has currently come to a temporary stop at the state of "DummyState". Section **??** elaborates on the difference of "inState" expression compared the other two types and how to establish the DD relationship between the transition carrying an *inState(DummyState)* guard condition and the state of "DummyState" in the model.

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

Table 3.3: How FormlSlicer extracts Monitored Variables from Guard

Consider all the monitored variables in a transition as a set $mv(t)$ and the world change event in that transition as a singleton $wce(t)$, one might observe that $wce(t) \subseteq mv(t)$ because the world change event is transformed into one of the many monitored variables, as shown in Table 3.1. Similarly, consider all the controlled variables in a transition as a set $cv(t)$ and the world change action in that transition as a singleton $wca(t)$, one might observe that $wca(t) \subseteq cv(t)$ because the world change action is transformed into one of the many controlled variables, as shown in Table 3.2[2].

The "o" notation in a Transition One might have observed that there is sometimes an "o" written inside the bracket of a WCE or function parameters. In FORML, it refers to the current message object. FormlSlicer will simply replace this "o" with its corresponding WCE. For example, if a transition is expressed as *t1: SetHeadway+(o) /a1: HC.headway := o.dist*, then *SetHeadway.dist* is considered as the monitored variable for the World Change Action *a1*; this means that the argument *dist* attached to the message object *SetHeadway* is watched by the transition *t1*.

The **key principle of slicing in FormlSlicer is that the monitored variables of FOI will be used as a slicing criteria to select the initial collection of transitions in other FOSMs in ROS**. But the controlled variables are not, although they might appear in the slicing criteria as well if they happen to be monitored variables too. FormlSlicer will only take note of those FOSMs in ROS that have one or more such FOI-monitored variables as their controlled variables; it will ignore the monitored variables of the other FOSMs in ROS. **This is because we only care how the other FOSMs influence FOI, instead of how FOI influence the other FOSMs.** A rule of thumb to remember the principle is "FOI monitors, other FOSMs controls".

[2]See Chapter 5 for more usage on $mv(t), cv(t), wce(t), wca(t)$

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.

The original FORML model, which consists of multiple FOSMs, is input to the preprocessing task. The task converts this FORML model into a group of Control Flow Graph (CFG)s, organized in terms of feature. Based on these CFGs, the task computes three types of dependencies and stores the generated results in their respective tables.

Next, FormlSlicer forks off multiple slicing processes to perform the slicing task. All the slicing processes are sharing the same resources on dependencies 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 the sliced FORML model.

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.

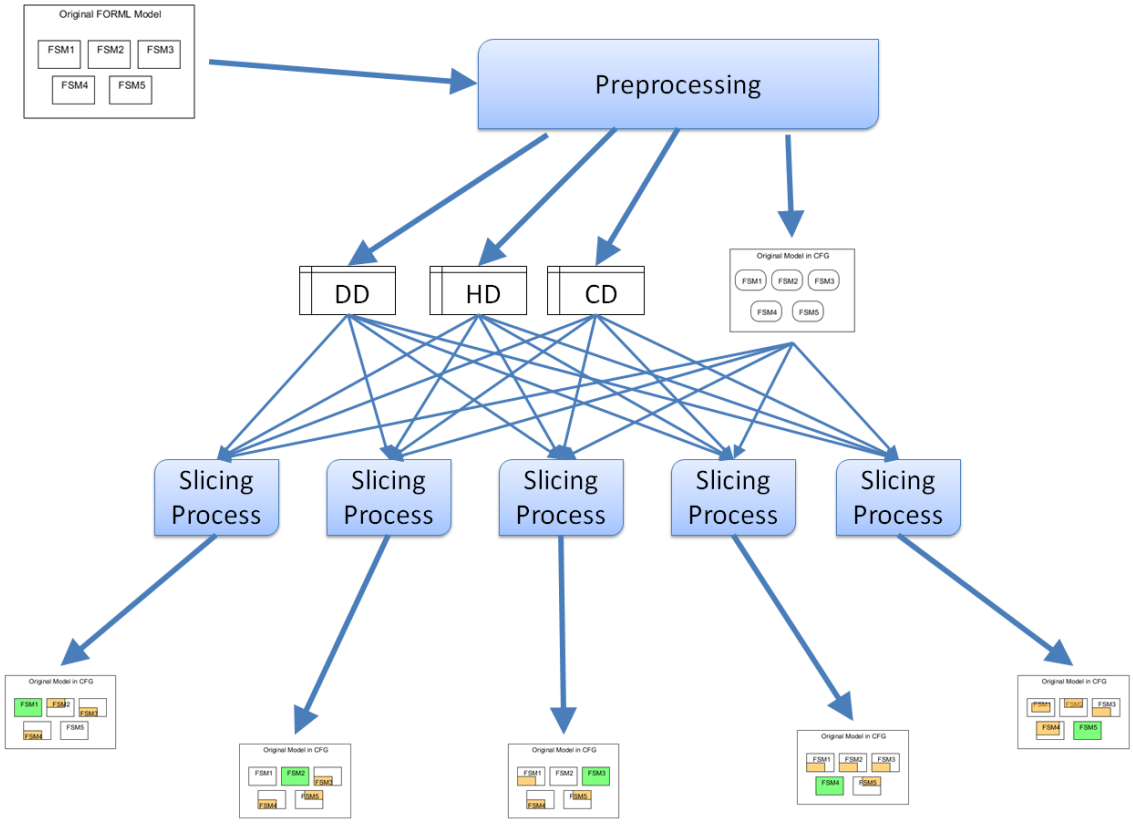


Figure 4.1: Overview of FormlSlicer's Workflow: the Preprocessing Task and the Slicing Task

The next three sections will dig deeper into the internals of the processing task and the slicing task.

Section 4.2 and Section 4.3 presents the processing task. It starts with a parser and a converter from FORML to CFG, then presents three types of dependencies, including Hierarchy Dependency (HD), Data Dependency (DD) and Hierarchy Dependency (HD).

Section 4.4 presents the workflow for one slicing process. The slicing is performed only on the FOSMs in ROS. It presents the multiple stages of the slicing process, including the Initiation Stage which utilizes a Variable Extractor, the General Iterative Slicing Stage that utilizes the dependencies, and the Model Enrichment Stage that de-fragment all resulting FOSMs to make them well-formed state-based machines.

Below we presents a list of sub-tasks performed throughout FormlSlicer's workflow:

1. Preprocessing Task
    (a) Parse and Convert FORML model to CFGs
    (b) Compute Hierarchy Dependency
    (c) Compute Data Dependency
    (d) Compute Control Dependency
2. Slicing Task[1]
    (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

[1]The multiple stages listed under slicing task are performed by one slicing process only. There are concurrently multiple slicing processes working on different FOI.

4.2 Preprocessing: Model Parsing and Conversion from FORML to CFG

FormlSlicer is equipped with a simple parser which can read an FOSM from an input file. 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
```

Table 4.1 explains the format of specifying all the components, including state, region, transition and macro, inside one FOSM. Note that the feature itself is a composite state that contains all the other components.

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

Control Flow Graph

Control Flow Graph is a directed graph, usually seen in compiler analysis to represent all execution paths through a program during its execution [12]. This concept is also useful

in model slicing. As introduced in Chapter 2, there are some implementations of model slicer (*e.g.* [13]) that convert a model into a CFG before slicing. FormlSlicer will do the same conversion.

By converting a FORML model into many CFGs, the FOSMs'[2] rich information will be stripped away so that they become simpler for further processing, particularly for the dependencies computations.

The CFG converted from FORML will consist of nodes and edges. As shown in Figure 4.2, there are two types of nodes: TNode and SNode. Each Node has an ID, a set of outgoing nodes, and a set of incoming nodes.

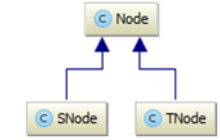


Figure 4.2: The Class Hierarchy of Node, TNode and SNode in CFG

TNode

Since transitions in FORML carry the richest information, including WCE, Guard and WCA, it is important to retain the structure of a transition after conversion from FORML to CFG.

FormlSlicer will convert each transition in FORML into a TNode. TNode contains the information of monitored variables and controlled variables of that corresponding transition.

The set of outgoing nodes and the set of incoming nodes of a transition are both singleton. They are IDs of the source state and destination state of that transition respectively.

SNode

It is important to retain the structure of a state after conversion from FORML to CFG; otherwise, it will be difficult to transform back from the CFG to another well-formed FORML after slicing.

FormlSlicer will convert each state in FORML 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 the generic type Node.

After Conversion After the FORML-to-CFG conversion, each FOSM will become one or more CFGs. The FORML example as shown in Figure 3.1 becomes the CFGs in Figure 4.3. If there are no cross-hierarchy-border transitions, the states and transitions

[2]The feature modules are still called "FOSM" even after their representations in implementation are changed from FORML to CFGs.

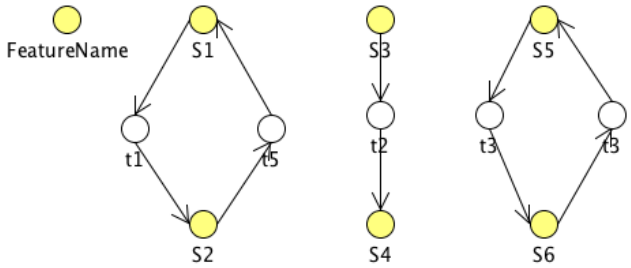


Figure 4.3: Several CFGs that are Converted from the FORML FOSM. Yellow Nodes are SNodes.

within each region of each composite state form one distinctly connected CFG. If there are cross-hierarchy-border transitions in the FOSM, that transition will be converted into a TNode which connects two distinct CFGs together.

When the CFGs for each FOSM are ready, it is time to move on to the next preprocessing step—dependencies calculations.

4.3 Preprocessing: Dependencies Computation

FormlSlicer starts from an empty slice set and gradually adds model components into it. In this slicing strategy, dependencies 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 FORML model have been converted into CFGs. CFG is a lightweight structure suitable for the dependencies computations, because dependencies computations concerns only the connectivity relationship among nodes.

This section introduces three types of dependencies that are computed by FormlSlicer on the CFGs. The computation results of dependencies serve as "dictionaries" for the slicing processes to look up.

4.3.1 Hierarchy Dependency

The states in a FORML model are organized in a hierarchy; some states are composite states and they contain one or multiple regions, in which more states are present. The states within the containing regions are called the **child states**; the state containing the regions is called the **parent state**. 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 start child state**[3] of its parent state.

Hierarchy Dependency (HD) is a dependency 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 calculates the HD while performing the FORML-to-CFG-conversion. It keeps two tables to record the HD results:

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

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

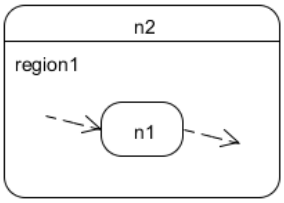


Figure 4.4: Hierarchy Dependency

4.3.2 Data Dependency

Data Dependency (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 is monitoring that variable and TNode B is controlling that 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**[5] 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 $[t_1 \cdots t_k](k \geq$

[3] FormlSlicer does not consider the scenario when a transition from a pseudo state to a default start child state carries non-trivial information, such as a monitored variable. If this is the case, users are supposed to treat the original pseudo state as a default start 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 start child states.

[5] The same definition is re-used in correctness proof in Section 5.2.4.

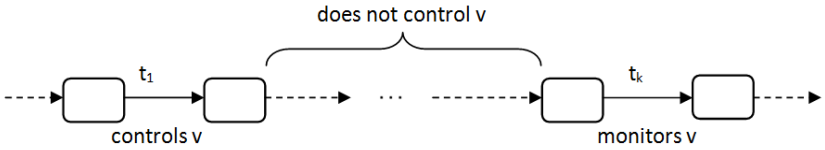


Figure 4.5: Data Dependency between t_k and t_1 w.r.t. v

1) such that $v \notin cv(t_j)$ for all $1 < j < k$. Here, t represents a TNode in the CFGs generated by FormlSlicer, v represents a system-controlled variable, $mv(t)$ represents the set of monitored variables contained in t and $cv(t)$ represents the set of controlled variables contained in t.

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.

DD is the most important dependency among the three. The concepts of monitored and controlled variables are specially created for it. The slicing principle of "FOI monitors, other FOSMs controls" mentioned at the end of Chapter 3 has adopted the notion of Data Dependency as well (*i.e.* in general, FOI is data dependent on other FOSMs).

InState Dependency FORML has 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 the purpose of consistency, FormlSlicer treats InState Dependency as a sub-type of Data Dependency. 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 dependency entry in the computation results of DD.

Algorithm 4.1 shows how FormlSlicer computes data dependency. 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 which controls a variable to another reachable TNode which monitors the same variable, and thereby establishes data dependency between

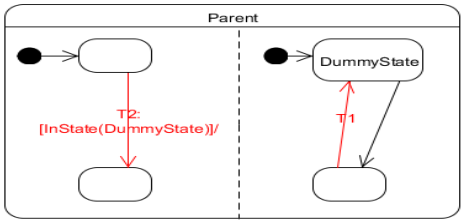


Figure 4.6: InState Dependency as a Variation of Data Dependency

Algorithm 4.1: Algorithm of Data Dependency 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 controlV in controlleds do
 5         set visitedSNodes := {}  // Empty visitedSNodes and qt
 6         set qt := empty queue
 7         set dependentOn := an empty pair consisting of two Nodes
 8         ADD the outgoing node of node1 to qt
 9         while qt is not empty do
10             set currNode := qt.poll();
11             if currNode == node1 then continue;
12             if currNode instanceof SNode then
13                 if visitedSNodes contains currNode then continue;
14                 if HDtable2 contains currNode then
15                     ADD the all the default start child SNodes of currNode to qt
16                 end
17                 ADD all the outgoing nodes of currNode to qt
18                 ADD currNode to visitedSNodes
19             else
20                 if dd contains controlV→(currNode,node1) then continue;
21                 if currNode.monitoredVars contains controlV then
22                     ADD (currNode,node1) to dependentOn;
23                 end
24                 set isContVRset := false
25                 foreach c in currNode.monitoredVars do
26                     if c == controlV then
27                         set isContVReset := true;
28                         break;
29                     end
30                 end
31                 if isContVRset := false then  ADD the outgoing node of currNode to qt
32             end
33         end
34         if DDtable contains Key controlV then  MERGE dependentOn to DDtable[controlV]
35         else  DDtable[controlV] := dependentOn;
36     end
37 end
```

the two. 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 9, it polls one node from *qt* and checks whether that node is monitoring the variable. If so, data dependency is established between the starting node and the polled node. There are two *ADD* statements, at Line 17 and Line 31, that add the outgoing nodes from *currNode* to *qt*, so as to allow the algorithm continue the breadth-first search to trace down the path beyond *currNode*.

The *foreach* statement at Line 25 is checking whether *currNode* is controlling the variable *controlV*. 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*.

4.3.3 Control Dependency

Non-termination Sensitive Control Dependency

Control Dependency (CD) captures the notion on whether one node can decide the execution of another node. Generally, a branching node[6] 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 dependency, 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 dependency, 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 start state is a branching state, using control dependency to add the default start state into the slice set is crucial. In FORML, the pseudo state points to one default start state; it is incorrect to have multiple default start states. If a default start state branches and it is not selected into the slice, we will run into trouble in determining which one can be a new default start state in the sliced model. This issue

[6] 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.

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 dependency is an important dependency in FormlSlicer's slicing task.

CD becomes complicated when it is computed in a CFG in which nodes can be connected in cycles. As elaborated in Chapter 2, researchers have proposed many different types of control dependency. FormlSlicer adopts the concept of **Non-termination Sensitive Control Dependency** as defined in [14]. A precise definition of control dependency 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;*
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 though n_j by taking one of the branch, and can possibly avoid n_j by taking another branch.

As 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. This key idea tackles the problem of cycles within nodes in CFG.

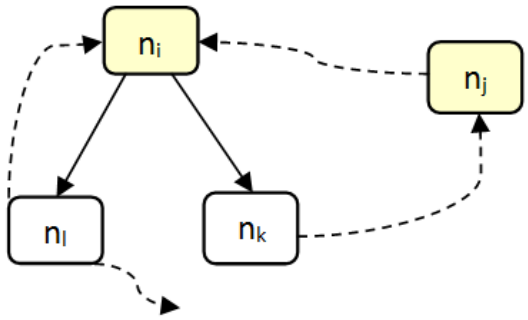


Figure 4.7: An Illustration of Non-termination Sensitive Control Dependency: n_j is control dependent on n_i

CD Algorithm: Representation of Paths for a Node

We have learnt from the algorithm of computing non-termination sensitive control dependency from [14] that main idea of computing control dependency 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. Based on that, we designed a dynamic algorithm suitable for our FOSMs.

We use an array p, which is a *String* 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.

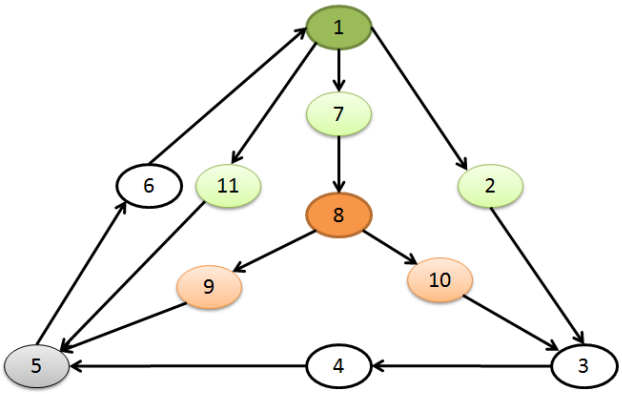


Figure 4.8: A CFG Example to Illustrate the Representation of Paths for Node 5 from Branching Node 1; Node 8 having Two Outgoing Nodes (9 and 10), Highlighted in Orange Colour; Node 1 having Three Outgoing Nodes (2, 7 and 10), Highlighted in Green Colour

The representation of paths for 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 can easily observe that there are four different possible ways to reach Node 5 from Node 1:

1. Passing through Nodes 1→11→5;
2. Passing through Nodes 1→7→8→9→5;
3. Passing through Nodes 1→7→8→10→3→4→5;
4. Passing through Nodes 1→2→3→4→5;

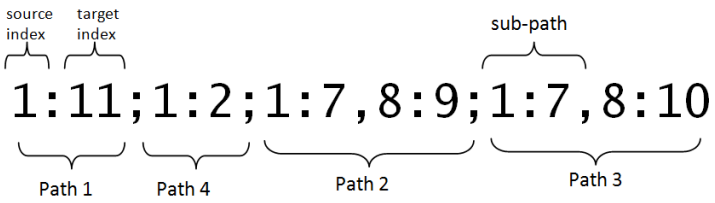


Figure 4.9: The Representation of Paths of Node 5 from Branching Node 1 in the CFG in Figure 4.8

There are only four paths, because reaching again Node 1 is analogous to reaching the end of path, as discussed in Subsection 4.3.3.

The representation of paths of Node 5 from the branching Node 1 is written in the format shown in Figure 4.9. The paths are separated by semicolon. Each path consists of one or more sub-paths. If a path consists of more than one sub-paths, it implies that by traversing the specific path we can encounter another branching node. Each sub-path consists of one source index and one target index.

Path 1 is represented as "1:11". It implies that 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.

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 has chosen the branch of Node 7; the second branching node is Node 8 and the path has chosen 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 has chosen the branch of Node 10 from Node 8. Some nodes along this path are neither branching nodes nor the next nodes of any branching nodes, such as Node 3 and Node 4. At either of these two nodes, the path has only one possible way to go. These nodes are not recorded in the path, because they do not carry useful information in distinguishing the current path from other paths.

Path 4 is represented as "1:2". At Node 1, the path has chosen to go through the branch of Node 2. Node 3 and Node 4 are ignored in the representation for the same reason in Path 3.

After collecting all the paths from Node 1 to Node 5. We can observe that no matter which path Node 1 takes, it always ends up at 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.

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. Based on the fact that Node 8 has only two outgoing neighbours (Node 9 and Node 10), the two paths "1:7,8:9" and "1:7,8:10" can be reduced to become "1:7". This implies that at the branching Node 8, no matter we take the branch of "8:9" or the branch of "8:10", we always end up at Node 5.

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

As the result, the representation of paths from Node 1 to Node 5 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 for 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.

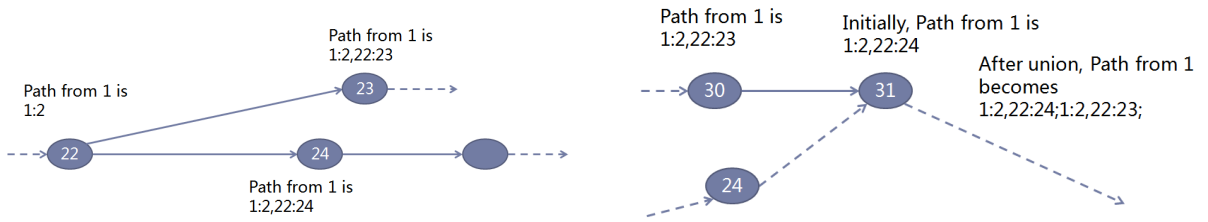
The representation of paths from Node 1 to some other nodes are not empty because we cannot reduce their paths to empty. For example, Node 3 will have a representation of paths from Node 1 to be "1:2;1:7,8:10". Node 3 is said to be control dependent on Node 1, because its representation of paths implies 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.

CD Algorithm: Pseudo-code and Explanations

Here we presents FormlSlicer's algorithm in computing the control dependency. 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, each cell storing the representation of paths for each node in CFG from a specific branching node. It is reset to *null* whenever a new branching node is considered in the computation.



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

(b) Union the Path Representation when *currNode* is not a Branching Node

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

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 neighbours of the branching node *node1*, by simply setting the first subpath *"n1idx:n2idx"* to indicate that the path from *node1* to its neighbour node *node2* has encountered branching once. These neighbour nodes are added into *workset*.

In each iteration of the *while* loop at Line 11, *currNode* is polled from the *workset* and analysed in three cases:

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

    - An example is shown in Figure 4.10a. The paths representation at Node 22 is "1:2"; the paths representations at its neighbours have been appended by a subpath to reflect what branch of path has been taken from Node 22, i.e. "22:23" and "22:24" respectively.
    - The function *Extend*, which performs the extending of paths representation, has been listed in Algorithm C.5.

2. If *currNode* is not a branching node, as detected at Line 21, then the paths representation of *currNode* needs to be merged with that of the only neighbour node of *currNode*.

    - An example is shown in Figure 4.10b. The paths representation at Node 31 was initially "1:2,22:24". After merging with the paths representation from the

Node 30, it now becomes "1:2,22:24;1:2,22:23"; the path "1:2,22:23" has been added into the representation of Node 31.

- The function *UnionPathHappens*, which performs the merging of paths representations, has been listed in Algorithm C.4.

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

As mentioned in the previous section, we cannot reduce the representation of paths to empty. These nodes are said to be control dependent on the branching node, because its representation of paths implies that, from the branching node, it is possible to take a path to pass through that node and take another path to avoid passing through that node. In this case, the branching node controls the execution of that node. Function *IsReachableFromPartialPaths* is constantly monitoring this scenario to determine the control dependency 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. In order not to break the flow of this chapter, the supporting functions are all listed in Appendix C.

Table 4.2 lists all the supporting functions' signatures and their goals.

4.3.4 Summary

In summary, we have computed all three dependencies 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

Algorithm 4.2: Main Algorithm of Control Dependency 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) OR
   |   |   |   IsReachableFromPartialPaths (n3index)) 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
```

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.

Function Signature	Goal of Function	Algorithm
HasReductionOccursBefore (targetIndex)	It determines whether the paths representation of the node identified by *targetIndex* has been reduced to empty.	C.1
IsReachableFromPartialPaths (targetIndex)	It determines whether the node identified by *targetIndex* is reachable from some but not all paths from the branching node, i.e., whether the node identified by *targetIndex* is control dependent on the branching node.	C.2
ReducePaths (originalPath)	It performs reduction on the representation of paths, which is *originalPath* in String format.	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 the paths in the node identified by *prevNodeIndex* to the paths in the node identified by *targetNodeIndex*. If the added paths are all duplicate to existing paths in the node identified by *targetNodeIndex*, the function returns false. If the paths identified by *targetNodeIndex* changes because of new paths added, the function returns true.	C.4
ExtendPath (srcIndex, newSubPath)	It adds the new sub path to each path in the paths representation in the node identified by *srcIndex* and returns the new paths representation.	C.5

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

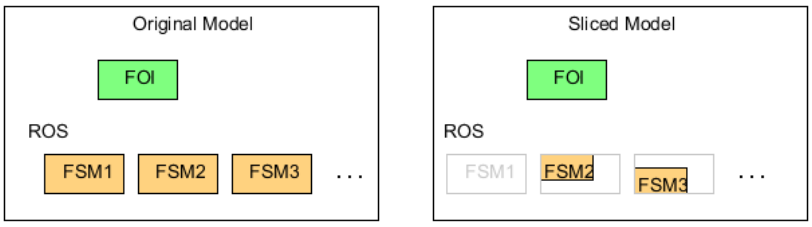


Figure 4.11: A Comparison between Original Model and Sliced Model

4.4 Multi-Stage Model Slicing

This section presents the workflow of model slicing for one slicing process forked off by FormlSlicer. We will name the workflow as **Multi-Stage Model Slicing**, because it is divided into several stages.

FormlSlicer's slicing strategy is to start with an empty slice set in ROS. At each step in the Multi-Stage Model Slicing process, certain new model elements (either an SNode or a TNode in the ROS) is selected into the slice set. In other words, the sliced model begins from emptiness and enlarges gradually after each step.

Throughout the slicing process, FOI will remain unchanged before and after slicing.

Figure 4.11 shows a side-to-side comparison between the original model and sliced model. Some FOSMs in ROS are completely sliced away; some are partially sliced away by FormlSlicer.

Table 4.3 lists the three stages in Multi-Stage Model Slicing process and explains their distinct purposes.

In the next subsections, we will present a slicing example. The example model consists of only two FOSMs; one of them is FOI and the other one is an FOSM in ROS. In this example, they are named as "FOI" and "ExampleFOSM" respectively.

Note that the Multi-Stage Model Slicing process is implemented to be performed on the converted CFG, instead of the FORML model itself. But it is easy to match CFG and its equivalent FORML model[8] In the following sections, we will present both the CFG and its equivalent FORML model side-by-side to show the effects of slicing at each step.

[8]If "state" or "transition" are mentioned, they refer to the SNode and TNode in CFG respectively.

Stage Name	Purpose of Stage
Initiation Stage	This stage aims to select the initial set of TNodes into the slice set.
General Iterative Slicing Stage	This stage utilizes the dependencies computed from the pre-processing task to add more SNodes and TNodes based on the initial slice set.
Model Enrichment Stage	This stage aims to ensure that after converting all the nodes in the slice set back to FORML model, they form well-formed FOSMs.

Table 4.3: Distinct Purpose of Each Stage in Multi-Stage Model Slicing Process

In addition, from Figure 4.13 to Figure 4.20, elements which are newly added into the slice set during the step of discussion are highlighted in red colour; elements that have been added into the slice set before the step of discussion are highlighted in black colour; elements that have not been added into the slice set are coloured in grey.

4.4.1 Initiation Stage

This stage aims to select the initial set of TNodes into the slice set in ROS, based on what FOI monitors.

4.4.1.1 Variable Extraction Step

In Section 3.2, we have discussed that the monitored variables of FOI will be used as a slicing criteria to select the initial collection of transitions in other FOSMs in ROS. This is because we are only concerned about how the other FOSMs in ROS influence FOI, instead of how the FOI influence the other FOSMs.

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

Figure 4.12 shows a simple example of FOI which is monitoring only one variable, $v1$. *Variable Extractor* takes in the FOI in its CFG form (shown in Figure 4.12a) and checks through all the TNodes' monitored variables. In the end, it outputs a set of variables, V_{Rv}, that are relevant to the FOI. in this example, the set contains only one variable, i.e., $V_{Rv} = [v1]$. We call this list of variables as **relevant variables**.

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

Figure 4.12: A Simple Example of Feature of Interest

4.4.1.2 Initial Transition Selection Step

Before the Initial Transition Selection step, we have an empty slice set in ROS. Now, FormlSlicer will select all the TNodes in ROS that control any relevant variable collected by *Variable Extractor* in previous step, and add them into the slice set.

A simple example of FOSM in ROS is shown in Figure 4.13. Figure 4.13b shows the original FORML model, whilst Figure 4.13a shows its CFG.

In our example, the transition *t11* in Figure 4.13b has the expression "t11:[v2==‘val2’] /a1:v1:=‘val’;". According to how FormlSlicer processes the transition expression as discussed in Section 3.2, the TNode in CFG which corresponds to *t7* has one monitored variable $v2$ and one controlled variable $v1$.

Because $v1$ is a relevant variable, *t11* becomes one of the initial transitions selected by FormlSlicer into the slice set. It has been highlighted in red colour in Figure 4.13. The monitored variables of *t11*, e.g., $v2$, need to be added to the set of relevant variables.

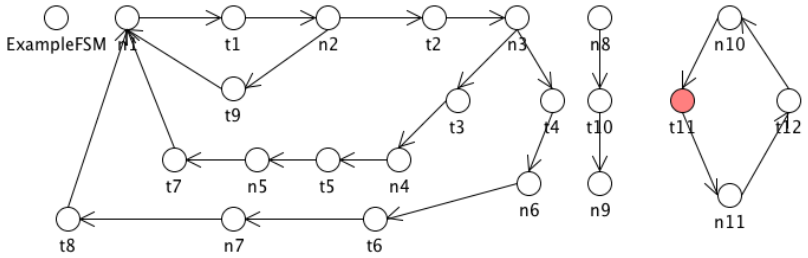
4.4.2 General Iterative Slicing Stage

As mentioned in Section 4.3, we have the results from dependencies computation, including *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 slice set.

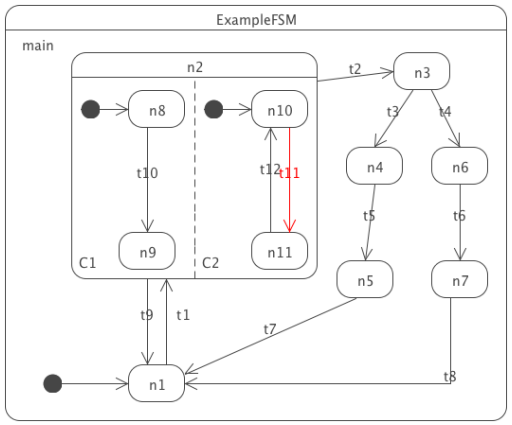
The General Iterative Slicing Stage consists of four steps:

DD Step Add more TNodes that are transitively data dependent on the TNodes in slice set with respect to one of the relevant variables

Transition-to-State Step Add more SNodes based on the TNodes in slice set



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



(b) The FORML model of the Example CFG in ROS

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

Replacing Cross-Hierarchy Transition Replace all cross-hierarchy transitions with "true" transitions

CD-HD Step Add more SNodes that are transitively control dependent or hierarchy dependent on the SNodes in slice set

The reason why we have the word "iterative" in the name of this stage is that both the DD Step and CD-HD Step perform the node adding activity **iteratively**, in order to add nodes that are **transitively** dependent on existing nodes in slice set.

Figure 4.14 outlines the basic workflow in these four steps.

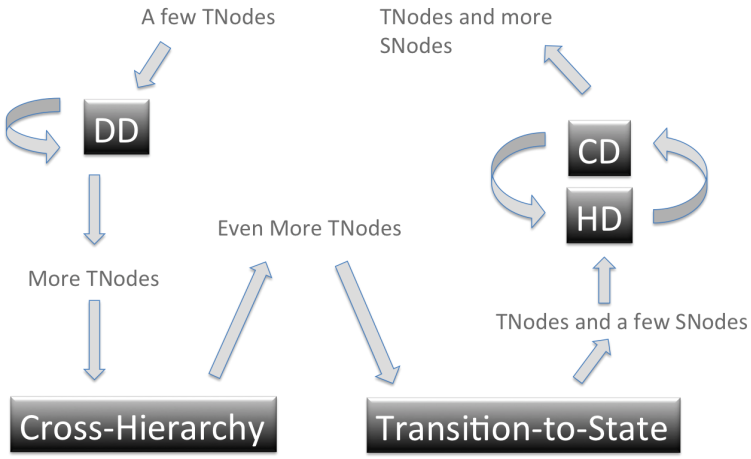


Figure 4.14: Four Steps in General Iterative Slicing Stage

4.4.2.1 DD Step

DD Step uses *DDtable* to add more TNodes that are transitively data dependent on the TNodes in slice set.

DD Step, or any steps similar to it, is an inevitable step in the workflow of almost all slicers. After all, slicing is a static analysis technique based on define-use relationships among elements. A program slicer needs to identify more lines of program codes iteratively based on define-use relationships of variables. Similarly, a model slicer needs to iteratively find more transitions based on define-use relationships of variables. This define-use relationship is the data dependency relationship among nodes.

Algorithm 4.3 shows a simplified version of what DD Step does. It starts with the set of relevant variables and the slice set, both of them obtained from the Initial Transition Selection Step. Then it iterates repeatedly, as shown at Line 2, and tries to enlarge the size of *sliceSet* and *relevantVariables* at each iteration until no more changes are possible.

At Line 5, the algorithm looks up *DDtable* and finds if any TNode in the slice set is data dependent on another TNode not in slice set with respect to a relevant variable. If that is the case, the TNode not in slice set, named as *anotherTNode*, will be added into the slice set, as shown at Line 6. Then the monitored variables of *anotherTNode* need to become relevant variables as well; they are added into *relevantVariables* at Line 7.

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 to the set of

relevant variables, it means that the algorithm has to continue searching more TNodes.

Algorithm 4.3: Simplified Algorithm of DD Step in General Iterative Slicing Stage

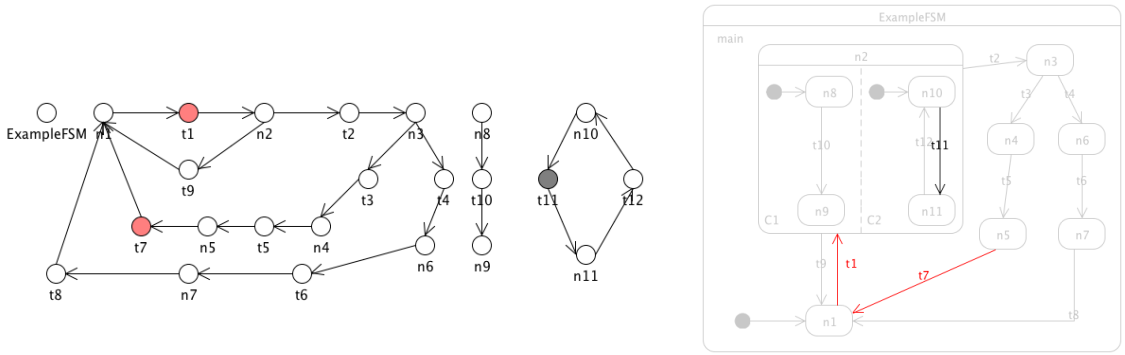
```
   Input: sliceSet, relevantVariables, DDtable
   Output: SliceSet, relevantVariables
 1 set listOfRelevantVariables := {};
 2 repeat
 3     set prevSizeRelVar := relevantVariables.size();
 4     foreach v in relevantVariables do
 5         if DDtable contains v⇒(tnode, anotherTNode) AND sliceSet 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 based on a growing slice set in order to include those TNodes on which the FOI is transitively data dependent. Re-consider the example in Figure 4.13. We know that $t11$ has been selected into the slice set during the Initial Transition Selection Step, because $v1$ is a relevant variable from FOI. We also know that the set of relevant variables now contain $v1$ and $v2$. Now, consider the transition *t1* which controls the variable $v2$ and monitors the variable $v3$. Then, *t11* is data dependent on *t1* with respect to $v2$ and we need to add *t11* into the slice set. However, $v3$ is monitored by *t1* and therefore its value is important to *t11*. If there is another transition (e.g., *t7*) which controls $v3$, that transition can influence *t1*, and transitively influence *t11*, which again transitively influences FOI. Therefore, we must include $v3$ into the set of relevant variables and perform the same procedure of node adding in another iteration.

This example is shown again in Figure 4.15. TNodes *t1* and *t7* are added into the slice set.

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



(a) The Control Flow Graph of the Example CFG in ROS **(b)** The FORML model of the Example CFG in ROS

Figure 4.15: The Example FOSM in ROS after DD Step

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. 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. 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.

The complexity comes from the composite states with their hierarchy boundaries involved in a cross-hierarchy transition t_{ch}. Consider this composite state $n_{composite}$; note that $n_{composite}$ is neither the source state nor the destination state of t_{ch}, but it is an ancestor state of either of them. If $n_{composite}$ 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 $n_{composite}$ 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. For example, if this composite state $n_{composite}$ is a source state of another important transition t_x, then any appropriate events can trigger t_x to leave $n_{composite}$. The correct triggering of t_x is therefore affected by a cross-hierarchy transition t_{ch} which either enters or exits $n_{composite}$. If we do not select t_{ch} into the slice,

$n_{composite}$ 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 $n_{composite}$ and create a path from a starting part-of-slice state to $n_{composite}$. The caveat of doing so is that it cannot accommodate many corner cases. For example, if after the cross-hierarchy transition t_{ch} reaches the descendant state of this composite state, there is another transition t_y which transits from this descendant state to another state within the hierarchy boundary, then t_y cannot be simulated in the sliced model because there is no way to transit from a state to its descendant state using a transition. To make the example more complex, consider the case if t_y is a transition after a long path from t_{ch} while that long path consists of many cross-hierarchy transitions. We will then need a way to fix all these 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 algorithm in this step looks for any cross-hierarchy transitions which have not been added into slice yet, 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 slice set.

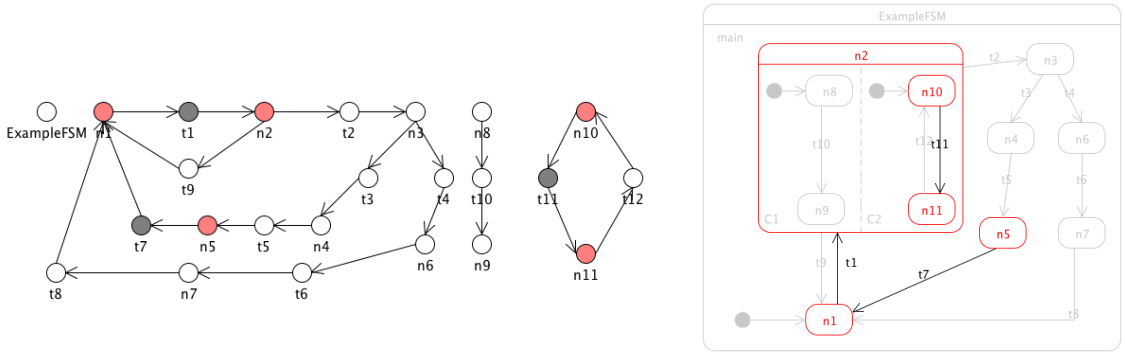
The rationale of replacing it with a "true" transition is that we are sure that this transition does not control any relevant variables. This is because the step of Replacing Cross-Hierarchy Transition comes after the DD Step, in which all transitions that can transitively control the relevant variables have already been identified and added into slice. Thus, any transitions that are neglected by the DD Step do not involve in any data dependency relationships that are relevant to the FOI. We can therefore safely ignore their monitored or controlled values.

The next two steps will cater the source and destination state of this newly added "true" transition, as well as any ancestor states of its source or destination state. Eventually, it will become part of a well-formed FOSM.

4.4.2.3 Transition-to-State Step

So far, all the model elements that have been added to the slice set are transitions (or TNodes in CFG). The Transition-to-State Step is a turning point to bring in the states (or SNodes in CFG).

For each of the transitions that have been added into the slice, this step adds its source and destination state into the slice as well. Figure 4.16 shows the effect of this step.



(a) The Control Flow Graph of the Example CFG in ROS **(b)** The FORML model of the Example CFG in ROS

Figure 4.16: The Example FOSM in ROS after Transition-to-State 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 states into the sliced model, which forms the starting point for the CD-HD Step.

4.4.2.4 CD-HD Step

Section 4.3.3 and Section 4.3.1 have explained the importance of control dependency and hierarchy dependency respectively. The results of computing these two dependencies, including *HDtable1* and *CDset*, are used in the CD-HD Step to add more SNodes that are transitively control or hierarchy dependent on the TNodes in the slice set.

Algorithm 4.4 shows a simplified algorithm of the CD-HD Step. At Line 4, it looks up the hierarchy dependency and adds an SNode into the slice if its child SNodes are part-of-slice. At Line 8, the algorithm looks up the control dependency and adds an SNode which controls the execution of other part-of-slice nodes.

The two lookups are repeated until no more changes occur, as shown at Line 12. The reason for such an repetition is that an SNode can be transitively control dependent or hierarchy dependent on another SNode. One example is that a composite state can contain many descendants in different ranks of state hierarchy, i.e., it contains many layers of

hierarchy boundaries. Once a descendant state is selected into the slice, the repetition in CD-HD Step will ensure that all the states from the root state to that descendant state along the state hierarchy are added into the slice, including the composite state mentioned.

Such a repetition of lookups needs to include both control dependency and hierarchy dependency together. For example, if $n1$ is a state in the slice and its parent state $n2$ is not, the algorithm will find $n2$ during the lookup in hierarchy dependency and add it into the slice; if $n2$ is control dependent on another state $n3$ which is not in the slice, the algorithm will find $n3$ during the lookup in control dependency.

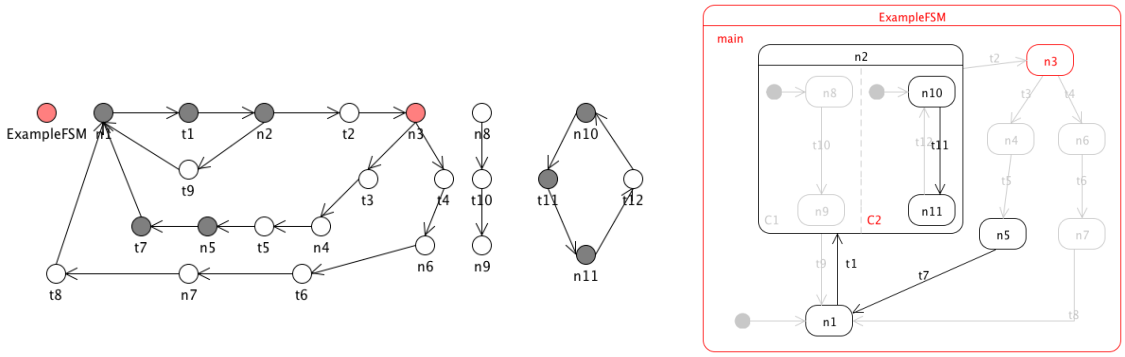
Algorithm 4.4: Simplified Algorithm of CD-HD Step in General Iterative Slicing Stage

```
   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
```

Figure 4.17 shows the same slicing example after the CD-HD Step. The state *n3* has been added into the slice 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. We can observe that the root state is always part-of-slice, as long as any state within the root state is part-of-slice.

4.4.3 Model Enrichment Stage

Model Enrichment Stage aims to make each fragmented CFG in the slice to become a connected graph, so that after converting all the CFG nodes in the slice back to a FORML model, all the FOSMs are well-formed. Informally, an FOSM is a well-formed FOSM when



(a) The Control Flow Graph of the Example CFG in ROS **(b)** The FORML model of the Example CFG in ROS

Figure 4.17: The Example FOSM in ROS after CD-HD Step

all the states are reachable from the pseudo state and all transitions can be triggered when appropriate events occur; in other words, it is in good shape.

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 and make the graph fragmented. There are two steps in this stage:

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 being a well-formed FOSM, but also increases the degree of reduction in slicing.

FormlSlicer will adopt the two state merging rules which have been defended in [15] to satisfy the *traversability* property[9]:

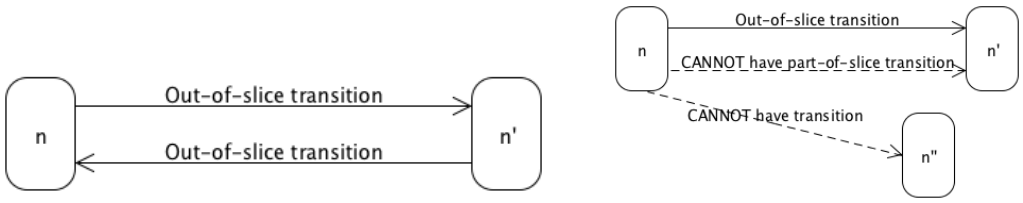
Rule 1 If there exist out-of-slice transitions from state n to n' and also from state n' to n, these two states are merged into one state "n,n'".

[9]See Section 2.2 on elaborations about traversability property on a slice.

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 4.18 illustrates these two rules.



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

Figure 4.18: Illustration of State Merging Rules in Step I of Model Enrichment Stage

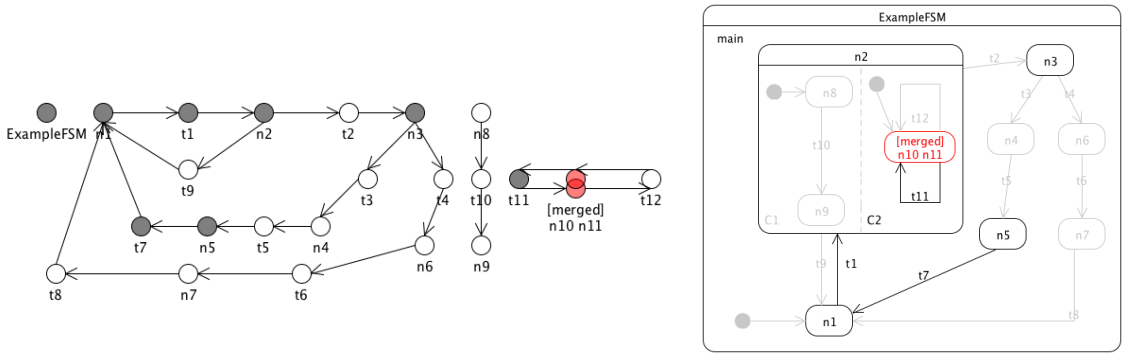
Intuitively, these two rules can be easily justified. In Rule 1, since the transitions between the two states are out-of-slice, any mutual traversal between the two states are considered as not important; otherwise, the transitions will be selected into the slice before reaching the State Merging Step. In Rule 2, n does not have anywhere else to move except n' and thus it is safe to merge n and n'.

FormlSlicer adds two more restrictions on these two rules.

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

2. It will perform state merging only when these two states have the same parent state. It will causes the sliced model to be incorrect if the source state and destination state of a cross-hierarchy transition are merged.

Figure 4.19 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*".



(a) The Control Flow Graph of the Example CFG in ROS **(b)** The FORML model of the Example CFG in ROS

Figure 4.19: The Example FOSM in ROS after Stage Merging Step

4.4.3.2 Step II: True Transitions Creation

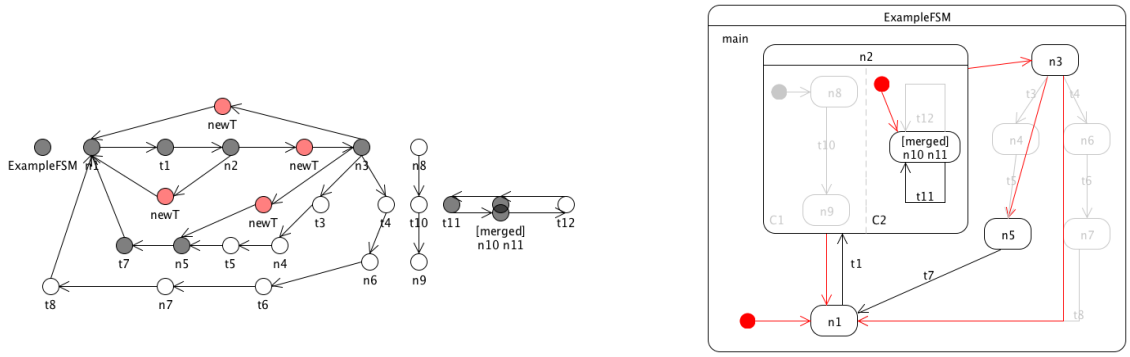
The True Transitions Creation step makes all part-of-slice states reachable from the pseudo state. It starts from the beginning of the FOSM and "probes" to connect the part-of-slice states together. This is the last step for the entire Multi-Stage Model Slicing process.

There are two sub-steps in Step II:

Sub-step 1: Search for New Default Start State It is possible that the original default start states of certain sub-state machines are not selected into the slice. The sub-step 1 searches for the new default start state for each sub-state machine.

This sub-step benefits greatly from the use of control dependency. As explained in Section 4.3.3, in the scenario when the default start state is a branching state, that default start state will be added into the slice because of control dependency. This prevents FormlSlicer to run into trouble of creating multiple default start states in the sliced model when the original branching default start state is out-of-slice. Therefore, for each sub-state machine in the resultant sliced FOSM, there will be only one default start state; it is either the original default start state, or the next-part-of-slice state of the original default start state.

Figure 4.20 shows the slicing example in which the pseudo states in regions *C2* and *main* pointing to their respective default start state. They happen to be the same as that in original FOSM. We cannot find a new default start state in the sub-state machine within region *C1*, because all the states within region *C1* are not added into the slice.



(a) The Control Flow Graph of the Example CFG in ROS **(b)** The FORML model of the Example CFG in ROS

Figure 4.20: The Example FOSM in ROS after True Transitions Creation Step

Sub-step 2: Search for Next Part-of-slice State Consider a path starting from a state which has been added into the slice in the previous steps. It has many outgoing transitions, but all the outgoing transitions are not selected into the slice, and thereby disconnecting this state from reaching other states. There may be another part-of-slice state that are far away, but reachable, from this state. We need to connect them together so that we form a path between the two states in the sliced FOSM. The path may be shorter (or having the same length, in the case when the two states are neighbouring states) in the sliced FOSM compared to that in the original FOSM; but it will make simulation of the original model by the sliced model possible. This "shorter path" will be a "true" transition[10].

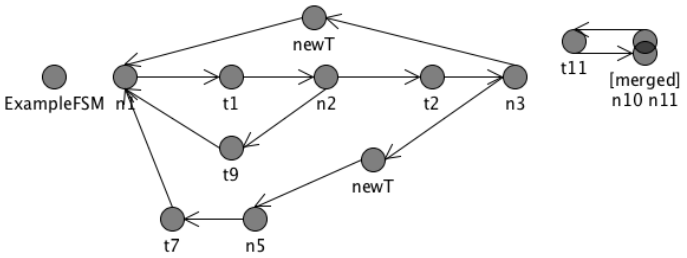
The challenge is to decide which next state is the appropriate state to connect with the starting state using a true transition. We call all these appropriate next states "**next part-of-slice states**". Informally, a part-of-slice state n' is the **next part-of-slice state** of state n if there is a path, consisting of many out-of-slice transitions, that starts from n and reaches n' and that all the states along this path are out-of-slice and are at the same rank of state hierarchy. A state may have more than one part-of-slice states because there may be branches along the path that leaves from that state. We call all of them as the **next part-of-slice state set**.

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.

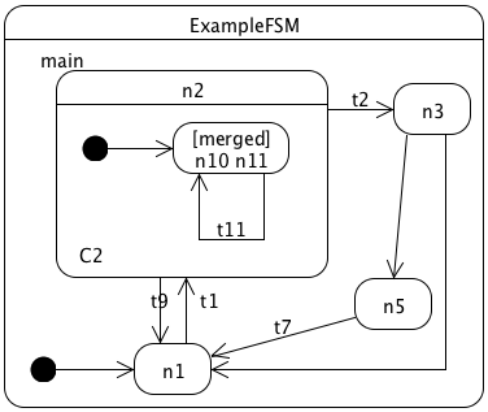
[10]Recall from Section 4.4.2.2 that a transition carrying only "true" in its guard condition is called a "true" transition.

Figure 4.20 shows the slicing example in which state *n3* has been connected to its next part-of-slice state *n1*. Note that the path between a part-of-slice state and its next part-of-slice state can be as short as one transition only. 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 too.

4.4.4 Summary



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



(b) The FORML model of the Example CFG in ROS

Figure 4.21: The Example FOSM in ROS after All Steps in Multi-Stage Model Slicing

In this chapter, we have presented all the steps for the Multi-Stage Model Slicing

process. At the end, FormlSlicer performs a **conversion from CFG to FORML** to convert the sliced CFGs into a sliced FORML model. It will then write the output of the sliced FORML model in the format as specified in Table 4.1.

The original FOSM example shown in Figure 4.13 has now become a smaller FOSM after all the steps, as shown in Figure 4.21.

In the next chapter, we will present a correctness proof on the Multi-Stage Model Slicing process to show that the resultant sliced model can simulate the original model.

Chapter 5

Correctness of FormlSlicer

This chapter presents a correctness proof to show that the sliced FORML model produced by the Multi-Stage Model Slicing process, as elaborated in Chapter 4, can simulate the original FORML model.

5.1 Overview

5.1.1 Purpose of the Proof

As the slices are used in replace with the original model for safety property checking, the non-negotiable requirement for the slicer 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 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.

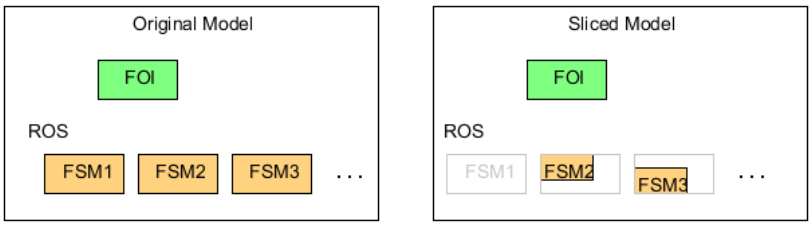


Figure 5.1: The Slicing Environment with Original Model and Sliced Model

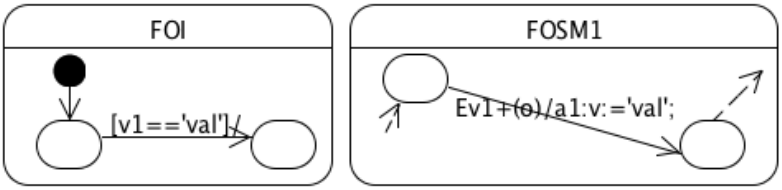


Figure 5.2: A Simple FOI Executing with Another FOSM in Rest of System

Figure 5.1 shows a side-to-side comparison between the original model and sliced model. The FOI remains unchanged in the sliced model. Some FOSMs in the ROS are completely sliced away; some are partially sliced away by FormlSlicer.

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 might be triggered only when via its guard condition 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.

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 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 mathematical induction to show that the simulation of the original model by the sliced model is maintained from the beginning of the execution trace in original model to the end.

Section 5.2 defines terminology that is useful for the proof. This section introduces some concepts, such as state configuration and interpretation, that are very important in formally defining what "projection" means between snapshots.

Section 5.3 describes the state transition rule that is standard to a hierarchical concurrent state machine. Due to the complexity of such a state machine, the rule is not strictly formalized.

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

Variables

A single variable is denoted as v; a set of variables is usually denoted as V with appropriate subscripts. The set of environment-controlled variables is denoted as V_{env}, whilst the set

of of system-controlled variables is denoted as V_{sys}[1].

This chapter intends to prove that any relevant variable[2] has same value in both original model and sliced model. Because the environment-controlled variables are influenced only by 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 "temperature" has a value of "larger than 37℃". Therefore, the following proof is going to consider only the 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. Among them, N^{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 • 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 regions and child 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 3.1 has shown a FORML example to illustrate this relationship.

We also need a notation to indicate that two orthogonal regions are parallel 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:

[1]Environment-controlled variables' values can only be changed by the external environment. System-controlled variables' values can be influenced by one or more FOSMs. Section 3.2 has defined these two sets of variables more clearly.

[2]The concept of "relevant variables" have been introduced informally in Section 4.4.1 and Section 4.4.2.1. During the Initiation Stage, all the monitored variables of all the transitions in FOI are added to the set of relevant variables. The set of relevant variables enlarges at each iteration of finding more nodes based on Data Dependency, during the DD Step in General Iterative Slicing Stage.

[3]See Chapter 3 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, $[r]$.

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

Transitions

A transition is a progression in an FOSM'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'.*

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 concurrent regions, we do not allow any transitions crossing hierarchy border from or to this state[5].

Model

A behaviour model in FORML consists of many feature modules [8]. In this proof, we simply use the word "model" to refer to the behaviour model which consist of all FOSMs in the model, including FOI. We normally use M to denote a model.

If M contains k FOSMs, $[F_1, \ldots, F_k]$, we write it as:

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

In FormlSlicer, a model is treated as one big state machine with k orthogonal regions, each containing the sub-state machine of an FOSM. This is a 150% SPL model to Jo: shall I insert citation from Sandy's newly published paper?.

[5] See Section 3.1 for more details on this restriction.

5.2.2 State Configuration and Interpretation

A Feature-oriented state machine (FOSM) F is a well-formed state machine in the behaviour model notation of FORML [8]. It consists of a finite set of states, S_F, including composite states and simple 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 simple 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 a set of states. Therefore, we use N to refer to the **state configuration** of the FOSM.

This is not saying that the machine could 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 $n \in N$, then so are all of n's ancestors, and if any composite state $n \in N$, then for each r in $ChildRegions(n)$, there must be one state contained in r that are in N too. (A state machine's active state configuration is the set of active states in a hierarchy tree where an active concurrent composite state contains one active substate per orthogonal region. [16])

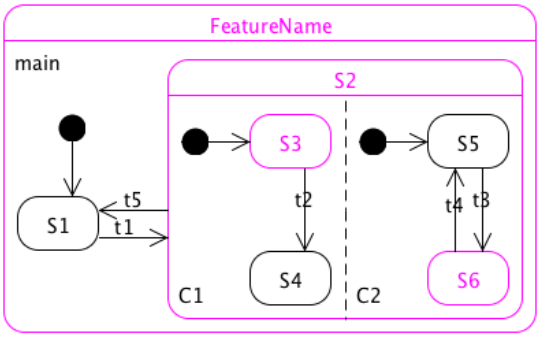


Figure 5.3: The FORML Example from Figure 3.1 with its Current States Highlighted in Magenta

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.

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 FOSM's execution [17]; it refers to the status of an FOSM between execution steps.

Definition 3. *We define the snapshot of an FOSM as a combination of the state configuration of the FOSM, 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 FOSM. Through an execution step, the FOSM effectively exits the set of current states of the FOSM and enters the set of destination states of the execution on the FOSM[6]; meanwhile, the values of some variables may be changed.

Here we define an execution step formally.

Definition 4. *We write $F \vdash e : (N, \sigma) \Rightarrow (N', \sigma')$ when we refer to an execution step, e, that occurs in an FOSM, F, 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 parallel regions and the various accessor functions in Section 5.2.1.

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

Definition 5. *The set of concurrent transitions that are triggered simultaneously in an execution step e in $F \vdash e : (N, \sigma) \Rightarrow (N', \sigma')$ 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 exist another state p such that $ss(t_i) \xRightarrow{hd} p$ and $ParentRegion(ss(t_i)) \parallel ParentRegion(p)$, for any $1 \leq i \leq k \wedge 1 \leq j \leq k \wedge i \neq j$;*

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 border on parallel regions, as discussed in Section 3.1 for more details.

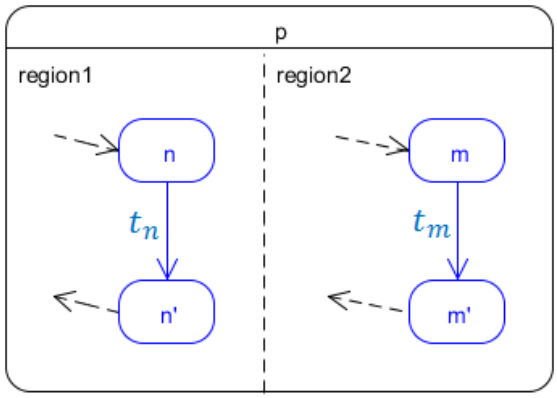


Figure 5.4: Concurrency in Orthogonal Regions as an Execution Step (Only Blue Coloured Components are Relevant in the Execution)

Slice Set We use $\mathcal{L}$ to represent a slice set. If a model element m is in the slice, we use $m \in \mathcal{L}$ to denote that it is part of the slice. This symbol $\mathcal{L}$ is also used as a subscript to annotate a similar meaning. Thus, an original FOSM F would become $F_{\mathcal{L}}$ after slicing. The subscript $\mathcal{L}$ is used in other notations as well to denote similar concepts, such as $N_{\mathcal{L}}$ or $\sigma_{\mathcal{L}}$ and so on.

Monitored, Controlled, World Change Events and Actions The set of monitored variables and controlled variables of an execution e are denoted as $mv(e)$ and $cv(e)$ respectively. The corresponding WCE and WCA of e are denoted as $wce(e)$ and $wca(e)$ respectively. As we have discussed in previous chapters, $wce(e) \subseteq mv(e)$ and $wca(e) \subseteq cv(e)$[7].

5.2.4 Dependencies

We denote T_M to represent all the transitions in all the FOSMs (including FOI) in the model M. Similarly, we denote S_M to represent all the states in all the FOSMs in the model M.

Definition 6. *We say $n_1 \in S_M$ is **hierarchy dependent** on $n_2 \in S_M$, denoted as $n_1 \xrightarrow{hd} n_2$, iff n_1 is a child state of n_2'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 $t \in T_M$ or $n \in S_M$ is **control dependent** on $n' \in S_M$, denoted as $t \xrightarrow{cd} n'$ or $n \xrightarrow{cd} n'$, iff t or n is directly non-termination sensitive control dependent[8] on n'.*

Transitivity on dependencies 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 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 same notation applies for control dependency.

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.

[7] See Table 3.1 and Table 3.2 on how FormlSlicer extracts the information about monitored and controlled variables from World Change Event and World Change Action respectively.

[8] See Section 4.3.3 on explanation of "directly non-termination sensitive control dependence".

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[9].

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 start states

Then, we define the least common ancestor between two states to the highest rank of state that is an ancestor state of both states of t [18, 8].

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 that have the following relationships with $ss(t)$:
  - 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 that have the following relationships with $ds(t)$:
  - 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 start states of $ds(t)$ and its entered descendant states, $InitDesc(ds(t))$;

[9] 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 [17]. By adding in concurrency, it becomes more complicated. Due to these challenges, here we will define a generic state transition rule in plain English for all scenarios.

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

$$N' = (N - ss(t) - AncestorTillLCA(ss(t), ds(t)) - Descendants(ss(t))) \cup ds(t) \cup AncestorTillLCA(ds(t), ss(t)) \cup InitDesc(ds(t))$$

5.4 FormlSlicer's Multi-Stage Model Slicing

Section 4.4 has elaborated on how the FormlSlicer performs a multi-stage slicing on an FOSM with respect to FOI. Now, this section expresses the multi-stage slicing process using the semantics defined in Section 5.2. At each step, certain new components (either a state or a transition) is selected into the slice set $\mathcal{L}$.

For brevity reason, we denote T_{ROS} to represent $\bigcup_{\forall f \in ROS. \forall t \in T_f} t$. Similarly, we denote S_{ROS} to represent $\bigcup_{\forall f \in ROS. \forall n \in S_f} n$.

The concept of the next part-of-slice state set has been informally introduced in Section 4.4.3.2. Here, we need a formal definition on this concept because it will be useful in explaining the last step of Model Enrichment Stage, in which the fragmented result sliced model is connected via newly created true transitions to form a well-formed FOSM.

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 $n' \in \widetilde{npos}(n)$:*

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

Note that k can be 1 in Definition 9. In this case, in between the state n and its next-part-of-slice state n' there exists only one out-of-slice transition.

1. Initiation Stage

   (a) Variable Extraction Step
   Let
   $$V_{FOI} = \bigcup_{\forall t \in T_{FOI}} mv(t)$$
   where T_{FOI} refers to the set of transitions in the Feature of Interest.

(b) Initial Transition Selection Step
$\forall v \in V_{FOI}.(\exists t \in T_{ROS}.v \in cv(t) \Rightarrow t \in \mathcal{L})$.

2. General Iterative Slicing Stage

(a) DD Step
$\forall t, t' \in T_{ROS}.(t \xRightarrow{dd} t'.t \in \mathcal{L} \Rightarrow t' \in \mathcal{L})$.

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

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

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

3. Model Enrichment Stage

(a) Step I: State Merging Step

i. Rule 1
Consider two states $n, n' \in N_{ROS}.n \in \mathcal{L}$. 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 $T_{n,n'} \neq \emptyset \wedge T_{n',n} \neq \emptyset \wedge (\forall t \in T_{n,n'} \cup T_{n',n} \Rightarrow t \notin \mathcal{L})$, then n and n' could be merged together.

ii. Rule 2
Consider two states $n, n' \in N_{ROS}.n \in \mathcal{L}$. 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 $(\forall t \in T_{n,n'}, t \notin \mathcal{L}.) \wedge (T_{n,n'} = T_n)$, then n and n' could be merged together.

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

5.5 Proof

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.

Before moving on to the proof itself, we need to formally define what "projection of snapshot in the original model to snapshot in the sliced model" means. In the FORML model, it includes two notions:

1. The values of some relevant variables are the same between the original model and the sliced model;
2. The state configuration of the original model has certain relation with that of the sliced model.

5.5.1 The Concept of Relevant Variables

The concept of "relevant variables" has been informally introduced in Section 4.4.1 and Section 4.4.2.1 where it is used as an important set of variables to initiate the initial selection of nodes into the slice in the algorithms. We need to formally define this concept.

Definition 10. *We define v to be a **relevant variable**, written $v \in Rv$, iff there exists $t \in \mathcal{L}$ such that $v \in mv(t)$, and there exists t' such that $v \in cv(t')$, and there exists a sequence of executions $[e_1 \cdots e_k]$ with $t' \subseteq e_1, t \subseteq e_k$ such that $\forall j$ in $1 \leq j < k$, $v \notin cv(e_j)$.*

In other words, a variable is a relevant variable if it appears in at least one data dependency entry in the DD table and there is at least one transition in the slice which uses it.

5.5.2 The Relation between the State Configurations of the Original and the Sliced model

Next, we need to specify the relation between the state configurations of the original and sliced model. Based on the fact that there are true transitions added into $\mathcal{L}$ during the Multi-Stage Model Slicing process[10], we can imagine that 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 some extra execution traces in the sliced model that are impossible in the original model, as stated in the purpose of the proof.

[10] See "Step III: True Transitions Creation" in Section 5.4

However, the state configuration in sliced model is not strictly a superset of that in original model. As there are some states that are not added into the slice by FormlSlicer, these states could not 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:

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

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

So far, we have defined two notions for "projection of snapshots"—relevant variables and state configuration relation. Now, we want to formally define the concept of "projection" as a simulation of the original model by the sliced model based on these two notions.

Recall the concept of "snapshot" in Definition 3. The state configuration and the interpretation of variables form a snapshot of the model.

As discussed in Section 5.2.1, this chapter intends to prove that any relevant variable has same value in both original model and sliced model. Therefore, we will only be concerned about the interpretation of relevant variables in a model.

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

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

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

5.5.4 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, such that if the snapshot in the original model

is projected to the snapshot in the sliced model before the transition, then the snapshot in the original model is still projected to the snapshot in the sliced model after the transition in the original model.

For brevity, in Lemma 1 we will denote $SameParent(t)$ to represent the condition of $(ParentState(ss(t)) = ParentState(ds(t)))$.

Lemma 1. *Consider a transition t in the original model M, $M \vdash t : (N, \sigma) \Rightarrow (N', \sigma')$[11] 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 sliced model is:*

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

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.

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:

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

[11] Recall from Section 5.2.1 that a model M is a big state machine consisting of k FOSMs contained as sub-state machine in 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.

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

When $(t \notin \mathcal{L}) \wedge (\neg SameParent(t))$, the step of Replacing Cross-Hierarchy Transition in General Iterative Slicing Stage will replace t with a "true" transition $t_{true} \in \mathcal{L}$, 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:

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

We can observe that the intersection of $Descendants(ss(t))$ in original model and slice set will be the set of descendants states of $ss(t)$ in sliced model. We can also observe that the intersection of $InitDesc(ds(t))$ in original model and slice set will be the set of default start states of relevant descendant states of $ds(t)$. Given all these observations, the condition (8), (11) and (12), we can deduce that $N' \cap \mathcal{L} \subseteq N'_{\mathcal{L}}$. In other words, the sliced model evolves through this newly added t_{true} in the same way as how the original model evolves through t, in terms of state configuration. Therefore,

$$N' \cap \mathcal{L} \subseteq N'_{\mathcal{L}}. \tag{13}$$

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 $t_x \in \mathcal{L}$ 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 $t_x \xrightarrow{dd}_v t, t_x \in \mathcal{L}$, so $t \in \mathcal{L}$ which contradicts with the given condition $t \notin \mathcal{L}$. In addition, $\sigma_{\mathcal{L}} = \sigma'_{\mathcal{L}}$ remains unchanged because t_{true} does not change values of any variables in the sliced model. Together with condition (9), we get

$$\sigma'_{\mathcal{L}}(v) = \sigma_{\mathcal{L}}(v) = \sigma(v) = \sigma'(v) \forall v \in Rv. \tag{14}$$

Given both (13) and (14), 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 $(ss(t) \in \mathcal{L}) \wedge (ds(t), t \notin \mathcal{L}) \wedge (SameParent(t))$, 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.

$$(SameParent(t)) \Rightarrow (AncestorTillLCA(ss(t), ds(t)) = AncestorTillLCA(ss(t), ds(t)) = \emptyset) \tag{15}$$

$$(ds(t) \notin \mathcal{L}) \Rightarrow (InitDesc(ds(t)) = \emptyset) \tag{16}$$

Because of (15) and (16), we get:

$$entered(t) \cap \mathcal{L} = \emptyset \tag{17}$$

Because of (17), (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

$$N' \cap \mathcal{L} \subseteq N_{\mathcal{L}}. \tag{18}$$

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

$$\sigma_{\mathcal{L}}(v) = \sigma'(v) \forall v \in Rv. \tag{19}$$

Given both (18) and (19), 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 $(ss(t), ds(t), t) \notin \mathcal{L} \wedge (SameParent(t))$, 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 projecting state configuration in the original model to that in the sliced model will be exactly the same as in the Case (2). Therefore:

$$N' \cap \mathcal{L} \subseteq N_{\mathcal{L}}. \tag{20}$$

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

$$\sigma_{\mathcal{L}}(v) = \sigma'(v) \forall v \in Rv. \tag{21}$$

Given both (20) and (21), 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 $(ss(t), t \notin \mathcal{L}) \wedge (ds(t) \in \mathcal{L}) \wedge (SameParent(t))$, 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:

$$AncestorTillLCA(n, ds(t)) = AncestorTillLCA(n_i, n_j) = \emptyset . \forall (i, j \in [1 \cdots k]) \wedge (i \neq j) \tag{22}$$

$$(ds(t) \in \mathcal{L}) \Rightarrow (InitDesc(ds(t)) \cap \mathcal{L} \subset InitDesc(ds(t))) \tag{23}$$

In the sliced model, the "true" transition has changed the state configuration such that $ds(t)$, $InitDesc(ds(t)) \cap \mathcal{L}$ will be entered and n and $Descendants(n)$ will be exited (because of (22), 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 (22), (23), and (8), we can get that

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

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. Together with condition (9), we get

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

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 t_{true} in the sliced model.

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

When $(ss(t), ds(t) \in \mathcal{L}) \wedge (t \notin \mathcal{L})) \wedge (SameParent(t))$, 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:

$$N' \cap \mathcal{L} \subseteq N'_{\mathcal{L}}. \tag{26}$$

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 $t \notin \mathcal{L}$ is that it does not control any relevant variables and thus is not selected into the slice during the DD Step. Similarly, we get:

$$\sigma'_{\mathcal{L}}(v) = \sigma'(v) \forall v \in Rv. \tag{27}$$

Given both (26) and (27), 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 $(n_{merged} = (ss(t) \vee ds(t)) \in \mathcal{L}) \wedge (t \notin \mathcal{L})$, $ss(t)$ and $ds(t)$, 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:

$$n_{merged} = (ss(t) \vee ds(t)) \tag{28}$$

$$n_{merged} \in N_{\mathcal{L}} \tag{29}$$

$$\text{Because (28) and (29), } ds(t) = n_{merged} \in N_{\mathcal{L}} \tag{30}$$

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 $ss(t), ds(t), n_{merged}$[12], p must have been added to the slice set. Because of (29) and the CD-HD Step,

$$p \in N_{\mathcal{L}}, \forall p.(n_{merged} \xrightarrow{hd} p) \tag{31}$$

$$\text{Because (31), } AncestorTillLCA(ds(t), ss(t)) \subseteq N_{\mathcal{L}} \tag{32}$$

[12]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.

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:

$$InitDesc(ds(t)) \subseteq N_{\mathcal{L}} \tag{33}$$

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 (30), (33) and (32). Given that (8), we can deduce that $N' \cap \mathcal{L}$ remains the same subset relation with $N_{\mathcal{L}}$, as

$$N' \cap \mathcal{L} \subseteq N_{\mathcal{L}} \tag{34}$$

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

$$\sigma'(v) = \sigma'_{\mathcal{L}}(v) \forall v \in Rv \tag{35}$$

Given both (34) and (35), 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.

All Other Cases This proves for **Case** (7).

In all other cases, t will be preserved in the sliced model, i.e., $t \in \mathcal{L}$. According to the Transition-to-State Step in General Iterative Slicing Stage, $ss(t), ds(t) \in \mathcal{L}$. The state configuration in original model changes in the same fashion as that in the sliced model. Because $t \in \mathcal{L}$, 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.

■

5.5.5 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 Lemma 1, we have proved that one transition in original model can always be projected to epsilon or one transition in sliced model, so that the projection of the original model's snapshot to the sliced model's snapshot is maintained through the transition in original model.

We can now compose the projections of concurrent transitions together:

$$P(e) = P\begin{pmatrix} t_1 \\ \vdots \\ t_k \end{pmatrix} = \begin{pmatrix} P(t_1) \\ \vdots \\ P(t_k) \end{pmatrix}$$

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

The projection of original model's snapshot to the sliced model's snapshot is maintained through the execution step e in original model.

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 start states of relevant descendant states of the root state. $\sigma^I(v)$ 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 minus all the states that are not selected into the slice. We also know that the values of all relevant variables in the sliced model are also at their default or uninitialized values. Therefore, $N^I \cap \mathcal{L} \subseteq N^I_{\mathcal{L}}$ and $\sigma^I(v) = \sigma^I_{\mathcal{L}}(v) \forall v \in Rv$. Then we can conclude that the initial snapshot in the original model can be projected to the initial snapshot in the sliced model.

Since the initial snapshot in the original model can be projected to the initial snapshot in the sliced model, and the projection of original model's snapshot to the sliced model's snapshot is maintained through one execution step e 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.

Chapter 6

Empirical Evaluations of FORML Slicer

See Appendix A.

Chapter 7

Conclusion

Appendix A

Automotive: A Slicing Example

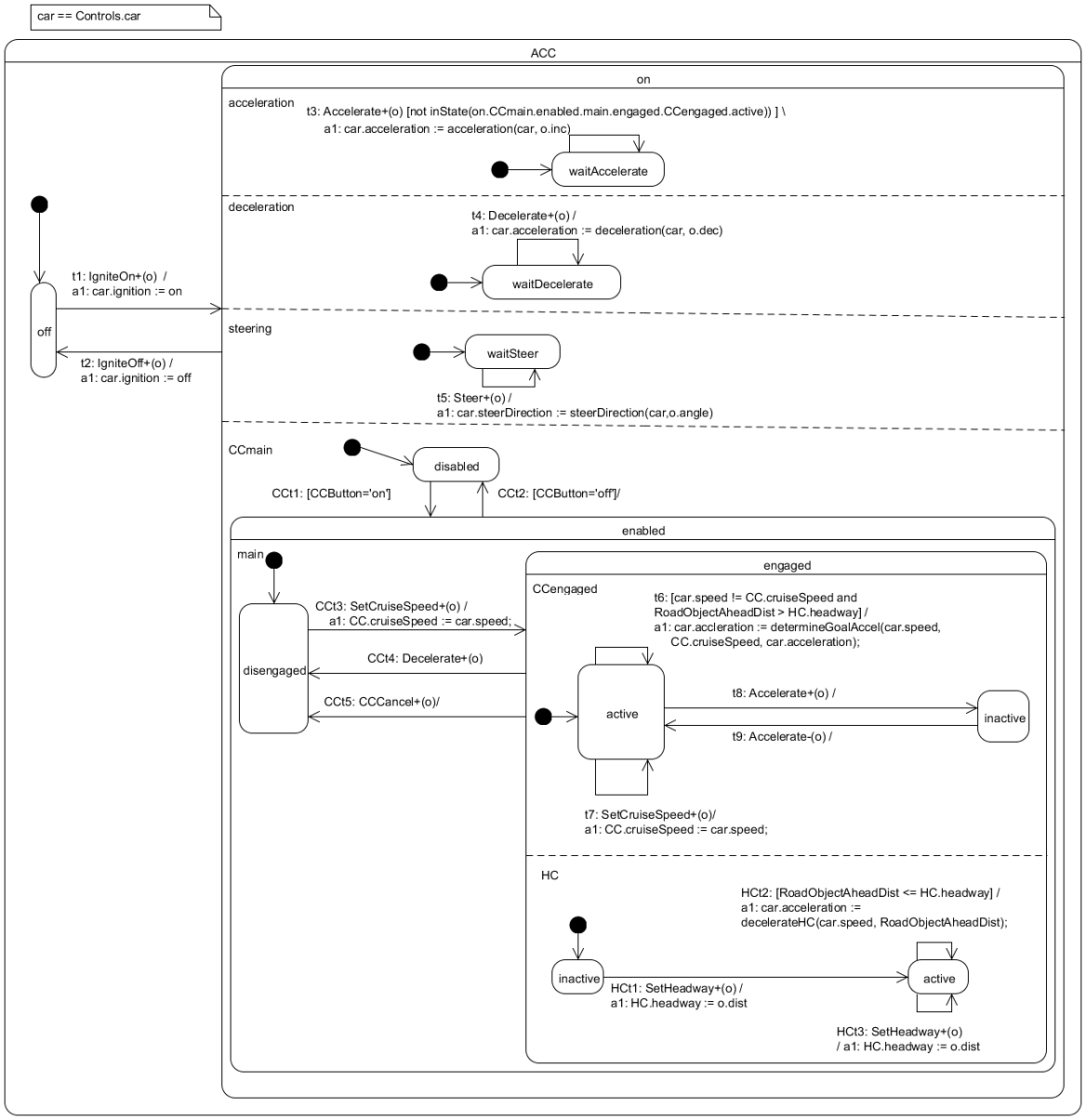


Figure A.1: Original Adaptive Cruise Control (ACC) feature

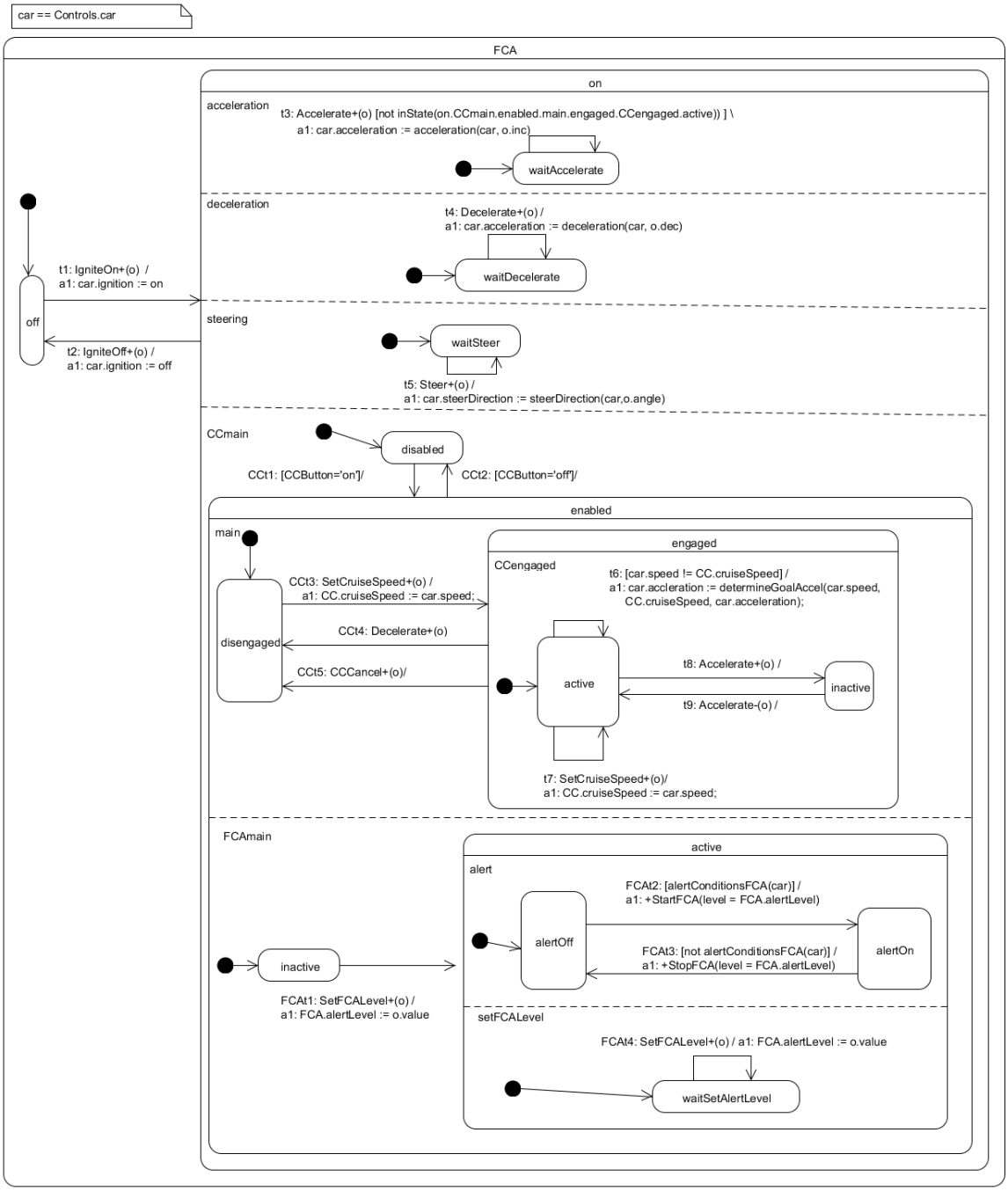


Figure A.2: Original Forward Collision Alert (FCA) feature

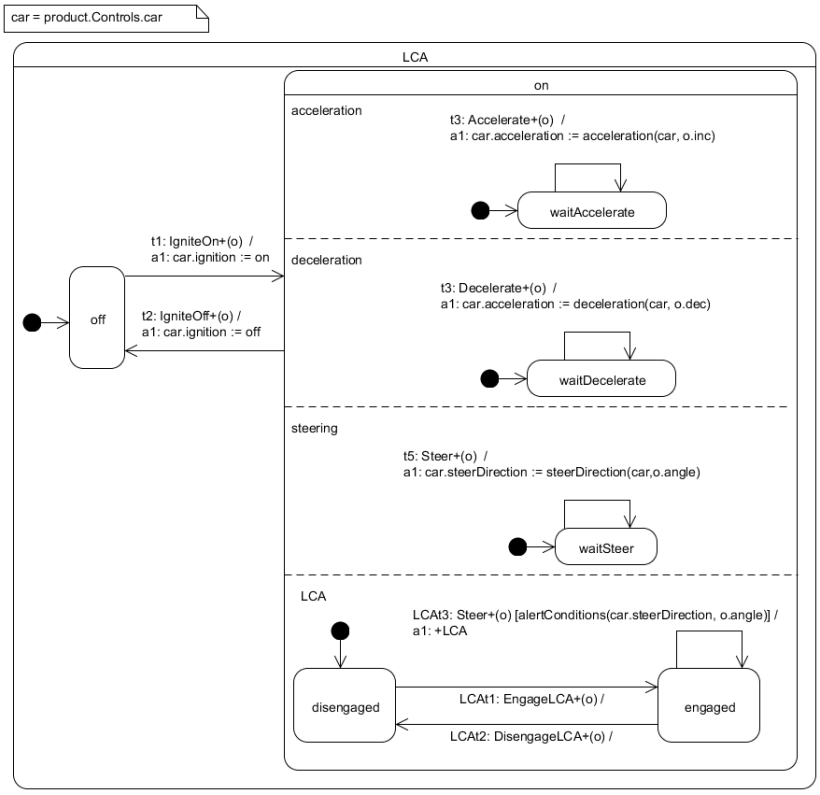


Figure A.3: Original Lane Change Alert (LCA) feature

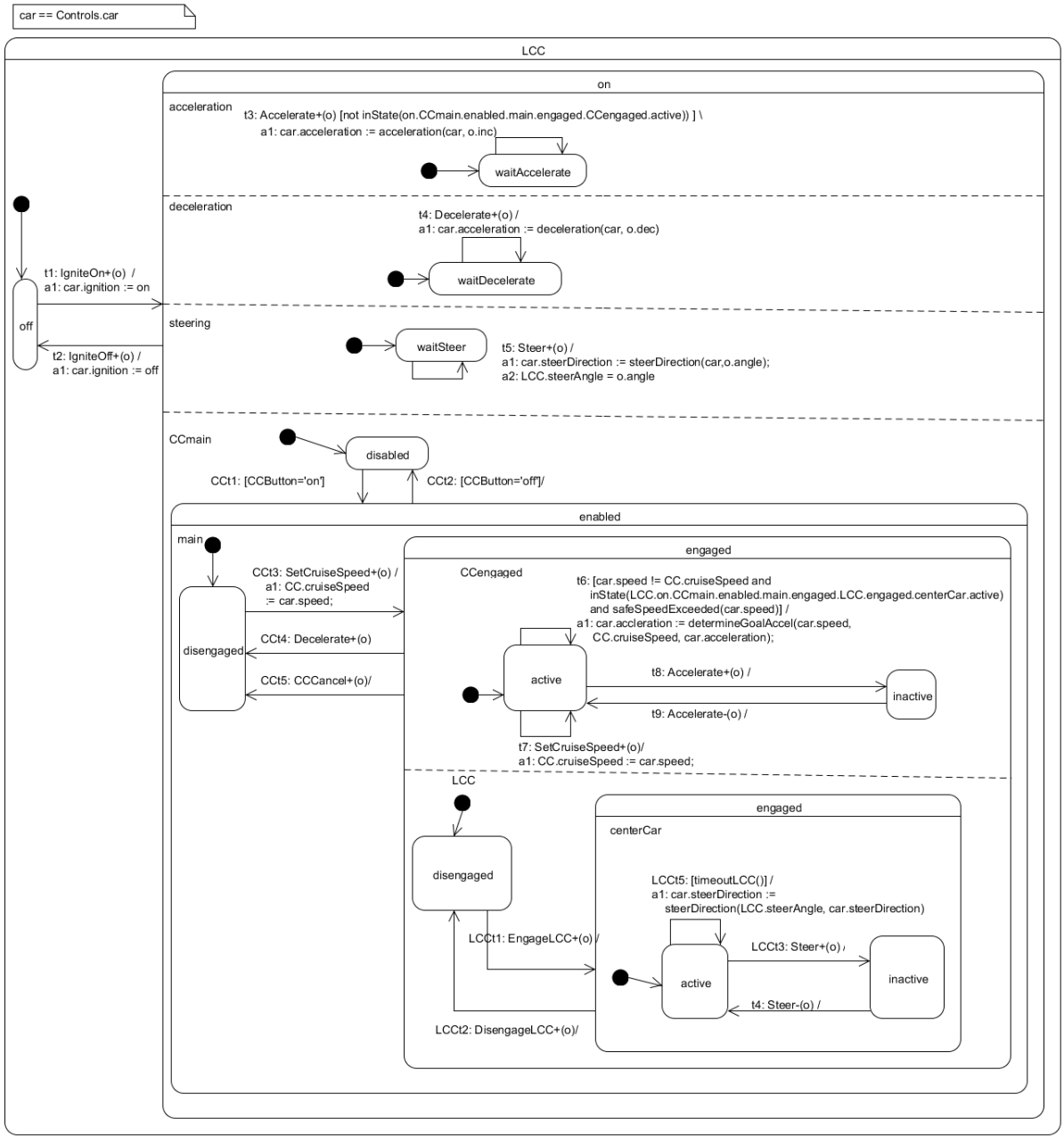


Figure A.4: Original Lane Centring Control (LCC) feature

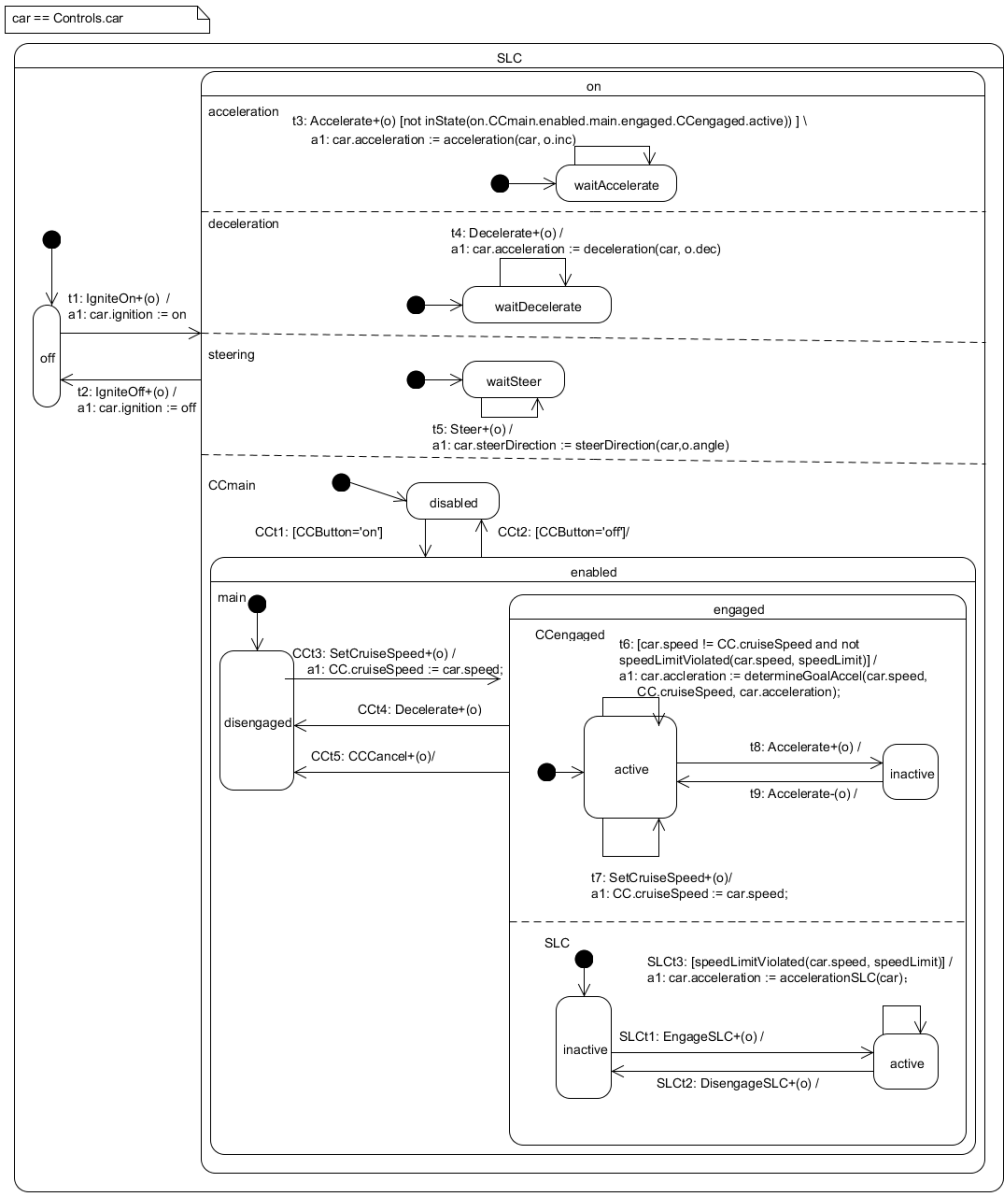


Figure A.5: Original Speed Limit Control (SLC) feature

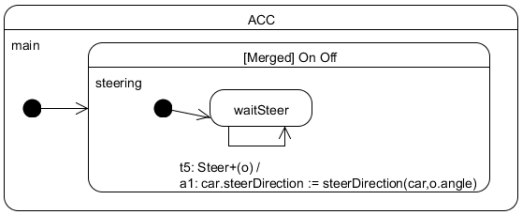


Figure A.6: Sliced ACC feature w.r.t. LCA

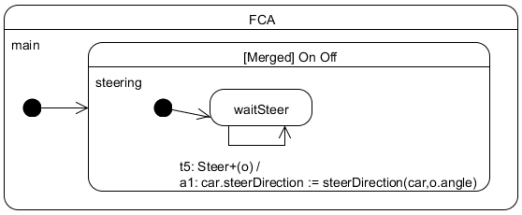


Figure A.7: Sliced FCA feature w.r.t. LCA

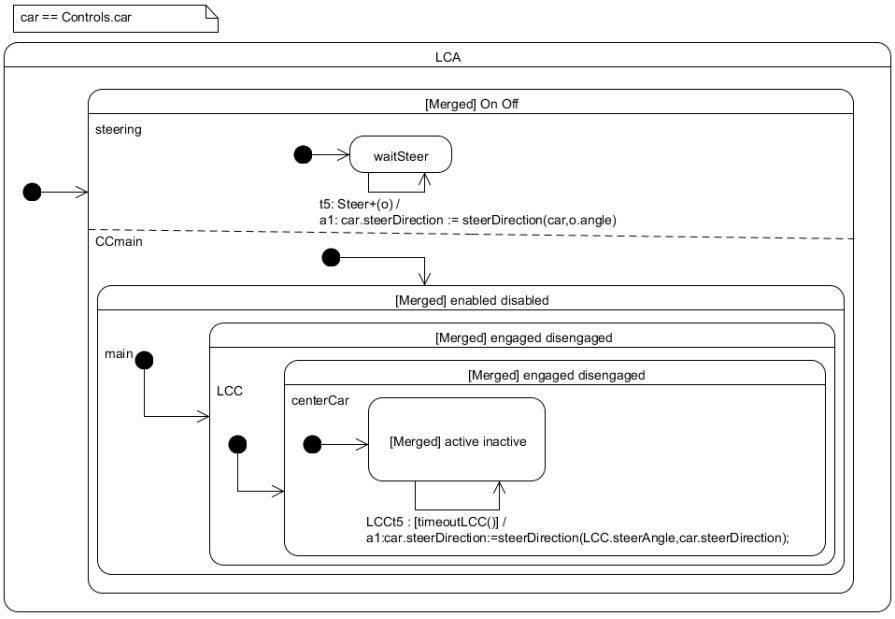


Figure A.8: Sliced LCC feature w.r.t. LCA

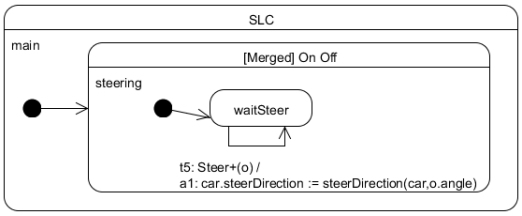


Figure A.9: Sliced LCC feature w.r.t. LCA

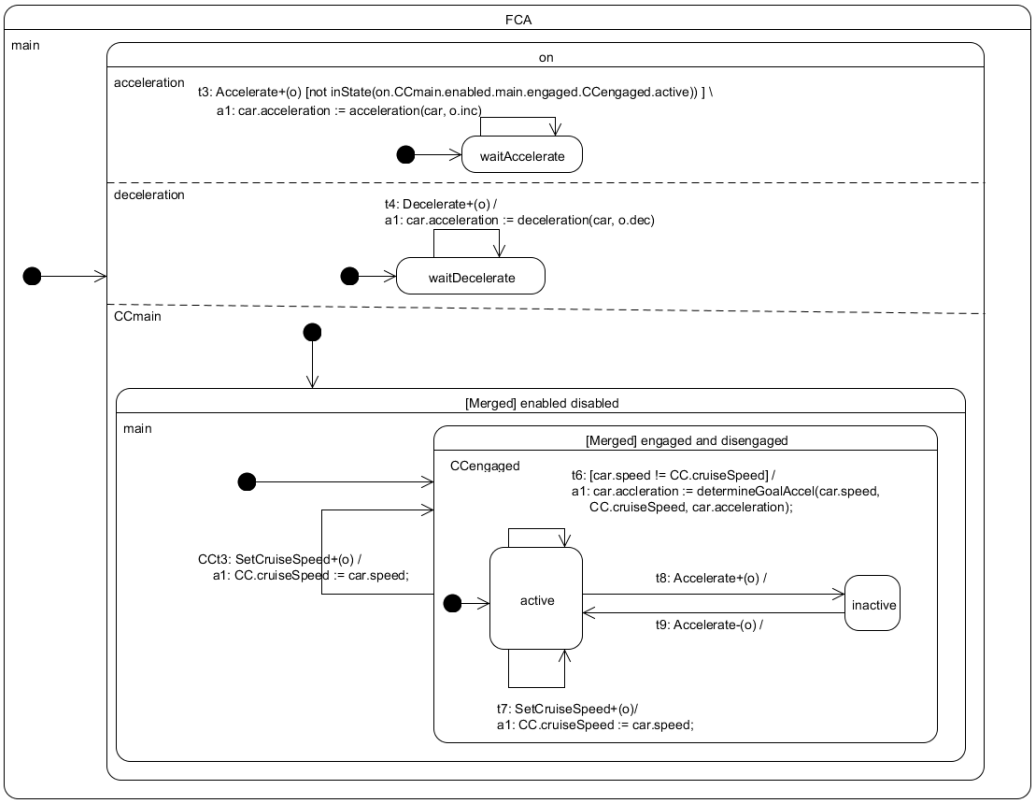


Figure A.10: Sliced FCA feature w.r.t. ACC

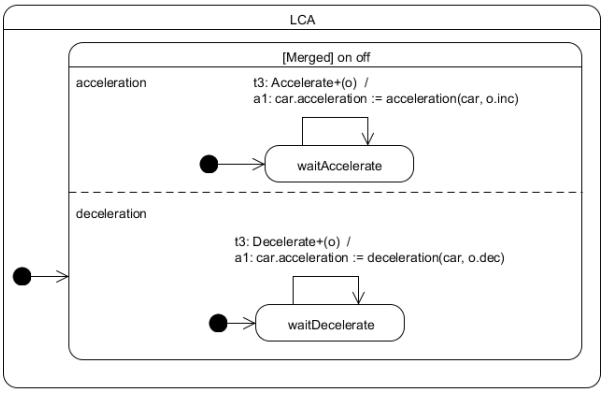


Figure A.11: Sliced LCA feature w.r.t. ACC

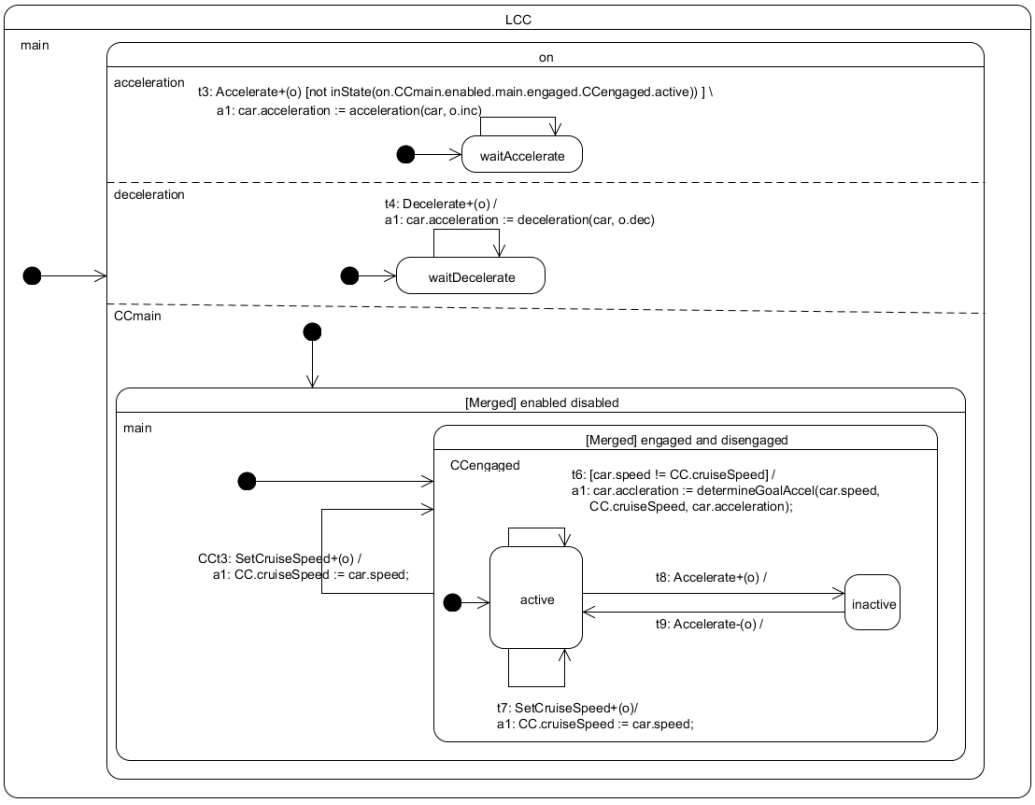


Figure A.12: Sliced LCC feature w.r.t. ACC

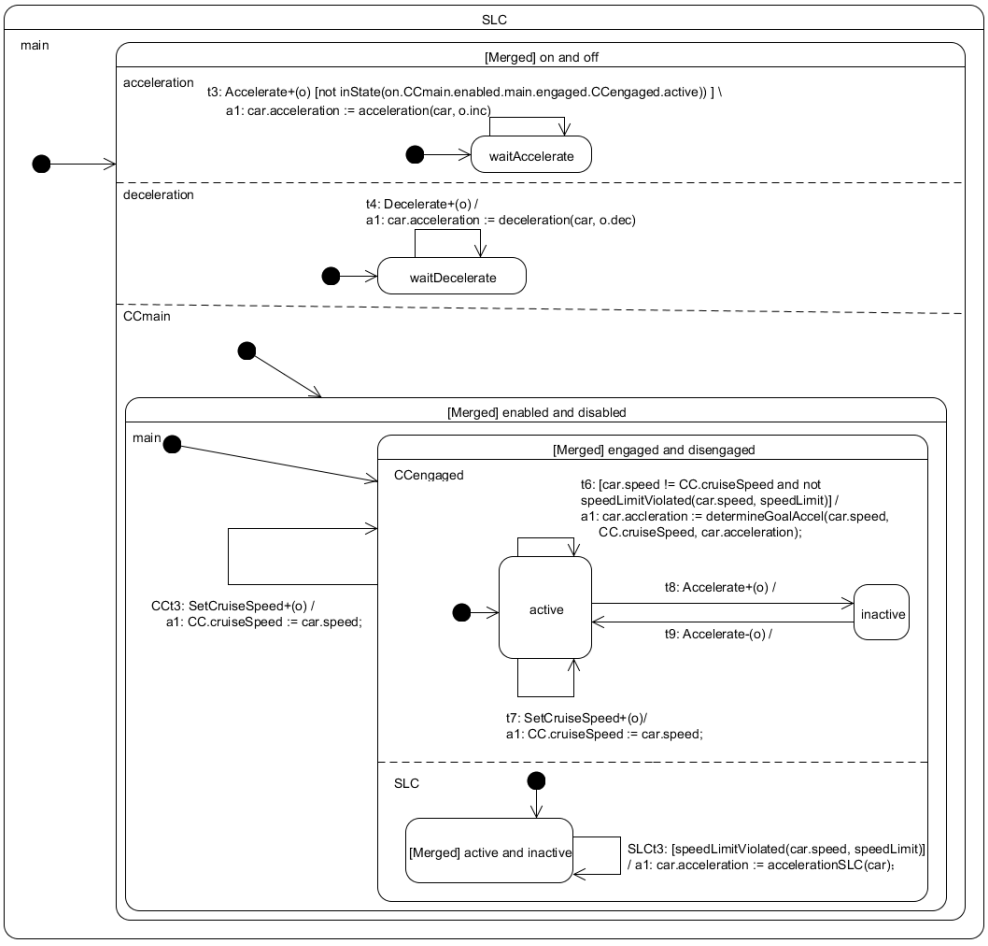


Figure A.13: Sliced SLC feature w.r.t. ACC

Appendix B

Some Appendix

The contents...

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 4.2.

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] 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.

[2] 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.

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

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] University of Waterloo. Necsis: Managing variability and configurability in an mde development process. http://gsd.uwaterloo.ca/necsis. Accessed: 2015-02-03.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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