# 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

## 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 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 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. xi, xii, 86, 91–94

**BDS** Basic Driving Service. 22

**CC** Cruise Control. 22

**CD** Control Dependence. 11, 14, 37, 41

**CFG** Control Flow Graph. 7, 10, 15, 17, 18, 28, 30, 31, 33, 38, 43, 53

**DD** Data Dependence. 11, 14, 23, 26, 30, 34, 35, 74

**EFSM** Extended Finite State Machine. 2, 11, 14, 17, 19

**FCA** Forward Collision Alert. xi, xii, 87, 91, 92

**FOI** Feature of Interest. 5–8, 24, 27, 28, 35, 44, 46, 47, 62, 63, 72, 74

**FORML** Feature-oriented Requirements Modelling Language. 7, 8, 12, 21–23, 25, 28, 30, 32, 33, 37, 47, 55, 62, 65, 83

**FOSD** Feature-oriented Software Development. 4, 21

**FOSM** Feature-oriented state machine. x, 7, 8, 22–25, 28, 30–33, 35, 38, 46, 47, 55, 63, 65, 67, 72

**HC** Headway Control. 4

**HD** Hierarchy Dependence. 30, 34

**LCA** Lane Change Alert. xi, xii, 88, 91–93

**LCC** Lane Centring Control. xi, xii, 89, 91–93

**NTSCD** Non-Termination Sensitive Control Dependence. 16–18, 38

**ROS** Rest of System. 5–7, 24, 27, 28, 46, 47, 62, 63

**SBM** State-Based Model. 2, 3, 7, 11, 12, 14, 22, 38

**SLC** Speed Limit Control. xi, xii, 4, 90, 94

**SNode** A Node in CFG representing a state in FORML model. 32, 33, 43, 46, 53

**SPL** Software Product Line. 3, 4, 6–8, 21, 66

**TNode** A Node in CFG representing a transition in FORML model. 32, 33, 44, 46, 53

**WCA** World Change Action. 23, 25, 26, 32, 70

**WCE** World Change Event. 23–25, 32, 70

# Chapter 1

# Introduction

## 1.1 Model Slicing

### 1.1.1 What Is Model Slicing

In English, “slicing” usually refers to the act of cutting a portion from something larger with a sharp implement. In the software engineering research community, the word was firstly adopted to refer to the reduction of a software program to its minimal form which still produces a subset of the program’s behaviour [2]. Since then, program slicing becomes a well-discussed topic; various slightly different notions of program slices have been proposed, as well as different methods to compute them [1]. In general, all of these methods try to extract parts of the original program that may influence a variable at some point of interest. After more than thirty years of development, program slicing has gradually become a mature source code analysis and manipulation technique, which can be used in software debugging, software maintenance, optimization, program analysis and information control flow.

As software production nowadays becomes more sophisticated, models that are used in specification become more unwieldy in scale. In recent years, researchers have begun to consider slicing from a program level to a model level [3]. Similar to program slicing, model slicing extracts a sub-model from the original model while preserving the behaviour with respect to a point of interest. Because of the graph-based nature and other features of models that are different from software program, model slicing needs to be considered as a distinct research area.

Among a diverse range of software modelling languages, State-Based Model (SBM) receives an abundant amount of attention from researchers. SBM is an umbrella term for a wide-range of related languages (Extended Finite State Machine (EFSM), UML state machines, etc.) [3]. These models are based on finite state machines which depict some sequences of execution. There are many variants on this notion; but in general, an SBM consists of a finite set of states, a set of transitions that moves from current state to next state based on an event. Many languages have defined additional features, such as global variables, hierarchical constructs, or parallelism constructs.

This thesis will mainly concern slicing on State-Based Models. In general, it is a procedure in reducing **the original model** to a smaller one, called **the sliced model**, so that the sliced model contains less number of transitions, states, or other model elements than that in the original model. At the same time, the sliced model preserves a subset of behaviours of the original model with respect to a **slicing criteria**. Usually, the slicing criteria refers to a set of relevant variables; it means that in spite of a smaller size, the sliced model can output the same values of these variables as the original model. Lastly, the sliced model must maintain its well-formedness as an independent SBM; it means that the model slicing procedure needs to "rewire" the SBMs to prevent nodes being orphaned.

### 1.1.2 Properties of a Useful Sliced Model

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

**correctness**
: The sliced model can simulate the original model with respect to the slicing criteria;

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

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

The property of correctness is non-negotiable. The slicer is incorrect if it produces a sliced model which cannot simulate the original model with respect to a user-defined slicing criteria. Some may define correctness in a stronger sense such that not only the

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

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

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

For slicing on an SBM, if we maximize the degree of compactness and minimize the degree of precision, we may obtain a single "super" state in the sliced model such that all the original transitions become self-looping transitions of the "super" state [5]. If, on the other hand, we minimize the degree of compactness and maximize the degree of precision, we may obtain a sliced model which is exactly same as the original model. Either of these two sliced models are correct[3], but not useful for comprehension purpose. Engineers will gain no useful information from either of these two sliced models.

Therefore, to make a sliced model useful, a good model slicer is supposed to strike a balance between precision and compactness. A sliced model must have appropriate trade-off between the properties of precision and compactness, while it maintains the correctness property.

## 1.2 Feature-oriented Model Slicing

The model slicing techniques can be applied in many different domains. This thesis will only focus on one specific domain—feature-oriented requirements in Software Product Line (SPL).

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

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

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

### 1.2.1 Feature Interactions in SPL Requirements

Many organizations develop a family of similar software systems in the same domain, called an SPL. For example, an automotive manufacturer can design a series of new vehicle models that are close variants of one another. These software systems have different combinations of features. For example, the 2015 Buick Encore premium model is equipped with the Forward Collision Alert feature whilst the convenience model in the same SPL is not; yet both convenience and premium models have many other common features, such as the Traction Control feature.

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

One well-known challenge of applying the FOSD paradigm in SPL requirements is managing feature interactions. In software requirements, features are usually 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 [7]. In a single system, such feature interactions can still be pre-determined during the modelling stage. But in an SPL, different system variants can be equipped with different subsets of all possible 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 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 [8].

### 1.2.2 Model Slicing used in Feature-oriented Requirements in SPLs

From Subsection 1.2.1, we have learnt that an SPL model can be modelled as a finite set of feature modules, each representing one feature's requirements. Also, we have learnt that detecting unintended feature interactions in an SPL model is a highly complicated task.

This is where model slicing comes into play as a useful technique.

Slicing can be performed on an SPL model with respect to only one feature, called the Feature of Interest (FOI), so that the huge SPL model can be reduced to a feasible size, while it still preserves the behaviour of the FOI. The slicing criteria is the variables related to the FOI. All the other features except the FOI form the Rest of System (ROS) in the SPL model; only a relevant portion of them will be kept after slicing. The (possibly) smaller ROS in a sliced SPL will form a "smaller context" for the FOI.

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

#### 1.2.2.1 Model Comprehension of Feature-oriented Requirements in SPLs

As described in Subsection 1.2.1, features are usually modelled in isolation by different groups of software engineers. When there are hundreds of features, which is typically the case in an SPL, it is likely that there are unintended feature interactions. In an industrial world, it is much better to detect and resolve these unintended feature interactions at a requirements stage, rather than at an implementation stage, in which the cost is substantially higher. However, engineers will be overwhelmed if they try to comprehend the whole SPL model. There is a fundamental limitation of human capacity to deal with such a highly complex model because there are far too many details for a single person to keep track at once [9].

Feature-oriented model slicing can facilitate model comprehension on one feature (i.e., the FOI) at a time. Because it removes irrelevant details of other features in the ROS and only keeps the portions that may (potentially) interact with the FOI, engineers can focus on understanding the FOI together with a smaller ROS. It will be much easier to detect any unintended feature interactions between the FOI and other features.

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

#### 1.2.2.2 Compositional Verification of SPLs

To ensure safe operation of all system variants in an SPL, there is a need to provide effective and scalable means of understanding and managing feature interactions. Formal method techniques like model checking have been proposed to detect feature interactions in SPL. In the past, several strategies for SPL analysis have been proposed, in particular, family-based, product-based and feature-based strategies [10]. Family-based strategy exploits commonality among products in an SPL and delivers sound and complete analysis results; but it is often computationally infeasible as the whole SPL is too huge to be 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 smaller, relevant ROS. In each iteration of analysis, a feature (i.e., the FOI) and its smaller ROS are fed into a model checker to check whether a set of safety properties related to the FOI is maintained in all executions. The benefits of this compositional verification are evident: linear number of models to check, possible parallel verification. Also, feature interactions are taken into considerations and this can yield complete verification result.

Model slicing is an essential technique in achieving compositional verification. It creates the FOI's smaller, relevant ROS. Because feature interactions between FOI and ROS is included in model verification, the combined verification results from individual feature verification can be complete.

## 1.3 Thesis Overview

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

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

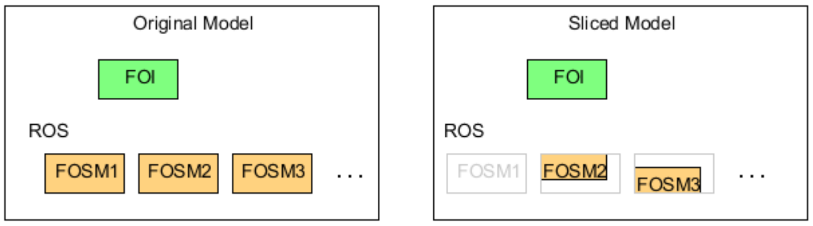


**Figure 1.1:** A Comparison of the SPL Model before and after Slicing by FormlSlicer

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

FormlSlicer undergoes multiple stages to produce the sliced model. It is divided into two tasks. In the preprocessing task, it needs to parse and process the whole original model into a set of Control Flow Graph (CFG)s so that it can compute the dependency relationship among the nodes in CFGs easily. In the slicing task, multiple slicing processes are created, each process treats one FOSM as the FOI, and slices all the other FOSMs with respect to the FOI. Since the slicing process starts from emptiness and gradually include model elements into slice, the FOSMs in the sliced model might become disconnected state-based machines; thus the slicing processes will undergo an "enrichment" stage to make these FOSMs to become well-formed state-based machines.

### 1.3.1 Thesis Statement

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

- the preprocessing task, that parses the FOSMs and compute dependencies,
- and the slicing task, that gradually adds model elements from the original model into the sliced model and enriches it at the end;

in order to produce a sliced model

- that can simulate the original model while keeping the constraints raised from feature interactions with the FOI,
- that achieves a satisfactory degree of compactness and precision,
- and that each FOSM in it remains as a well-formed finite state machine;

so that the sliced models generated by repeating the slicing task for all features can be useful in detecting unintended feature interactions in the SPL.

## 1.4 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 FORML, as well as other important concepts used extensively throughout the thesis, e.g., 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.

Chapter 5 is the theoretical evaluation of FormlSlicer. It takes on an abstract approach to prove the correctness of FormlSlicer, by showing how a sliced model produced by FormlSlicer can simulate its original model.

Chapter 6 is the empirical evaluation of FormlSlicer. There are two empirical evaluations performed on FormlSlicer.

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

# Chapter 2

# Related Work

## 2.1 Program Slicing

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

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

Figure 2.1a [1] is an example program to show the effect of program slicing. It computes both the sum and the product of the first n positive numbers. Figure 2.1b shows its program slice with respect to a slicing criterion of (12,product). We can see that all computations that are irrelevant to the value of variable at Line 12 have been "sliced away".

Program slicing has been further divided into different categories. There is a distinction between static slicing and dynamic slicing. The former is computed without making assumptions regarding a program's input, whereas the latter relies on some selected variables and inputs [12]. Another distinction is made between forward slicing and backward slicing.

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

**(a)** The Original Program

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

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

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

write(product);
```

**(b)** Program Slice w.r.t. (12,product)

**Figure 2.1:** An Example Program [1]

Informally, a forward slice consists of all statements and control predicates dependent on the slicing criterion, i.e., the program subset that is influenced by the slicing criterion; a backward slice is the program subset that may affect the slicing criterion.

In this thesis, we will only concern static slicing and backward slicing.

### 2.1.1 Dependence-based Slicing

Slices are computed by finding consecutive sets of transitively relevant statements, according to data flow and control flow dependencies. This is generally referred to as dependence-based slicing [13], because it involves the computation of dependence relations between the program statements. Using dependence relations information, slicing can be reduced to a simple reachability problem.

There are different computation approaches based on different graph representations of a program. Control Flow Graph (CFG) is one graph representation to capture the reachability of program statements in a program. A CFG contains a node for each statement and control predicate in the program; an edge from node $i$ to node $j$ indicates the possible flow of control from the former to the latter. Dependencies are defined in terms of the CFG of a program.

Dependencies arise as the result of two separate effects [14]:

1. First, a dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed. In the example program in Figure 2.1a, if Line 12 appears before Line 4, the value of `product` written will be incorrect at Line 12. Therefore we know that the statement at Line 12 is dependent on the statement at Line 4. This is Data Dependence (DD).

2. Second, a dependence exists between a statement and the predicate whose value immediately controls the execution of the statement. In the example program in Figure 2.1a, the statement at Line 8 is dependent on the predicate at Line 5 since the condition in the while loop at Line 5 determines whether the statement at Line 8 is executed. This is Control Dependence (CD).

## 2.2 Slicing on State-Based Models (SBMs)

As software production nowadays becomes more sophisticated, models that are used in specification become more unwieldy in scale. In recent years, researchers have begun to consider slicing from a program level to a model level [3]. Similar to program slicing, model slicing extracts a sub-model from the original model while preserving a subset of the original model's behaviour with respect to a slicing criterion.

### 2.2.1 What Is an SBM

Among a diverse range of software modelling languages, State-Based Model (SBM) receives an abundant amount of attention from researchers. SBM is an umbrella term for a wide-range of related languages (EFSM, UML state machines, etc.) [3]. These models are based on finite state machines which depict some sequences of execution. There are many variants on this notion; many languages have defined additional features, such as global variables, hierarchical constructs, or parallelism constructs [15].

The basic definition of an SBM comes from Mealy machine [16]. In general, an SBM consists of a finite set of states (including start states), a set of events (or "inputs") and a transition function that determines the next state based on the current state and event. Each transition has a source state, a destination state and a label. The label is of the form $e[g]/a$, where each part is optional: $e$ is the event necessary to trigger a possible change of state; $g$ is the guard (i.e. a boolean expression) that further constraints a possible change of state; and $a$ is a sequence of actions (mostly updates to variables in the environment or generation of events) to be executed when the transition occurs.

The notion of an SBM with hierarchy and parallelism constructs has been introduced long time ago [15]. In a hierarchical state machine, a state may be further refined into another sub-state machine; this is a composite state. The hierarchy can be arbitrarily deep. If the state machine incorporates parallelism construct into its semantics, a state can consist of multiple orthogonal regions, each containing a sub-state machine; the sub-state machines are executing concurrently. Researchers have identified that the states in a sub-state machine follow an XOR relationship (i.e., it can be in exactly one state at a time) and the orthogonal regions in a state follow an AND relationship (i.e., when the system is in the state, it must be in all of its containing regions) [15]. Communication between concurrent SBMs can be synchronous (the SBM blocks until the receivers consume the event) or asynchronous (non-blocking).

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

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

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

### 2.2.2 Challenges of SBM Slicing Compared to Program Slicing

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

The first difference is the level of granularity between a program and an SBM [3]. Program slicing often operates at a line of code, which is a natural level of granularity. Program slices are thus typically subsets of the lines of code in the original program. However, the level of granularity in an SBM is unclear: an individual node may represent the equivalent of several lines of code, or several nodes may represent the equivalent of a single line of code. Because of this difference of granularity, it is very hard to automate the translation of an SBM into a program.

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

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

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

### 2.2.3 Dependencies in SBM Slicing

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

DD is the most commonly mentioned dependence in the model slicing works. DD in an EFSM (one type of SBM languages) has been defined as [5]: DD captures the notion that one transition defines a value to a variable and another transition may potentially use this value. More formally, transition $T_k$ is dependent on $T_i$ with respect to variable $v$ if (1) $v$ is defined by $T_i$, (2) $v$ is used by $T_k$, and (3) there exists a path (transition sequence) in the EFSM model from $T_i$ to $T_k$ along which $v$ is not re-defined; such a path is referred to as *definition-clear path* [5]. This definition on DD will be used in this thesis.

CD in an SBM is a lot more complicated. It will be elaborated in a separate subsection 2.2.3.

Besides DD and CD, researchers have used a large variety of different dependencies in their model slicers [17, 19]. Most of these dependencies arise due to more advanced constructs added onto a basic SBM. We presents some of them here:

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

Unlike DD and CD which serve for more general purposes in model slicing, each of the above-mentioned dependencies is only restricted to some advanced SBMs with special semantics. In spite of this, they are useful in showing that one of the best ways to cope with an SBM with more complex semantics is to improve the dependence-based slicing technique by including more types of dependencies.

## Control Dependence

In program slicing, control dependence is traditionally defined in terms of a post-dominance relation in a CFG [14, 20]. Intuitively, a node $n_i$ is post-dominated by a node $n_j$ in a CFG if in every execution of the program, $n_j$ always occurs after $n_i$. Having calculated the post-dominance relation in a CFG, the control dependence relation can be obtained according to this: a statement $n_j$ is said to be control dependent on a statement $n_i$ if there exists a nontrivial path $p$ from $n_i$ to $n_j$ such that every statement $n_k \neq n_i$ in $p$ is post-dominated by $n_j$ and $n_i$ is not post-dominated by $n_j$.

Although this definition of control dependence is widely used in program slicing, there are some other slightly different definitions as well. Researchers have distinguished two different notions on control dependence—strong control dependence and weak control dependence [21]:

1. $n_j$ is strongly control dependent on $n_i$ if there is a path from $n_i$ to $n_j$ that does not contain the immediate post-dominator of $n_i$;

2. $n_j$ is weakly control dependent on $n_i$ if $n_j$ strongly post-dominates $n_k$, a successor of $n_i$, but does not strongly post-dominates $n_l$, another successor of $n_i$.

The notion of strong control dependence is similar to the traditional notion, except that it captures both direct and indirect control dependences. For example, if $n1$ is control dependent on $n2$ and $n2$ is control dependent on $n3$, then $n3$ is strongly control dependent on $n1$, which is not the case in the traditional definition of control dependence. Other than that, it roughly corresponds to the traditional definition in [14, 20].

Weak control dependence shows a dependence relationship between the predicate condition of a loop (i.e. the branching node in a CFG) and a statement outside the loop that may be executed after the loop is exited, while strong control dependence does not [21]. Re-consider the program in Figure 2.1a; the statement at Line 11 is weakly control dependent, but not strongly control dependent, on the predicate condition at Line 5.

As we can see, both the traditional control dependence and strong control dependence ignores the possibility of an infinite loop. They determine that a statement outside the loop (e.g. Line 11 in Figure 2.1a) must always be executed after the predicate condition (e.g. Line 5), and therefore the latter is post-dominated by the former. In other words, they are *non-termination insensitive* [22]. However, they are widely used in most program slicing tasks which focus on debugging and program visualization and understanding; these slicing tasks do not consider preserving non-termination properties as an important requirement.

Because the post-dominance relation assumes a single end node in the CFG, all of these definitions of control dependence assume that the single end node property in CFG is satisfied.

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

**Definition** (Control Dependence [5])**.** *Transition $T_i$ has control dependence on transition $T_k$ (transition $T_k$ is control dependent on transition $T_i$) if (1) $T_k$'s source state does not post-dominate $T_i$'s source state, and (2) $T_k$'s source state post-dominates transition $T_i$.*

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

Later, some researchers in program slicing community have recognized that these traditional notions on control dependence are no longer applicable to most modern programs. Ranganath et al.[22] have identified two trends in modern program structures:

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

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

Therefore, in most modern programs, their corresponding CFGs do not satisfy the single end node property. Ranganath et al.[22] proposed a new definition on control dependence:

**Definition** (Non-Termination Sensitive Control Dependence (NTSCD) [22])**.** *In a CFG, $n_j$ is (directly) non-termination sensitive control dependent on node $n_i$ if $n_i$ has at least two successors, $n_k$ and $n_l$,*

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

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

The key observation on NTSCD is that reaching again a start node in a loop is analogous to reaching an end node. This intuition is similar to weak control dependence [21] which considers that an infinite loop is important in determining control dependence. This implies that NTSCD is *non-termination sensitive* [22], like the notion on weak control dependence. In addition, NTSCD gets rid of the concept of post-dominance and uses the concept of a maximal path. A maximal path is any path that terminates in a final transition, or is infinite [23]. The consideration of maximal paths imply that NTSCD is applicable to CFGs that do not satisfy the single end node property.

There are more "advanced" definitions that extend the definition on NTSCD. In summary, these definitions replace the "maximal paths" in NTSCD to other types of paths [23].

- By replacing "maximal paths" with "sink-bounded paths" we obtain the definition on Non-Termination Insensitive Control Dependence (NTICD). This definition does not calculate control dependencies within control sinks[1] [22].
- By replacing "maximal paths" with "unfair sink-bounded paths" we obtain the definition on Unfair Non-Termination Insensitive Control Dependence (UNTICD). This definition is in essence a version of NTICD modified to EFSMs rather than CFGs [24].

As a summary, we can see that the definition of control dependence has been improved by researchers in several generations, because of a change in modern program structures and a need to apply control dependence from program slicing to model slicing. Nevertheless, they all aim to capture the dependence that one node is determining the execution of another node. Among a diverse range of definitions, this thesis will simply choose the NTSCD as a guideline in implementing the control dependence computation and follow an algorithm presented by Ranganath et al. [22] with some modifications to fit our own needs. Since the algorithm is used in CFGs, our model slicer will also want to convert the original model into its CFG representation.

[1]A control sink is a set of nodes that form a strongly connected component such that for each node $n$ in the control sink each successor of $n$ is also in the control sink [22]

### 2.2.4 Relevant SBM Slicing Techniques

One of the main tasks for this thesis project is to implement a model slicer. Therefore our literature survey focuses more on implementations. In an implementation document for a slicer of UML State Machine [18], a CFG is constructed to capture all the possible executions of the UML state machines. There are different types of CFG nodes, including BRANCH, SIMPLE and SEND. We observe that the conversion of CFG from model makes it easy for the slicer to directly use the notion of NTSCD as defined in [22].

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

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

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

In Section 2.2.3, we have also shown a wide range of dependencies used in many slicing algorithms. They show that one of the best ways to cope with an SBM with more complex semantics is to improve the dependence-based slicing technique by including more types of dependencies.

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

### 2.2.5 Correctness of SBM Slices

As introduced in Section 1.1.2, a useful sliced model should be correct, precise and compact. We summarize these properties based on information learnt from literature on evaluating an SBM slice.

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

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

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

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

In this thesis, the same condition is expressed as that the sliced model can simulate the original model with respect to the slicing criteria. Chapter 5 will prove this basic correctness property on the model slicer we have developed.

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

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

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

Precision property corresponds to the "strong" correctness property, with some differences: a sliced model does or does not satisfy the "strong" correctness property (a yes-or-no answer), but can be precise to a different degree. In this thesis, we will use the two state merging rules proposed by Korel et al. which satisfy the "strong" correctness property. But in all other cases, we will favour compactness over precision.

# Chapter 3

# Preliminaries

This chapter presents the preliminaries before the design of FormlSlicer. Section 3.1 introduces the semantics of FORML [11] and explains how FormlSlicer treats the feature modules in FORML. Then Section 3.2 discusses the scope of FormlSlicer. Lastly, Section 3.3 illustrates a big picture of the slicing task and how FormlSlicer extracts monitored and controlled variables.

## 3.1 What is FORML

FORML is a modelling language designed to specify feature-oriented models of SPL requirements. It is based on the paradigm of FOSD and accompanied by a taxonomy for explicitly modelling intended feature interactions [25]. A model expressed in this language consists 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 [11].

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

- FORML is a UML-like language. Its feature modules are designed based on UML state machine, which is one type of State-Based Model (SBM). This makes the design of our slicing tool much easier, because we can utilize some existing literature on SBM slicing.

- It is a good practice in software engineering research community to adopt and extend state-of-the-art analysis tools [26]. 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 [26]—a modelling language used in model checking by the NuSMV model checker [28]. It is beneficial to the community in further extending this line of works by creating a model slicer to support the compositional model verification.

FormlSlicer treats each feature module in FORML as a state machine. If a feature is an independent feature (e.g., the Basic Driving Service (BDS) feature which controls basic acceleration, deceleration and steering functionalities of a car), the feature module is treated as a complete state machine. If a feature is an enhancement or modification of another feature (e.g., the Cruise Control (CC) feature which is built on top of the BDS feature), the feature module is represented as a fragment in FORML. However, we will compose the fragmented feature module with its base feature modules together to create one complete state machine.

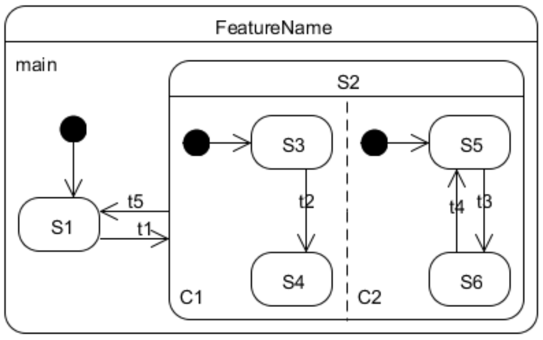


**Figure 3.1:** The Structure of FOSM in FormlSlicer

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 machine.

An FOSM is the sub-state machine contained within the *main* region of a composite state carrying the name of the feature. A typical example of an FOSM is shown in Figure 3.1. This FOSM contains two states, *S1* and *S2*. *S1* is a simple state. *S2* is a composite state which consist of two orthogonal regions, *C1* and *C2*. The state and region form a

[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 [27].

containment relation in hierarchy (each composite state contains one or more regions; each region contains a sub-state machine). We can see that such a mutual relationship between state and region is recursive. The black solid circle, called *pseudo state*, is a notation to point to the default start state. For example, *S1*, *S3* and *S5* in Figure 3.1 are default start states in their respective region.

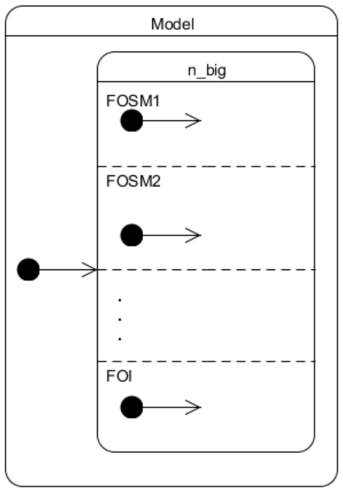


**Figure 3.2:** An Example Model

**FormlSlicer treats the whole Behaviour Model as one big state machine.** This big state machine only has a composite state containing many orthogonal regions. Each orthogonal region contains one sub-state machine, which is an FOSM. Figure 3.2 shows an example of such a big state machine in FormlSlicer.

Strictly speaking, this is not an SPL model, because:

- the FOSMs in different orthogonal regions are likely to have overlapping components;
- the FOSMs may be mutually exclusive features.

In some literature, this is called an 150% model [29]. The 150% model contains the behaviour of all features, no matter if those features exclude each other within a configuration of the SPL. The 150% model can be converted to an SPL model and vice versa.

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

> 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). Section 3.3.1 will explain how FormlSlicer extracts information from the transition.

## 3.2 Scope of FormlSlicer

The task of FormlSlicer is to produce a sliced model, in which some model elements that are irrelevant to our interest are removed.

FormlSlicer does not resolve conflicts which arise from unintended feature interactions. However, it keeps the conflicts in the sliced model; because the sliced model has a smaller size than the original model, the conflicts become much more obvious and easily detectable.

FormlSlicer does not compose feature modules in FORML. The FOSMs must be composed first before being passed to FormlSlicer as inputs.

FormlSlicer does not deal with a real SPL model. As introduced in the previous section, it takes in the 150% model. However, the 150% model can be converted to an SPL model and vice versa.

FormlSlicer focuses slicing on the feature modules in the Behaviour Model of FORML. It does not perform slicing on the World Model, which is the world domain designed based on UML class diagram. However, it is very easy to perform slicing on the World Model by using the set of relevant variables in FormlSlicer's slicing output. We can simply identify those fields that are not in the set of relevant variables and slice them away from the World Model; if all fields in a class are sliced away, we slice away the class as well. This can be done either manually, or by writing a small program to automate it.

FormlSlicer imposes certain restrictions on the input model.

The first restriction is that 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 or 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.

The second restriction is that if a state is within an orthogonal region which has another parallel orthogonal region (i.e., it is part of a concurrency construct), any transitions crossing hierarchy border from or to this state are not allowed. This kind of transition is illustrated in Figure 3.3. Because FormlSlicer has many other more important challenges to solve, we do not want to further complicate FormlSlicer by considering this special case which occur very rarely in real-life models.

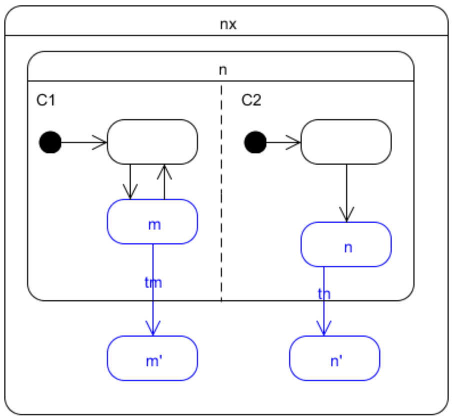


**Figure 3.3:** Concurrent Transitions crossing Hierarchy Border are not allowed in FormlSlicer

## 3.3 The Big Picture of Slicing Environment

Figure 3.4 shows the big picture of environment during one iteration of the slicing task.

In Section 3.1, we have introduced that a model consists of many FOSMs, each representing a feature in the SPL. One of them is the FOI. The FOI is the feature that we are interested in. All other features need to be sliced so that only the components that are relevant to FOI are kept in the sliced model. We say that all these other features form the ROS.

In each of all the iterations of the slicing task, a different FOSM is selected to be the FOI.

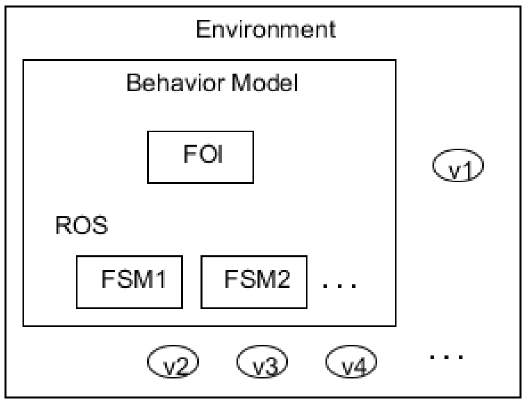


**Figure 3.4:** The Big Picture of Slicing Environment in FormlSlicer

There are a collection of variables that are influencing the behaviours of one or more features. They are divided into two groups:

- The first group is the set of environment-controlled variables, i.e., their values can only be changed by the external environment. We use $V_{env}$ to refer to 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 directly modify the actual speed; instead, they can only monitor it.
- The second group is the set of system-controlled variables, i.e., their values can be influenced by one or more FOSMs. We use $V_{sys}$ to represent them. For example, the acceleration of a car is a system-controlled variable.

*Monitored Variables* represent phenomena that are sensed by or inputs to the system; *controlled Variables* represent phenomena that are controlled or affected by system outputs [30]. Based on the definitions, we know that any $v \in V_{env}$ can only be monitored variables.

### 3.3.1 How FormlSlicer Extracts Monitored and Controlled Variables

In the previous section, we introduce the definitions on monitored variables and controlled variables of an FOSM. The same definitions apply to a transition as well.

We can extract the monitored variables of a transition from its WCE, as shown in Table 3.1. We can also extract the monitored and controlled variables of a transition from its WCA, as shown in Table 3.2.

In FORML, some message objects that are generated by a transition $t_1$ can be received by another transition $t_2$[2]; in this case, the message object $C$ appears in the WCE of $t_2$ (i.e., *C+(o)*) and in the WCA of $t_1$ (i.e., *+C(list(param = e))*). In order to match $t_1$ and $t_2$, FormlSlicer use the same String "*+C*" as a monitored variable of $t_2$ and a controlled variable of $t_1$.

| **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

In Table 3.1, the first two columns show the types and formats of WCE in a FORML transition. The types include: creation of a message object, deletion of a message object, and changing value of an attribute of a message object. The third column shows how FormlSlicer converts each of them into a monitored variable.

| **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

Table 3.2 shows how FormlSlicer deals with different types of WCA. The first two columns show the three types and formats of WCA in FORML, including generation of a message object, deletion of a message object[3], and assignment. The fourth column specifies how FormlSlicer converts each of them into a controlled variable. In the type of assignment action, there are also monitored variables in the arguments of the function, as shown in the third column.

[2]This is the construct of event generation. See Section 2.2.1 that many SBMs have this advanced construct.

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

The guard expression can be one of the three different types, as shown in the first two columns in Table 3.3. In particular, the guard condition of *inState(DummyState)* is an interesting construct. It means that this guard condition is only true when the model's current state configuration contains the state *DummyState*. The third column shows how FormlSlicer converts each of the different types of guard expression into monitored variables.

| **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

All the monitored variables in a transition $t$ form a set $mv(t)$. All the controlled variables in a transition $t$ form a set $cv(t)$[4].

**The "o" notation in a Transition** One may have observed that there is sometimes an "o" written inside the brackets 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 WCA *a1*.

Only the monitored variables of the FOI will be used as a slicing criterion to select the initial set of transitions in other FOSMs in ROS. But the controlled variables of the FOI are not. **The key principle of slicing in FormlSlicer is that it only concerns with how the other FOSMs influence the FOI, but not with how the FOI influence the other FOSMs.**

[4]See Chapter 5 for more usage on $mv(t)$ and $cv(t)$.

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

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

The next three sections will explore further into the internals of the preprocessing task and the slicing task.

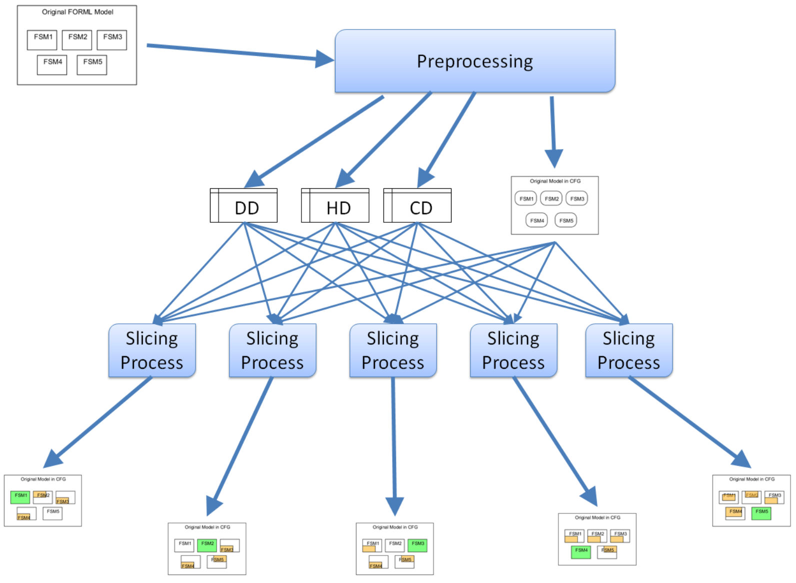


**Figure 4.1:** Overview of FormlSlicer's Workflow: the Preprocessing Task and the Slicing Task

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

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

This is a list of the sub-tasks performed in FormlSlicer:

1. Preprocessing Task
   (a) Parse and Convert the input model to CFGs
   (b) Compute Hierarchy Dependence
   (c) Compute Data Dependence

    (d) Compute Control Dependence

2. Slicing Task[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

## 4.2 Preprocessing: Model Parsing and Conversion from FORML to CFG

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

| 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

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

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

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
```

### 4.2.1 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 [31]. This concept is also useful in model slicing. As introduced in Chapter 2, some model slicers (e.g., [19]) convert a model into CFGs before slicing. FormlSlicer will do the same conversion.

The main benefit of converting an input model into CFGs is to simplify the representation of the model so that it becomes easy for further processing, particularly for the dependencies computations.

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

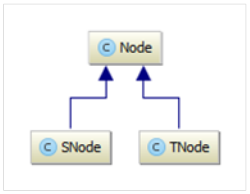


**Figure 4.2:** The Class Hierarchy of Node, TNode and SNode in CFG

**TNode**

FormlSlicer will convert each transition into a TNode. A TNode contains the set of monitored variables and controlled variables of its 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 and destination state of that transition respectively.

### SNode

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

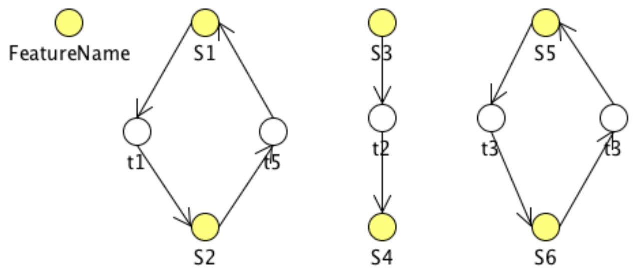


**Figure 4.3:** Several CFGs that are Converted from the FORML FOSM. Yellow Nodes are SNodes.

**After Conversion** After the conversion, each FOSM in the input model will become one or more CFGs. The FOSM example in Figure 3.1 becomes the CFGs in Figure 4.3. We can see that there is one distinctly connected CFG for each sub-state machine in this example. But if there is a transition that crosses the hierarchy border, it will be converted into a TNode that connects two distinct CFGs together.

When the CFGs for the input model are ready, it is time to move on to the next preprocessing step—dependencies computation.

## 4.3 Preprocessing: Dependencies Computation

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

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

### 4.3.1 Hierarchy Dependence

The states in a FORML model are organized in a hierarchy. A state may be a composite state which contains one or multiple regions; each region contains a sub-state machine which consists of more states, called the *child states* of the composite state. On the other hand, the composite state is called the *parent state* of the *child states*. As an example, Figure 4.4 shows such a state hierarchy; state $n1$ resides within $region1$ of state $n2$ and thus $n1$ is $n2$'s child state and $n2$ is $n1$'s parent state. The first state immediately after the pseudo state in the region is called the *default start child state*[2] of its parent state.

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

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

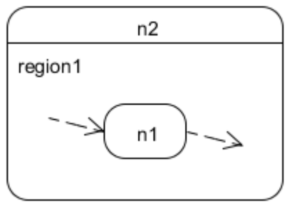


**Figure 4.4:** Hierarchy Dependence

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

**HDtable2** a 2D-table mapping each SNode to its default start child states[3]

[2]The arrow from a pseudo state to its default start child state (i.e., the state pointed by the arrow from the pseudo state) cannot be labelled. If users need a transition from the pseudo state to its default start state, users can make the original pseudo state as a new default start state and create a new pseudo state to point to it.

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

### 4.3.2 Data Dependence

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

TNode A can be data dependent on TNode B with respect to a certain variable, if TNode A 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** on $t_1$ with respect to $v$, denoted as $t_k \xrightarrow{dd}_v t_1$, iff there exists a variable $v \in mv(t_k) \cap cv(t_1)$ and there exists a path $[t_1 \cdots t_k](k \geq 1)$ such that $v \notin cv(t_j)$ for all $1 < j < k$. Here, $t$ represents a TNode in a CFG produced by FormlSlicer, $v$ represents a system-controlled variable, $mv(t)$ and $cv(t)$ are monitored and controlled variables contained in $t$ respectively[4].

The path of $[t_1 \cdots t_k](k \geq 1)$ such that $v \notin cv(t_j)$ for all $1 < j < k$ is called a **definition-clear path**.

Figure 4.5 has illustrated this concept.

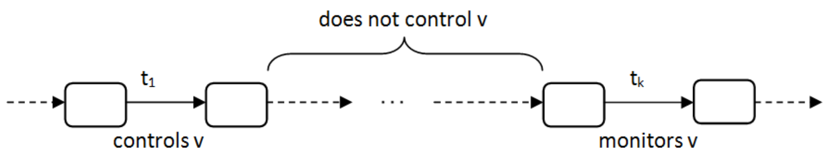


**Figure 4.5:** Data Dependence between $t_k$ and $t_1$ w.r.t. $v$

DD is the most important dependence among the three. The concepts of monitored and controlled variables are specially used by it. At the end of Chapter 3, we say that FormlSlicer only concerns with how the other FOSMs influence the FOI, but not with how the FOI influence the other FOSMs. In general, FormlSlicer is looking for relevant parts in the other FOSMs which the FOI is (directly or indirectly) data dependent on.

**InState Dependence** 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

[4]See Section 3.3 on definitions on system-controlled variables, monitored and controlled variables.

the model contains the state of *DummyState*. For consistency, FormlSlicer treats InState Dependence as a sub-type of Data Dependence. In this example, FormlSlicer considers T2 as data dependent on all the incoming transitions to *DummyState*, e.g., T1, with respect to the variable "DummyState". FormlSlicer will stores this information as a data dependence entry in the computation results of DD.

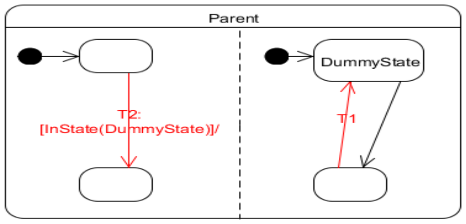


**Figure 4.6:** InState Dependence as a Variation of Data Dependence

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

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

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

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

**Algorithm 4.1:** Algorithm of Data Dependence Computation used in FormlSlicer

```
Input: allNodes, HDtable2
Output: DDtable
foreach node1 in allNodes do
    if node1 instanceof SNode OR node1.controlledVar = null then continue;
    set controlleds := node1.controlledVars // node1's controlled variables
    foreach v in controlleds do
        set visitedSNodes := {} // Empty visitedSNodes
        set qt := [the outgoing node of node1] // Initialize qt
        set dependentOn := an empty pair consisting of two Nodes
        while qt is not empty do
            set currNode := qt.poll();
            if currNode == node1 then continue;
            if currNode instanceof SNode then
                if visitedSNodes contains currNode then continue;
                if HDtable2 contains currNode then
                    ADD the all the default start child SNodes of currNode to qt
                end
                ADD all the outgoing nodes of currNode to qt
                ADD currNode to visitedSNodes
            else
                if dd contains v→(currNode,node1) then continue;
                if currNode.monitoredVars contains v then
                    ADD (currNode,node1) to dependentOn;
                end
                set isContVRset := false
                foreach c in currNode.monitoredVars do
                    if c == v then
                        set isContVReset := true;
                        break;
                    end
                end
                if isContVRset := false then  ADD the outgoing node of currNode to qt
            end
        end
        if DDtable contains Key v then  MERGE dependentOn to DDtable[v]
        else  DDtable[v] := dependentOn;
    end
end
```

### 4.3.3 Control Dependence

#### 4.3.3.1 Non-termination Sensitive Control Dependence

As introduced in Chapter 2, Control Dependence (CD) captures the notion on whether one node can decide the execution of another node. Generally, a branching node[5] is a decision point on whether the nodes along one of its branch can be passed through, and thus the branching node decides the execution of those nodes along one of its branch.

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

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

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

As elaborated in Section 2.2.3, researchers have proposed many different definitions on control dependence in both program slicing and model slicing. Until today, there is no standard, 100% correct algorithm to compute control dependence for slicing on SBMs.

FormlSlicer simply adopts one of these definitions—the NTSCD as defined in [22]. A precise definition of control dependence is presented as below:

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

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

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

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

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

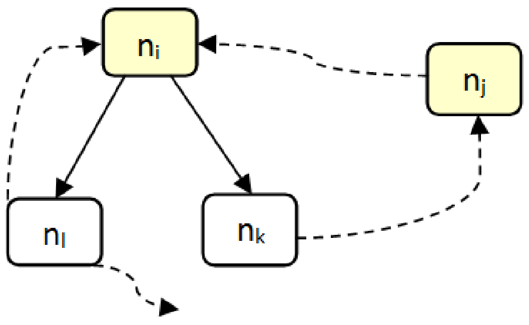


**Figure 4.7:** An Illustration of Non-termination Sensitive Control Dependence: $n_j$ is control dependent on $n_i$

#### 4.3.3.2 CD Algorithm: Paths Representation for a Node

Ranganath et al.[22] has presented a dynamic algorithm of computing NTSCD. Its main idea is to start from each branching node and search for any other nodes along the path that can be control dependent on that branching node. The algorithm represents sets of CFG paths symbolically and propagates these symbolic values to collect the effects of particular control flow choices at program points in the CFG. Based on the main idea, we adapted the algorithm to our CFGs and designed a dynamic algorithm to fit our own needs[6].

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 $i$.

[6]There are a few reasons why we do not directly use the algorithm in [22]: (1) The paper only shows the skeleton of the algorithm and does not explain certain details, such as why cardinality of sets of paths and branching node's outdegree can be used in determining control dependence; (2) the CFG used in their algorithm is slightly different from ours. However, the main idea remains the same.

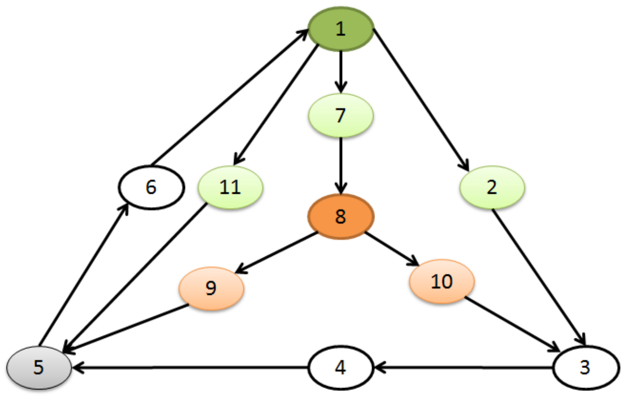


**Figure 4.8:** A CFG Example to Illustrate the Paths Representation of Node 5 from Node 1; Node 8 has Two Outgoing Nodes (9 and 10), Highlighted in Orange Colour; Node 1 has Three Outgoing Nodes (2, 7 and 10), Highlighted in Green Colour

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

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

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

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

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

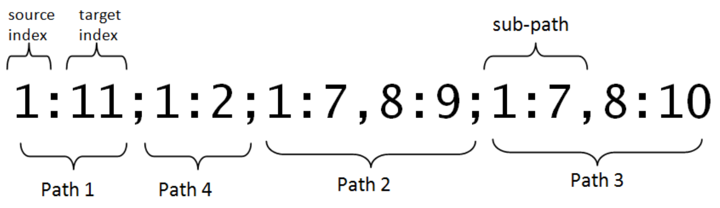


**Figure 4.9:** The Representation of Paths of Node 5 from Branching Node 1 in the CFG in Figure 4.8

Path 3 is represented as "1:7,8:10". The only difference between Path 3 and Path 2 is that at the second branching node (Node 8), Path 3 takes the branch of Node 10 from Node 8. Although Node 3 and 4 are along this path, they are neither branching nodes nor the next nodes of any branching nodes; at either of these two nodes, the path has only one possible way to go. Therefore, Node 3 and 4 do not carry useful information in distinguishing the current path from other paths, and we do not record them in the path.

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

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

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

FormlSlicer can automatically detect the fact that Node 5 is not control dependent on Node 1 by reducing the paths representation. Based on the fact that Node 8 has only two outgoing neighbours (Node 9 and Node 10), the two paths "1:7,8:9" and "1:7,8:10" can be reduced to "1:7". This implies that at Node 8, no matter we choose the branch of "8:9" or the branch of "8:10", we always reaches Node 5.

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

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

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

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

#### 4.3.3.4 CD Algorithm: Pseudo-code and Explanations

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

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

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

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

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

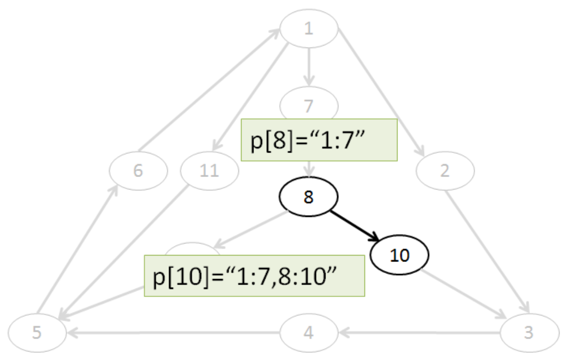
In each iteration of the *while* loop at Line 11, *currNode* is polled from the *workset* and analysed in three cases. We reuse the same CFG from Figure 4.8 to explain.

1. If *currNode* is a branching node, as detected at Line 13, then the paths representation of each outgoing node of *currNode* needs to be extended.
    - An example is shown in Figure 4.10a. The paths representation of Node 10 copies the paths representation of Node 8 and appends a subpath "8:10", to show that the path has branched at Node 8. We say that the paths representation of Node 10 is extended from that of Node 8.
    - The function *Extend*, as listed in Algorithm C.5, extends the paths representation of each outgoing node of a branching node.

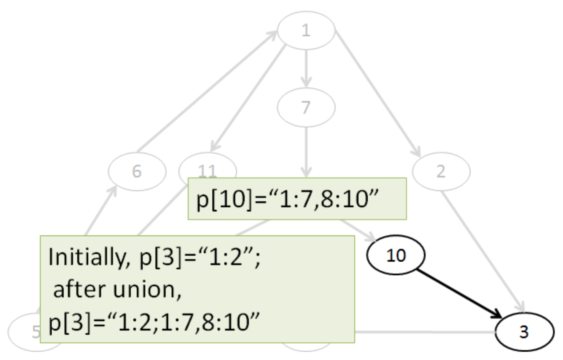
**Algorithm 4.2:** Main Algorithm of Control Dependence Computation

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

2. If *currNode* is not a branching node, as detected at Line 21, then the paths representation of *currNodes*'s only outgoing node is *union-ed* with that of *currNode*.
    - An example is shown in Figure 4.10b. The paths representation of Node 3 was initially "1:2" because of Path 4 as shown in Section 4.3.3.2. After union-ing with the paths representation of Node 10, it now becomes "1:2;1:7,8:10".
    - The function *UnionPathHappens*, as listed in Algorithm C.4, unions the paths representation of the outgoing node of a non-branching node.
3. If *currNode* does not have any outgoing nodes, nothing is performed because *currNode* is a terminating node.



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



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

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

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

The algorithm is long and thus it is modularized into several functions. There are in total six supporting functions for the main algorithm in Algorithm 4.2. Table 4.2 lists all the supporting functions' signatures and their goals[7]. In the table, p[$i$] refers to the paths representation of the node with an ID of $i$.

| Function Signature | Goal of Function | Algorithm |
|---|---|---|
| *HasReductionOccursBefore (targetIndex)* | It determines whether p[*targetIndex*] has been reduced to empty. | C.1 |
| *IsReachableFromPartialPaths (targetIndex)* | It determines whether p[*targetIndex*] contains some non-reducible paths, i.e., whether the node with an ID of *targetIndex* is control dependent on the branching node. | C.2 |
| *ReducePaths (originalPath)* | It performs reduction on the input paths representation. | C.3 |

[7]In order not to break the flow of this chapter, the supporting functions' algorithms are all listed in Appendix C.

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

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

### 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[8]

**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

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.

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

## 4.4 Multi-Stage Model Slicing Process

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

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

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

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

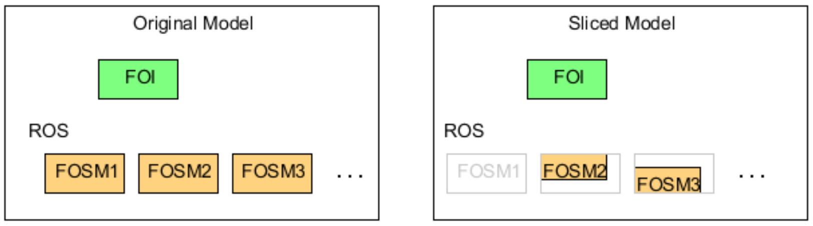


**Figure 4.11:** A Comparison between Original Model and Sliced Model

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

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

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

We will use a slicing example to illustrate each step of the Multi-Stage Model Slicing Process. The example model consists of only two FOSMs; one of them is the FOI and the other one is an FOSM in ROS.

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

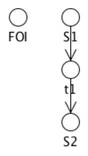
For brevity, if an element is in the sliced model, we say that it is *part-of-slice*; on the other hand, an element is *out-of-slice* if it is not in the sliced model.

### 4.4.1 Initiation Stage

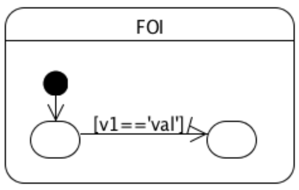
This stage selects the initial set of TNodes into the sliced model in ROS, based on what FOI monitors.

#### 4.4.1.1 Variable Extraction Step

In Section 3.3, we have discussed that the monitored variables of FOI will be used as a slicing criterion to select the initial set of transitions in other FOSMs in the ROS. This is because FormlSlicer only concerns with how the other FOSMs influence the FOI, but not with how the FOI influence the other FOSMs.



(a) The Control Flow Graph of Feature of Interest



(b) The FOSM of Feature of Interest

**Figure 4.12:** A Simple Example of Feature of Interest

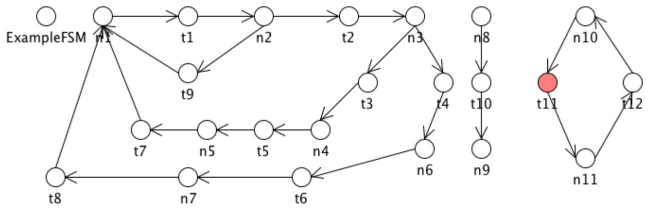
Based on this idea, *Variable Extractor* is a module in FormlSlicer to extract a collection of monitored variables from the FOI.

Figure 4.12 shows a simple example of the FOI. It monitors one variable, $v1$. *Variable Extractor* takes in the CFGs of this FOI (shown in Figure 4.12a) and extracts all the TNodes' monitored variables. In the end, it outputs a set of variables that are relevant to the FOI. We denote this set of variables as $V_{Rv}$ and call them **relevant variables**. In this example, $V_{Rv} = [v1]$.

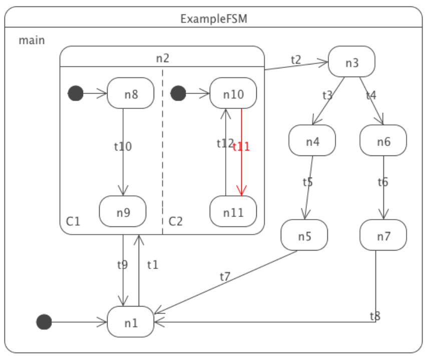
#### 4.4.1.2 Initial Transition Selection Step

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

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



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



**(b)** The Example FOSM in the ROS

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

In our example, the transition *t11* in Figure 4.13b has the label of "t11:[v2==‘val2’] /a1:v1:=‘val’;". According to how FormlSlicer extracts the monitored and controlled variables from a transition label as discussed in Section 3.3.1, the TNode of *t11* has one monitored variable $v2$ and one controlled variable $v1$.

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

### 4.4.2 General Iterative Slicing Stage

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

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

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

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

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

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

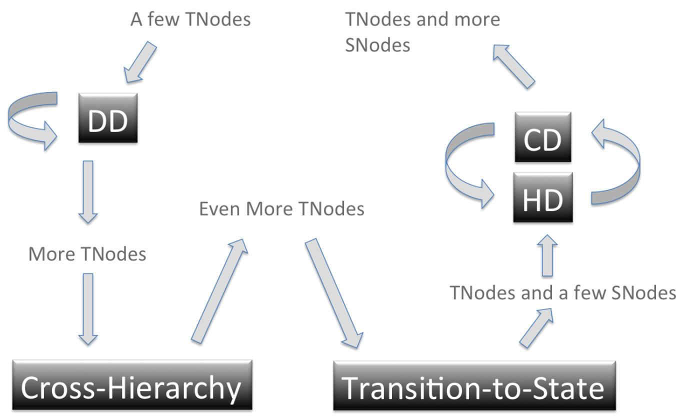


**Figure 4.14:** Four Steps in General Iterative Slicing Stage

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

#### 4.4.2.1 DD Step

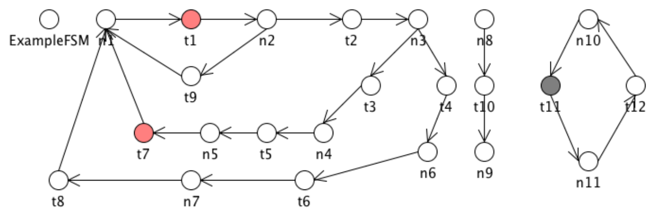
DD Step uses *DDtable* to add more TNodes that are transitively data dependent on the part-of-slice TNodes.

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

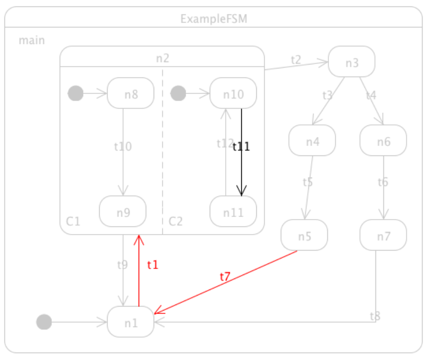
DD Step starts with $V_{Rv}$ and the partial sliced model with a few TNodes, from the previous step. Then it iterates repeatedly to (potentially) add more TNodes into the sliced model, and (potentially) enlarges the size of $V_{Rv}$. It terminates when no more change to the $V_{Rv}$ is possible.

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

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



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



**(b)** The Example FOSM in the ROS

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

**Algorithm 4.3:** DD Step in General Iterative Slicing Stage

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

We need to repeatedly add more TNodes in order to add more TNodes on which the FOI is transitively data dependent. Re-consider the example in Figure 4.13. We know that $V_{Rv} = [v1, v2]$ and $t11$ is in the sliced model, from the previous step. Now, consider the transition *t1* with a controlled variable $v2$ and a monitored variable $v3$. Then, *t11* is data dependent on *t1* with respect to $v2$; and thus *t1* is added to the sliced model as well. However, the DD Step cannot simply terminate here. Because $v3$ is monitored by *t1*, this variable is important to the sliced model. If there is another transition (e.g., *t7*) which controls $v3$, that transition affects *t1*, and indirectly affects *t11*, which again indirectly affects the FOI. Therefore, we must include $v3$ into $V_{Rv}$ and perform the node adding operation in another iteration.

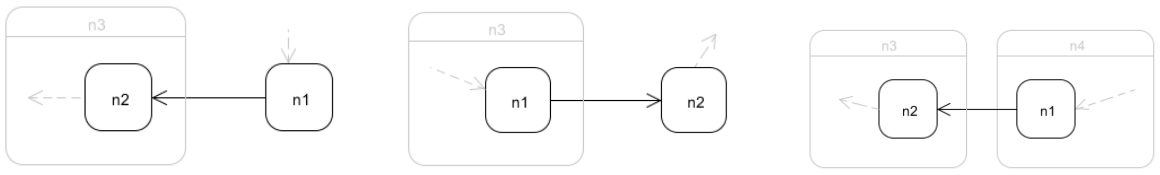
The sliced model now contains TNodes *t1* and *t7*, as shown in Figure 4.15.

#### 4.4.2.2 Replacing Cross-Hierarchy Transition

A cross-hierarchy transition is a transition which crosses the hierarchy boundary, so that its source state and destination state do not have a common parent state. We determine that due to the complexities brought by any cross-hierarchy transitions in the FOSM, these transitions need to be preserved in order for the sliced model to correctly simulate the original model.

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

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



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

**Figure 4.16:** Examples of Cross-Hierarchy Transitions

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

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

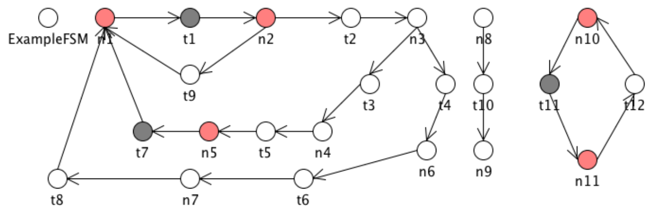
Some may ask why we cannot simply detect that important composite state $n_{cps}$ and create a path from a starting part-of-slice state to $n_{cps}$. The caveat of doing so is that it cannot accommodate many corner cases. For example, in the example of Figure 4.16a, if there is another transition $t_y$ from *n2* to another descendant state of *n3*, then $t_y$ cannot be simulated in the sliced model because there is no way to transit from *n3* to its descendant state using a transition. The example can be made more complex. We will then need a way to fix all possible corner cases; this will make the algorithm long and inelegant and yet it is still hard to guarantee that the sliced model is 100% correct.

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

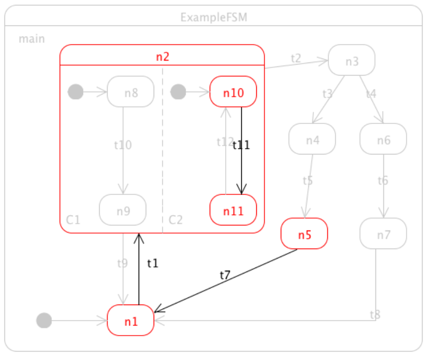
The reason of using a "true" transition is that this transition is guaranteed not to control any relevant variables. The previous step—DD Step—has already identified all the transitions which transitively control the relevant variables. Thus, any out-of-slice transitions at this step do not affect the relevant variables. We can therefore safely ignore their monitored or controlled values.

#### 4.4.2.3 Transition-to-State Step

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



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



(b) The Example FOSM in the ROS

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

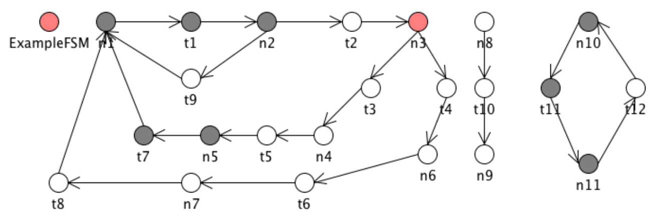
For each part-of-slice transition, this step makes its source state and destination state become part-of-slice. In other words, the SNodes that correspond to these source and destination states are added into the sliced model. Figure 4.17 shows the effect of this step.

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

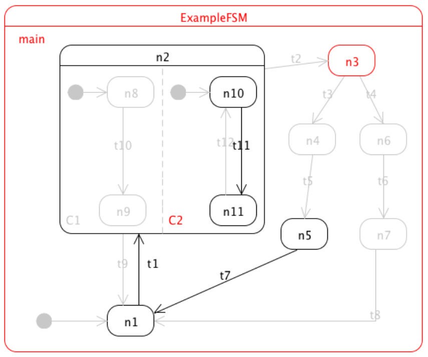
#### 4.4.2.4 CD-HD Step

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

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



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



**(b)** The Example FOSM in the ROS

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

The reason for such an repetition is that an SNode can be transitively control dependent or hierarchy dependent on another SNode. For example, a composite state can contain many descendants in different ranks of state hierarchy. Once a descendant state is selected into the slice, the repetition in CD-HD Step will ensure that all the states from the descendant state to the root state along the state hierarchy are all added into the slice.

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

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

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

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

### 4.4.3 Model Enrichment Stage

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

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

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

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

#### 4.4.3.1 Step I: State Merging

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

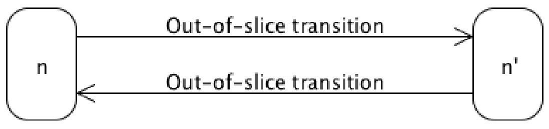
FormlSlicer uses Korel et al.'s two state merging rules that satisfy the *traversability* property[9] [5] :

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

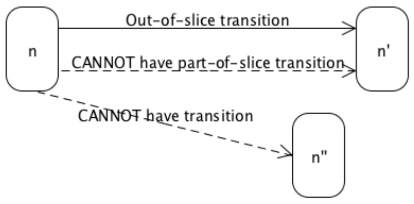
**Rule 2** States $n$ and $n'$ can be merged into one state "$n$,$n'$" if:

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

Figure 4.19 illustrates these two rules.



(a) State Merging Rule 1



(b) State Merging Rule 2

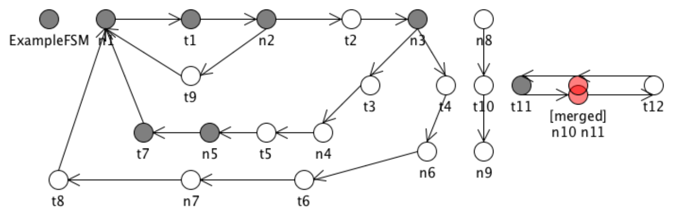
**Figure 4.19:** Illustration of State Merging Rules in Step I of Model Enrichment Stage

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

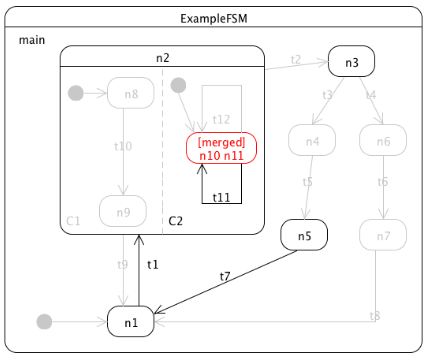
Intuitively, these two rules can be easily justified. In Rule 1, since the transitions between the two states are out-of-slice, they are not important; otherwise, they have already become part-of-slice during DD Step. In Rule 2, from $n$ the machine does not have anywhere else to move except $n'$. Therefore,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

FormlSlicer adjusts the two rules for its own use.

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



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



(b) The Example FOSM in the ROS

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

Figure 4.20 shows the same slicing example after the State Merging Step. State *n11* has only one out-of-slice transition to *n10* and therefore these two states satisfy Rule 2. They are merged together to become a new state "*[merged] n10 n11*".

#### 4.4.3.2 Step II: True Transitions Creation

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 States** We know that each sub-state machine has one default start state. Graphically, it is the state pointed by a transition without label from the pseudo state. If the default start state is out-of-slice, this sub-step 1 searches for the new default start state for the sub-state machine.

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

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

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

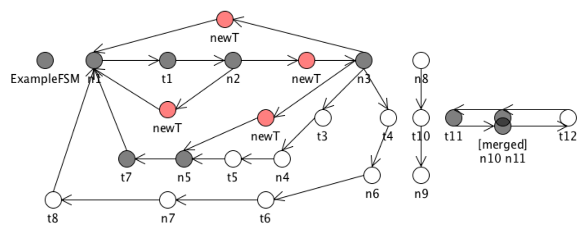
**Sub-step 2: Search for Next Part-of-slice State** Consider a path starting from a part-of-slice state $n$ to another part-of-slice state $n'$, and all the other states and transitions along this path are out-of-slice because they are not selected during all the previous steps in Multi-Stage Model Slicing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

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

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

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.


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

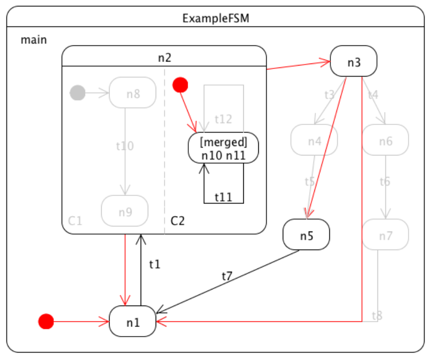

**(b)** The Example FOSM in the ROS

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

Figure 4.21 shows the slicing example after this step. We can see that *n3* is now connected to its next part-of-slice state *n1*. In fact, the path between a part-of-slice state and its next part-of-slice state can be as short as one transition. For example, *n2* has an out-of-slice path to *n3* and it consists of only one transition, *t2*; it is replaced by a "true" transition.

### 4.4.4 Summary and Postprocessing

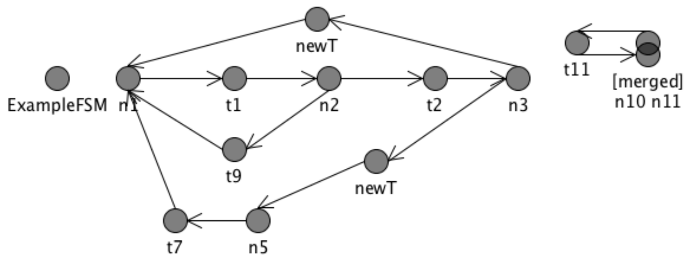
We have presented all the steps for the Multi-Stage Model Slicing Process. Figure 4.22a shows the resultant CFGs for the example feature. It looks much smaller than the original CFGs in Figure 4.13a.

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

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

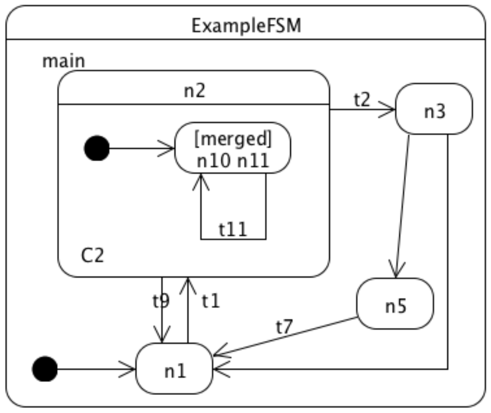
As mentioned in Section 4.1, FormlSlicer will perform the Multi-Stage Model Slicing Process for multiple times, each time with a different feature as the FOI.

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.



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

//



**(b)** The Example FOSM in the ROS

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

# Chapter 5

# Correctness of FormlSlicer

This chapter presents a correctness proof to show that the sliced model produced by the Multi-Stage Model Slicing Process, which is described in Chapter 4, can simulate the original 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 $\varphi$ 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.

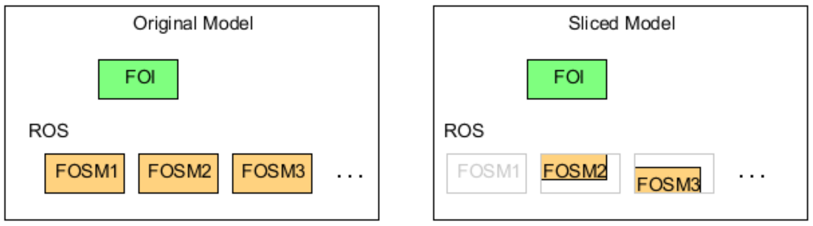


**Figure 5.1:** The Slicing Environment with Original Model and Sliced Model

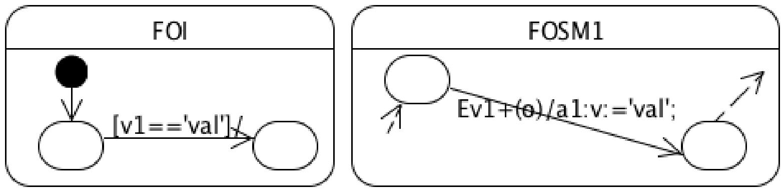


**Figure 5.2:** A Simple FOI Executing with Another FOSM in Rest of System

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 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 "environment 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 $\bullet$ in an FOSM; it is a notation to point to the default start 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 **??** 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 3.3 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 Dependence, 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 $ParentStateOfRegion(r) = ParentStateOfRegion(r')$ and $r \neq r'$.*

## 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 orthogonal 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 [11]. 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, $[F_1, \dots, F_k]$, 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 **??** 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 [11]. 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. [32])

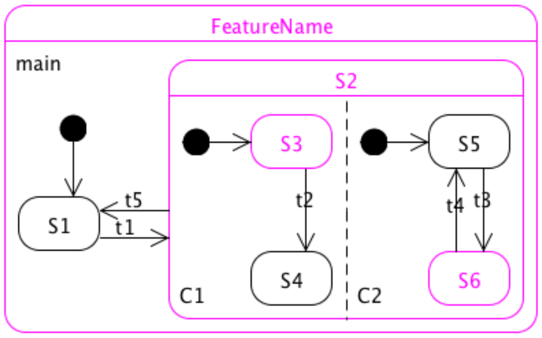


**Figure 5.3:** The FORML Example from Figure 3.1 with its Current States Highlighted in Magenta

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 $\sigma$ 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 $\sigma$ 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 [33]; 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, $\sigma$, at that particular point in time; we denote the snapshot as $(N, \sigma)$.*

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 $(N, \sigma)$ evolves to $(N', \sigma')$ 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, $(N^I, \sigma^I)$.

**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 **??** for more details.

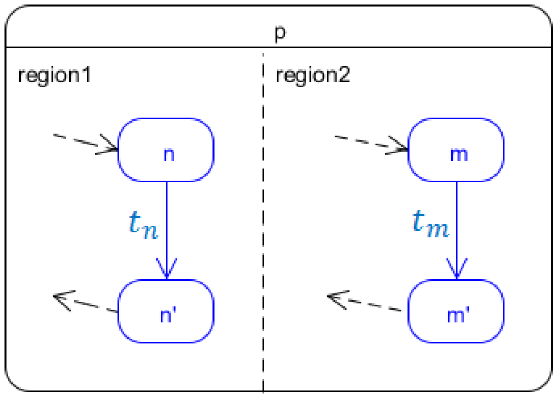


**Figure 5.4:** Concurrency in Orthogonal Regions as an Execution Step (Only Blue Coloured Components are Relevant in the Execution)

**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 $M$.

As introduced in Section 4.3, there are three types of dependencies used by FormlSlicer: Hierarchy Dependence (HD), Data Dependence (DD), and Control Dependence (CD).

**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 dependence.

## 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 $t$ [34, 11].

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

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

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

- $N \subseteq S_M$ and $N' \subseteq S_M$ are the current and next state configuration of the model;
- $exited(t)$ is the set of all states 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 [33]. 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:

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

## 5.4 FormlSlicer's Multi-Stage Model Slicing

Section 4.4 has elaborated on how the FormlSlicer performs a multi-stage 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 $t \in T_{FOI}$ such that either $v \in mv(t)$ or there exists $t'$ such that $t' \xRightarrow{dd} t$ and $v \in mv(t')$.*

In other words, a variable is a relevant variable if it directly or indirectly influences the FOI. It appears in at least one data dependence 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 original model $(N, \sigma)$ is* ***is projected to*** *another snapshot in sliced model, $(N_{\mathcal{L}}, \sigma_{\mathcal{L}})$, when*

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

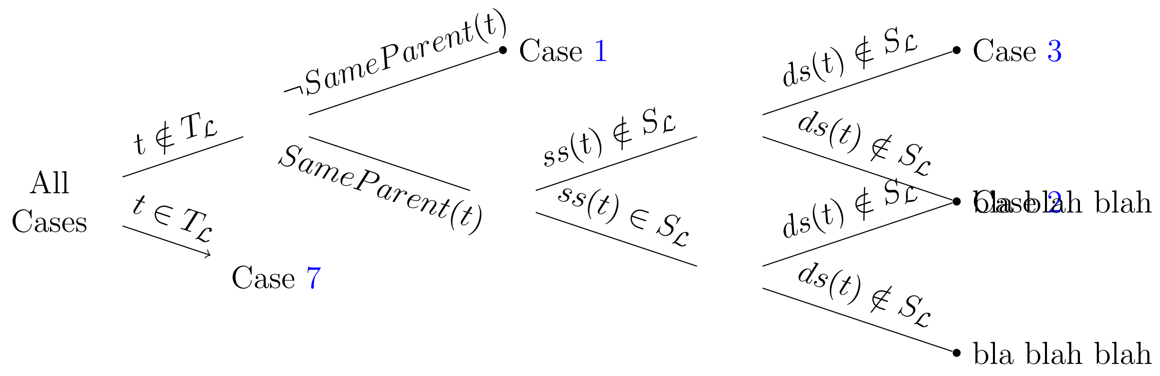
*We write it as $P((N, \sigma)) = ((N_{\mathcal{L}}, \sigma_{\mathcal{L}}))$.*

### 5.5.4 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.

This can be proved by case-based analysis. Based on whether $t$, its source state $ss(t)$ and its destination state $ds(t)$ are out-of-slice, we divide the situation into 7 different cases as shown in the decision tree below.

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


All Cases
$t \notin T_{\mathcal{L}}$
$t \in T_{\mathcal{L}}$
Case 7
$\neg SameParent(t)$
Case 1
$SameParent(t)$
$ss(t) \notin S_{\mathcal{L}}$
$ss(t) \in S_{\mathcal{L}}$
$ds(t) \notin S_{\mathcal{L}}$
Case 3
$ds(t) \notin S_{\mathcal{L}}$
Case 2
bla blah blah
$ds(t) \notin S_{\mathcal{L}}$
$ds(t) \notin S_{\mathcal{L}}$
bla blah blah

**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:*

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

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

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

[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 $t$.

*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:

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

**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 $\epsilon$ 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 $\epsilon$ 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,

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

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

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 $\epsilon$ in the sliced model.

[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.

**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 $(N^I, \sigma^I)$. $N^I$ 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

It derives the FORML models in Automotive case study from [35] and [11] and uses them as slicing targets. FormlSlicer produces satisfactory slicing results.

See Appendix A.

# Chapter 7

# Conclusion

# Appendix A

# Automotive: A Slicing Example

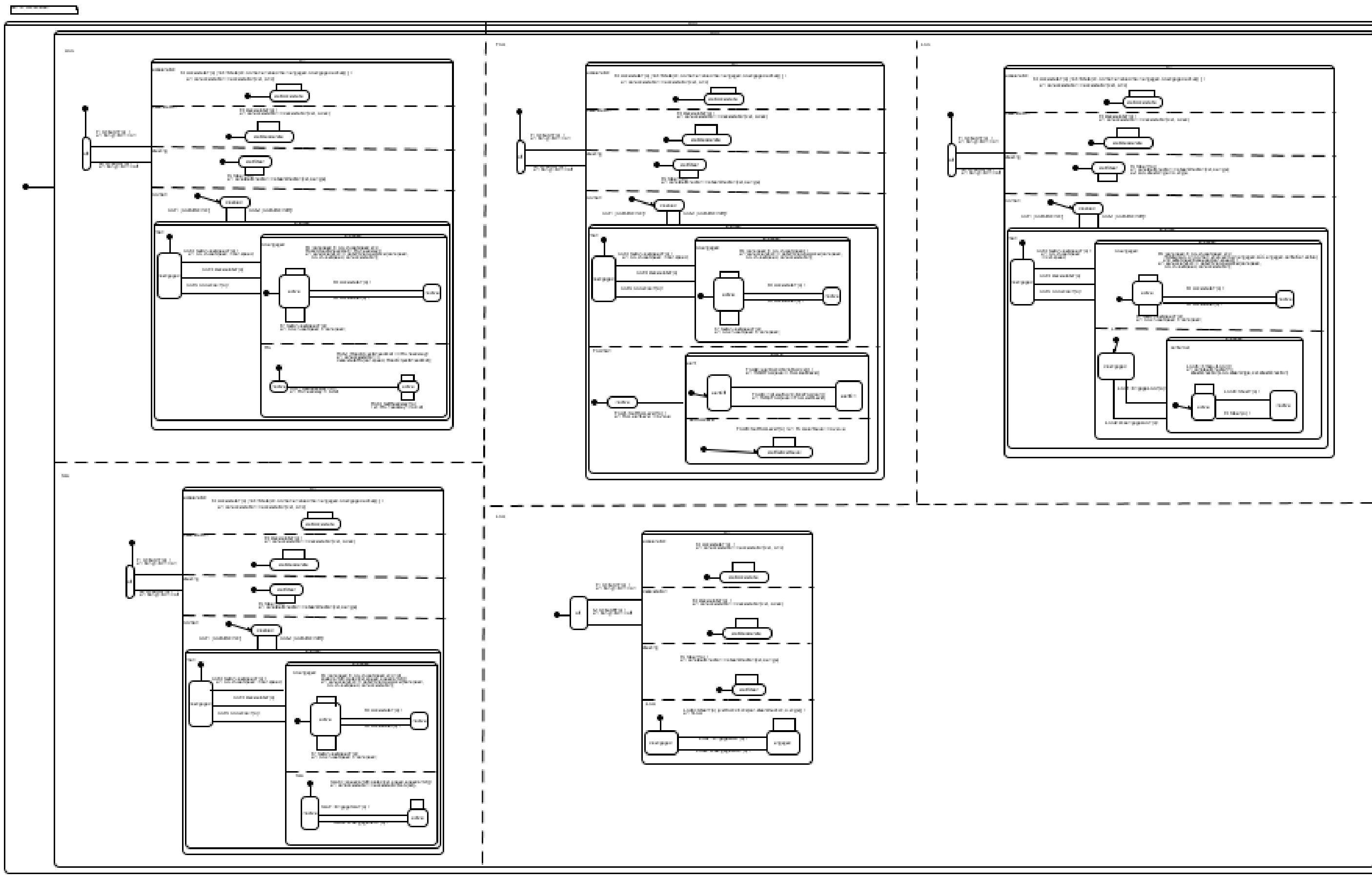


**Figure A.1:** Original Model Consisting of Five Feature-oriented State Machines

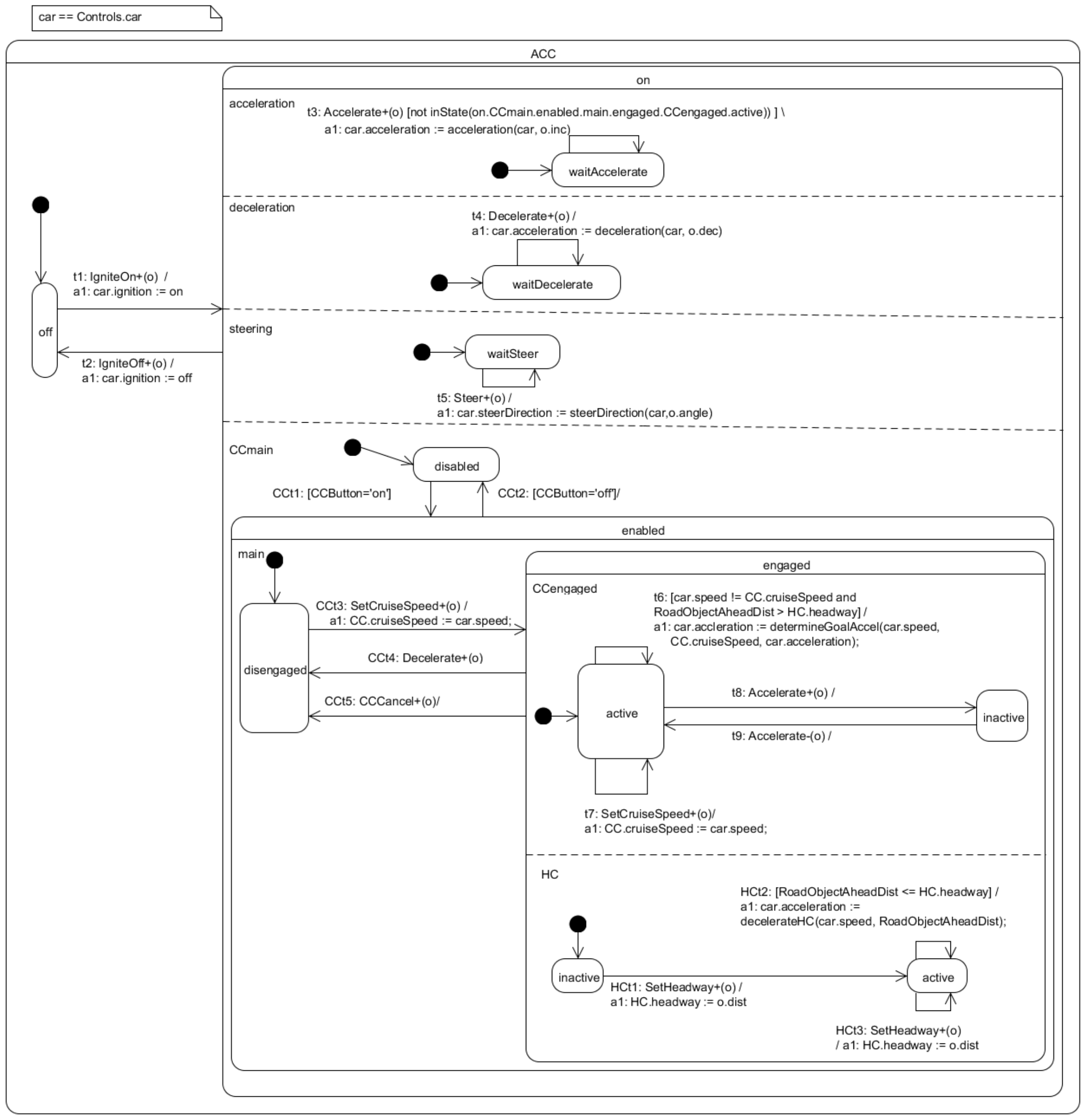

car == Controls.car
ACC
on
acceleration
t3: Accelerate+(o) [not inState(on.CCmain.enabled.main.engaged.CCengaged.active)) ] \
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
deceleration
t4: Decelerate+(o) /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
t1: IgniteOn+(o) /
a1: car.ignition := on
off
t2: IgniteOff+(o) /
a1: car.ignition := off
steering
waitSteer
t5: Steer+(o) /
a1: car.steerDirection := steerDirection(car,o.angle)
CCmain
disabled
CCt1: [CCButton='on']
CCt2: [CCButton='off']/
enabled
main
engaged
CCengaged
CCt3: SetCruiseSpeed+(o) /
a1: CC.cruiseSpeed := car.speed;
t6: [car.speed != CC.cruiseSpeed and
RoadObjectAheadDist &gt; HC.headway] /
a1: car.accleration := determineGoalAccel(car.speed,
CC.cruiseSpeed, car.acceleration);
CCt4: Decelerate+(o)
disengaged
CCt5: CCCancel+(o)/
active
t8: Accelerate+(o) /
inactive
t9: Accelerate-(o) /
t7: SetCruiseSpeed+(o)/
a1: CC.cruiseSpeed := car.speed;
HC
HCt2: [RoadObjectAheadDist &lt;= HC.headway] /
a1: car.acceleration :=
decelerateHC(car.speed, RoadObjectAheadDist);
inactive
HCt1: SetHeadway+(o) /
a1: HC.headway := o.dist
active
HCt3: SetHeadway+(o)
/ a1: HC.headway := o.dist

**Figure A.2:** Original Adaptive Cruise Control (ACC) feature

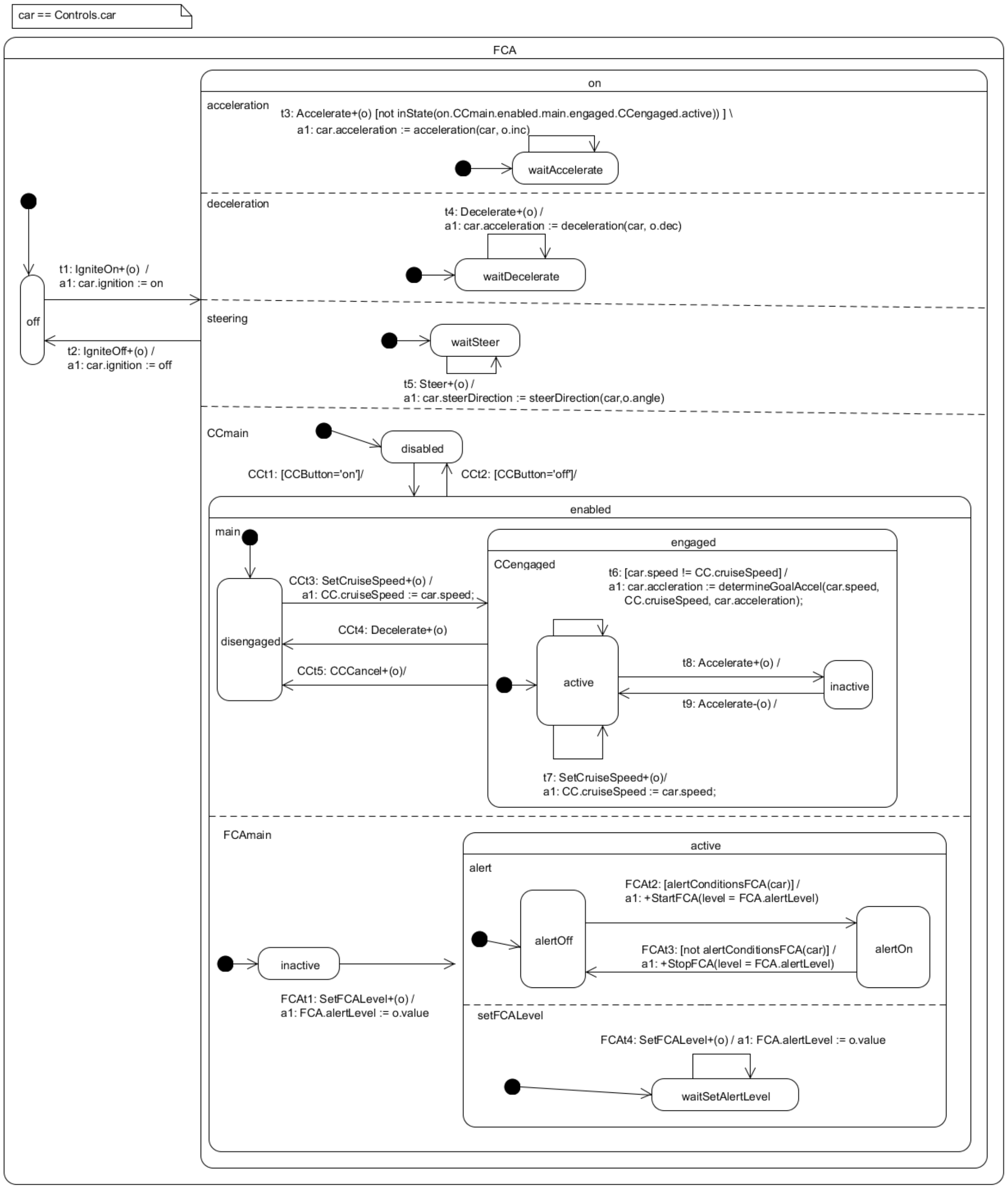

car == Controls.car
FCA
on
acceleration
t3: Accelerate+(o) [not inState(on.CCmain.enabled.main.engaged.CCengaged.active)) ] \
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
deceleration
t4: Decelerate+(o) /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
t1: IgniteOn+(o) /
a1: car.ignition := on
off
t2: IgniteOff+(o) /
a1: car.ignition := off
steering
waitSteer
t5: Steer+(o) /
a1: car.steerDirection := steerDirection(car,o.angle)
CCmain
disabled
CCt1: [CCButton='on']/
CCt2: [CCButton='off']/
enabled
main
engaged
CCengaged
t6: [car.speed != CC.cruiseSpeed] /
a1: car.accleration := determineGoalAccel(car.speed, CC.cruiseSpeed, car.acceleration);
CCt3: SetCruiseSpeed+(o) /
a1: CC.cruiseSpeed := car.speed;
CCt4: Decelerate+(o)
disengaged
CCt5: CCCancel+(o)/
active
t8: Accelerate+(o) /
inactive
t9: Accelerate-(o) /
t7: SetCruiseSpeed+(o)/
a1: CC.cruiseSpeed := car.speed;
FCAmain
active
alert
FCAt2: [alertConditionsFCA(car)] /
a1: +StartFCA(level = FCA.alertLevel)
alertOff
FCAt3: [not alertConditionsFCA(car)] /
a1: +StopFCA(level = FCA.alertLevel)
alertOn
inactive
FCAt1: SetFCALevel+(o) /
a1: FCA.alertLevel := o.value
setFCALevel
FCAt4: SetFCALevel+(o) / a1: FCA.alertLevel := o.value
waitSetAlertLevel

**Figure A.3:** Original Forward Collision Alert (FCA) feature

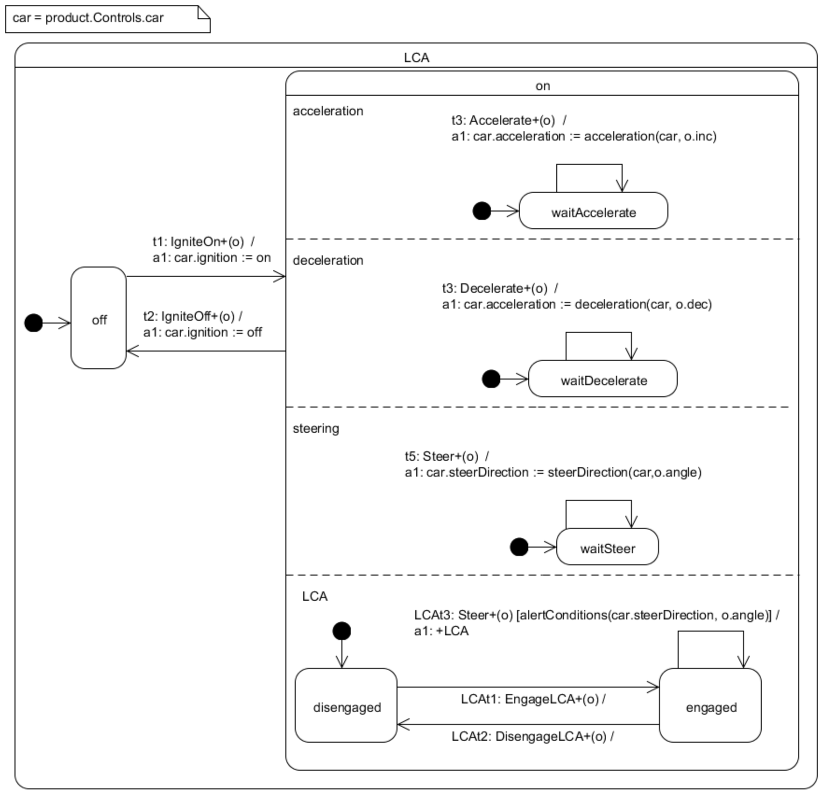

car = product.Controls.car
LCA
on
acceleration
t3: Accelerate+(o) /
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
deceleration
t3: Decelerate+(o) /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
steering
t5: Steer+(o) /
a1: car.steerDirection := steerDirection(car,o.angle)
waitSteer
LCA
LCAt3: Steer+(o) [alertConditions(car.steerDirection, o.angle)] /
a1: +LCA
disengaged
engaged
LCAt1: EngageLCA+(o) /
LCAt2: DisengageLCA+(o) /
off
t1: IgniteOn+(o) /
a1: car.ignition := on
t2: IgniteOff+(o) /
a1: car.ignition := off

**Figure A.4:** Original Lane Change Alert (LCA) feature

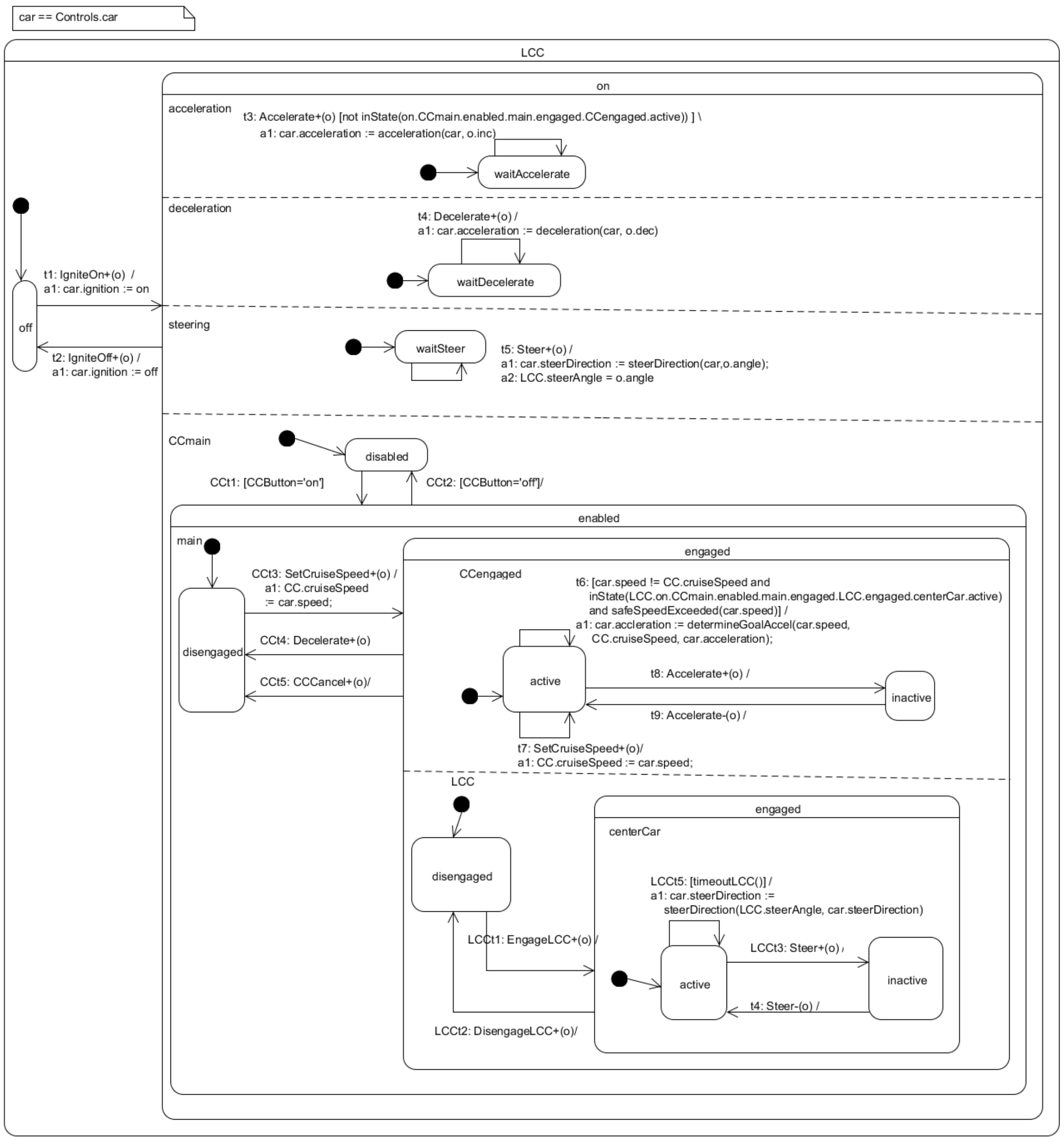

car == Controls.car
LCC
on
acceleration
t3: Accelerate+(o) [not inState(on.CCmain.enabled.main.engaged.CCengaged.active)) ] \
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
deceleration
t4: Decelerate+(o) /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
t1: IgniteOn+(o) /
a1: car.ignition := on
off
t2: IgniteOff+(o) /
a1: car.ignition := off
steering
waitSteer
t5: Steer+(o) /
a1: car.steerDirection := steerDirection(car,o.angle);
a2: LCC.steerAngle = o.angle
CCmain
disabled
CCt1: [CCButton='on']
CCt2: [CCButton='off']/
enabled
main
engaged
CCt3: SetCruiseSpeed+(o) /
a1: CC.cruiseSpeed
:= car.speed;
CCengaged
t6: [car.speed != CC.cruiseSpeed and
inState(LCC.on.CCmain.enabled.main.engaged.LCC.engaged.centerCar.active)
and safeSpeedExceeded(car.speed)] /
a1: car.accleration := determineGoalAccel(car.speed,
CC.cruiseSpeed, car.acceleration);
disengaged
CCt4: Decelerate+(o)
CCt5: CCCancel+(o)/
active
t8: Accelerate+(o) /
inactive
t9: Accelerate-(o) /
t7: SetCruiseSpeed+(o)/
a1: CC.cruiseSpeed := car.speed;
LCC
engaged
centerCar
disengaged
LCCt5: [timeoutLCC()] /
a1: car.steerDirection :=
steerDirection(LCC.steerAngle, car.steerDirection)
LCCt1: EngageLCC+(o) /
LCCt3: Steer+(o) /
active
inactive
t4: Steer-(o) /
LCCt2: DisengageLCC+(o)/

**Figure A.5:** Original Lane Centring Control (LCC) feature

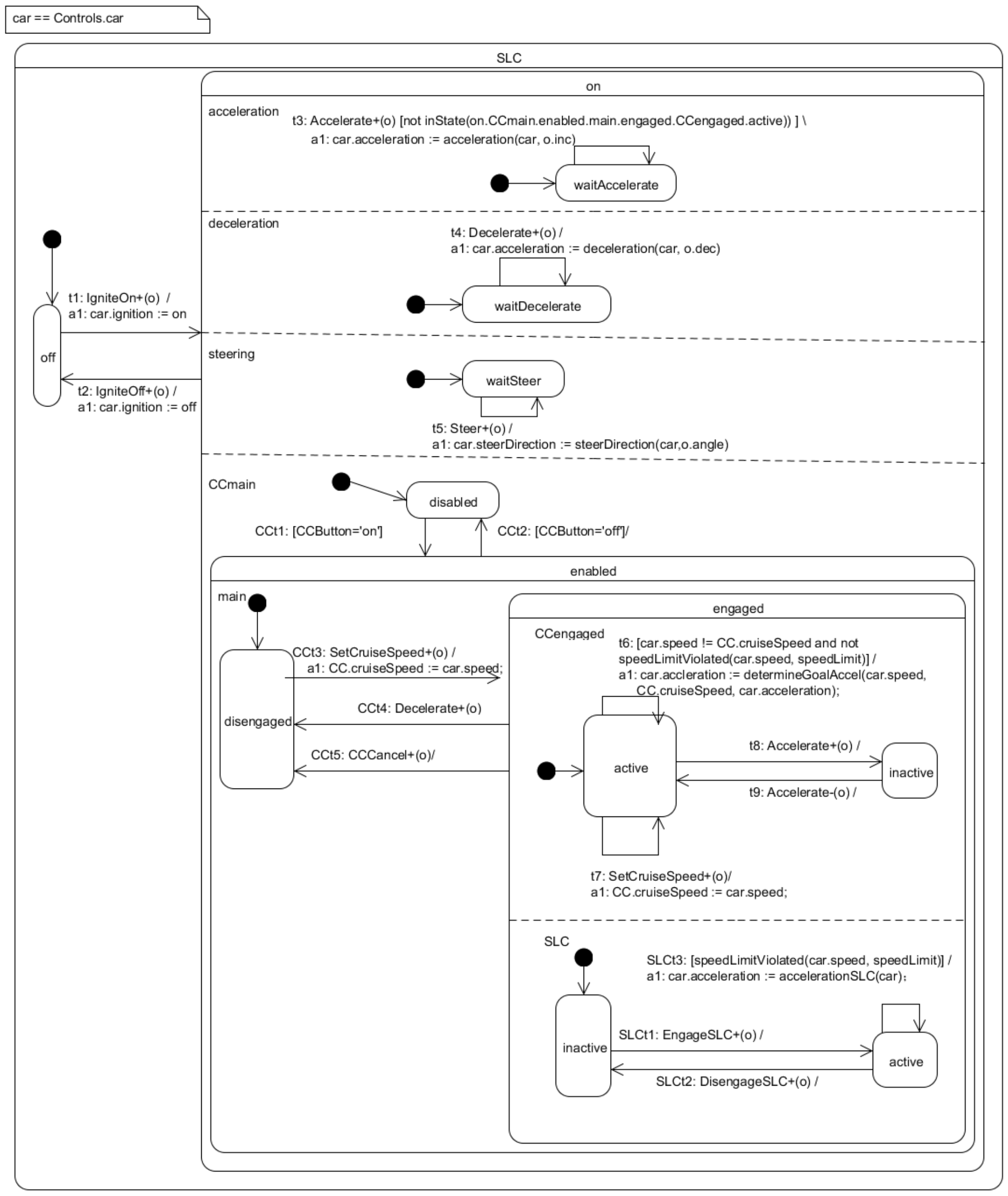

car == Controls.car
SLC
on
acceleration
t3: Accelerate+(o) [not inState(on.CCmain.enabled.main.engaged.CCengaged.active)) ] \
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
deceleration
t4: Decelerate+(o) /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
t1: IgniteOn+(o) /
a1: car.ignition := on
off
t2: IgniteOff+(o) /
a1: car.ignition := off
steering
waitSteer
t5: Steer+(o) /
a1: car.steerDirection := steerDirection(car,o.angle)
CCmain
disabled
CCt1: [CCButton='on']
CCt2: [CCButton='off']/
enabled
main
engaged
CCengaged
t6: [car.speed != CC.cruiseSpeed and not speedLimitViolated(car.speed, speedLimit)] /
a1: car.accleration := determineGoalAccel(car.speed, CC.cruiseSpeed, car.acceleration);
CCt3: SetCruiseSpeed+(o) /
a1: CC.cruiseSpeed := car.speed;
disengaged
CCt4: Decelerate+(o)
CCt5: CCCancel+(o)/
active
t8: Accelerate+(o) /
inactive
t9: Accelerate-(o) /
t7: SetCruiseSpeed+(o)/
a1: CC.cruiseSpeed := car.speed;
SLC
SLCt3: [speedLimitViolated(car.speed, speedLimit)] /
a1: car.acceleration := accelerationSLC(car);
inactive
SLCt1: EngageSLC+(o) /
active
SLCt2: DisengageSLC+(o) /

**Figure A.6:** Original Speed Limit Control (SLC) feature

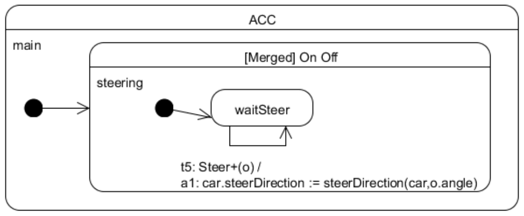


**Figure A.7:** Sliced ACC feature w.r.t. LCA

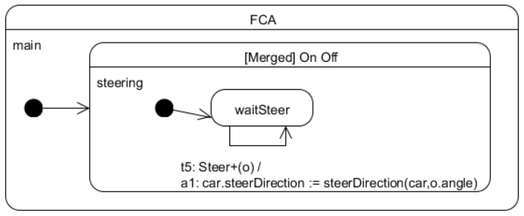
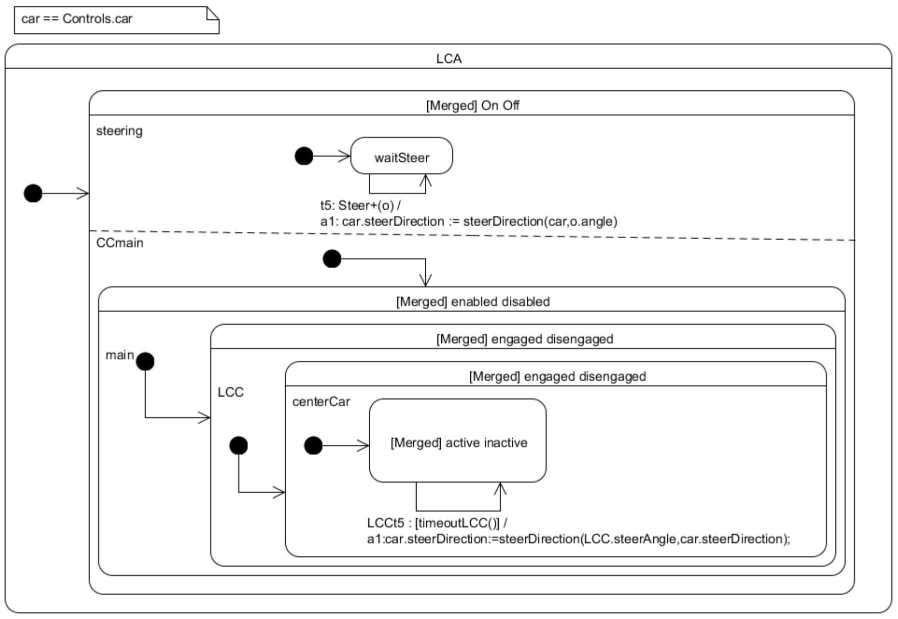


**Figure A.8:** Sliced FCA feature w.r.t. LCA



car == Controls.car
LCA
[Merged] On Off
steering
waitSteer
t5: Steer+(o) /
a1: car.steerDirection := steerDirection(car,o.angle)
CCmain
[Merged] enabled disabled
main
[Merged] engaged disengaged
LCC
[Merged] engaged disengaged
centerCar
[Merged] active inactive
LCCt5 : [timeoutLCC()] /
a1:car.steerDirection:=steerDirection(LCC.steerAngle,car.steerDirection);

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

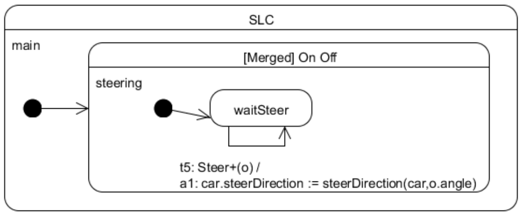


**Figure A.10:** Sliced LCC feature w.r.t. LCA

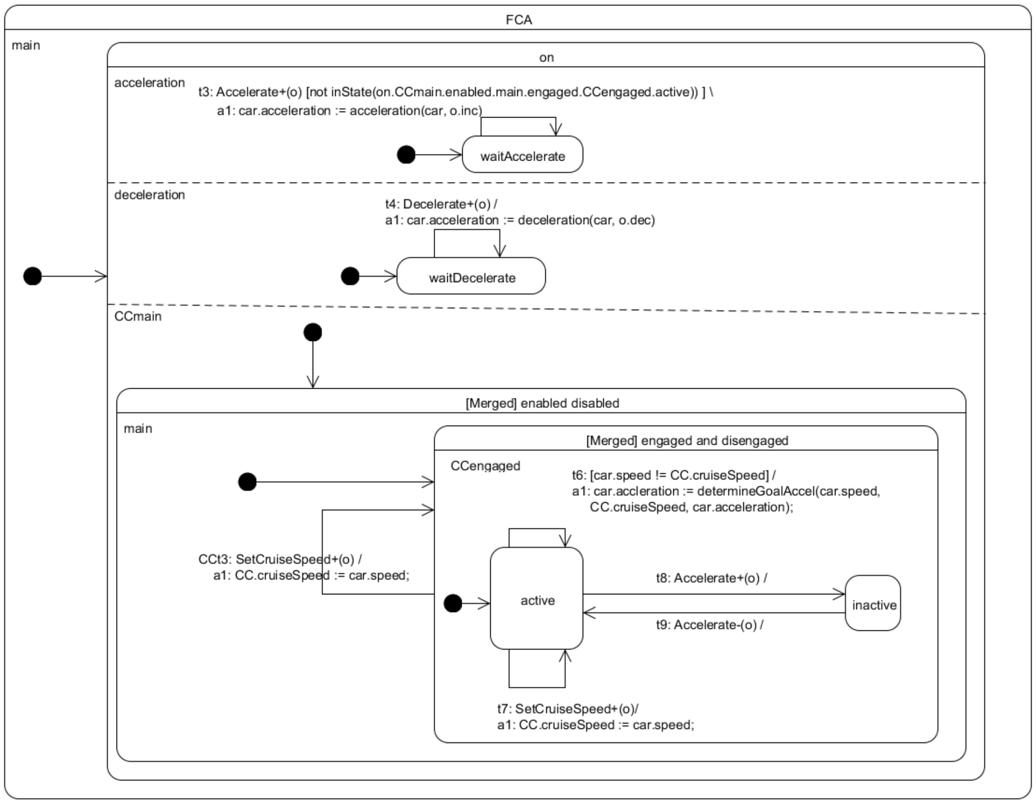

FCA
main
on
acceleration
t3: Accelerate+(o) [not inState(on.CCmain.enabled.main.engaged.CCengaged.active)) ] \
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
deceleration
t4: Decelerate+(o) /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
CCmain
[Merged] enabled disabled
main
[Merged] engaged and disengaged
CCengaged
t6: [car.speed != CC.cruiseSpeed] /
a1: car.accleration := determineGoalAccel(car.speed,
CC.cruiseSpeed, car.acceleration);
CCt3: SetCruiseSpeed+(o) /
a1: CC.cruiseSpeed := car.speed;
active
t8: Accelerate+(o) /
inactive
t9: Accelerate-(o) /
t7: SetCruiseSpeed+(o)/
a1: CC.cruiseSpeed := car.speed;

**Figure A.11:** Sliced FCA feature w.r.t. ACC

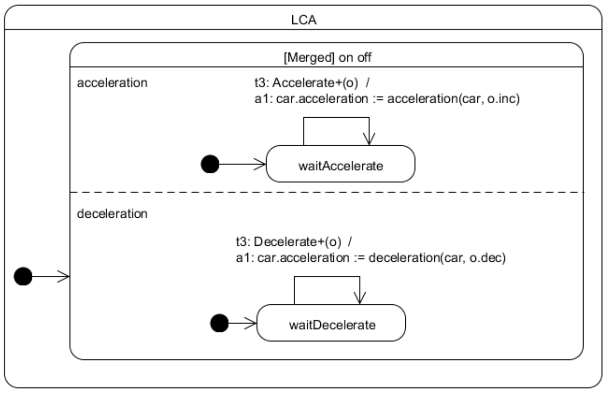
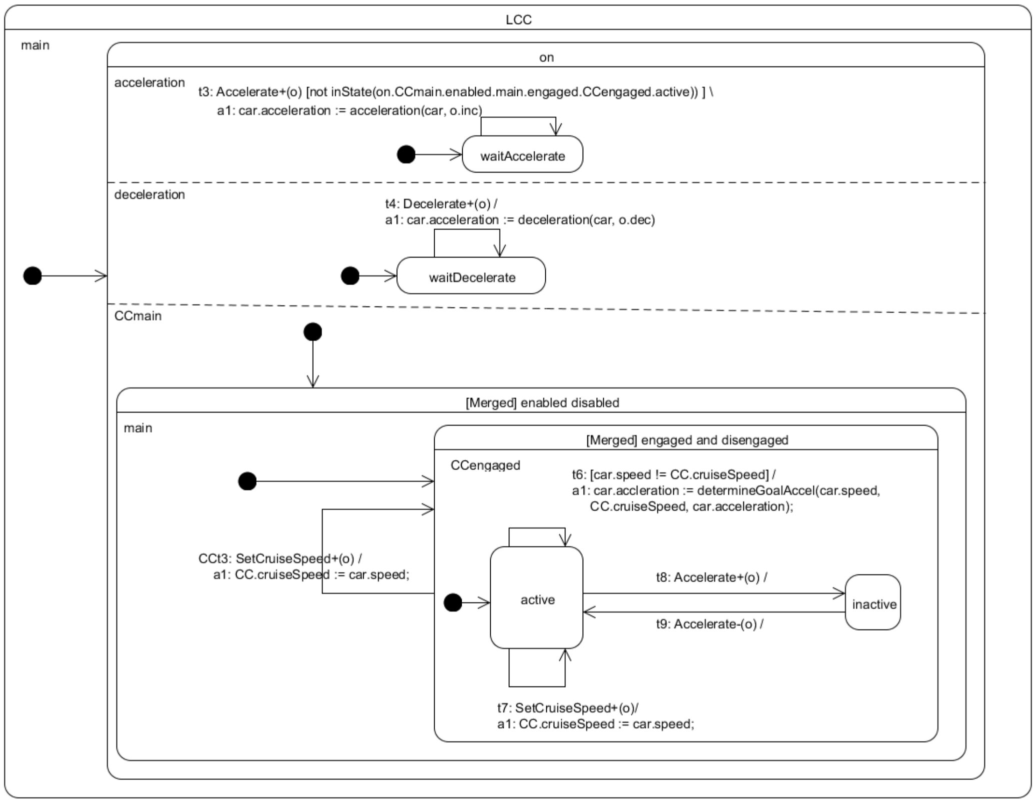


**Figure A.12:** Sliced LCA feature w.r.t. ACC



LCC
main
on
acceleration
t3: Accelerate+(o) [not inState(on.CCmain.enabled.main.engaged.CCengaged.active)) ] \
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
deceleration
t4: Decelerate+(o) /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
CCmain
[Merged] enabled disabled
main
[Merged] engaged and disengaged
CCengaged
t6: [car.speed != CC.cruiseSpeed] /
a1: car.accleration := determineGoalAccel(car.speed, CC.cruiseSpeed, car.acceleration);
CCt3: SetCruiseSpeed+(o) /
a1: CC.cruiseSpeed := car.speed;
active
t8: Accelerate+(o) /
inactive
t9: Accelerate-(o) /
t7: SetCruiseSpeed+(o)/
a1: CC.cruiseSpeed := car.speed;

**Figure A.13:** Sliced LCC feature w.r.t. ACC

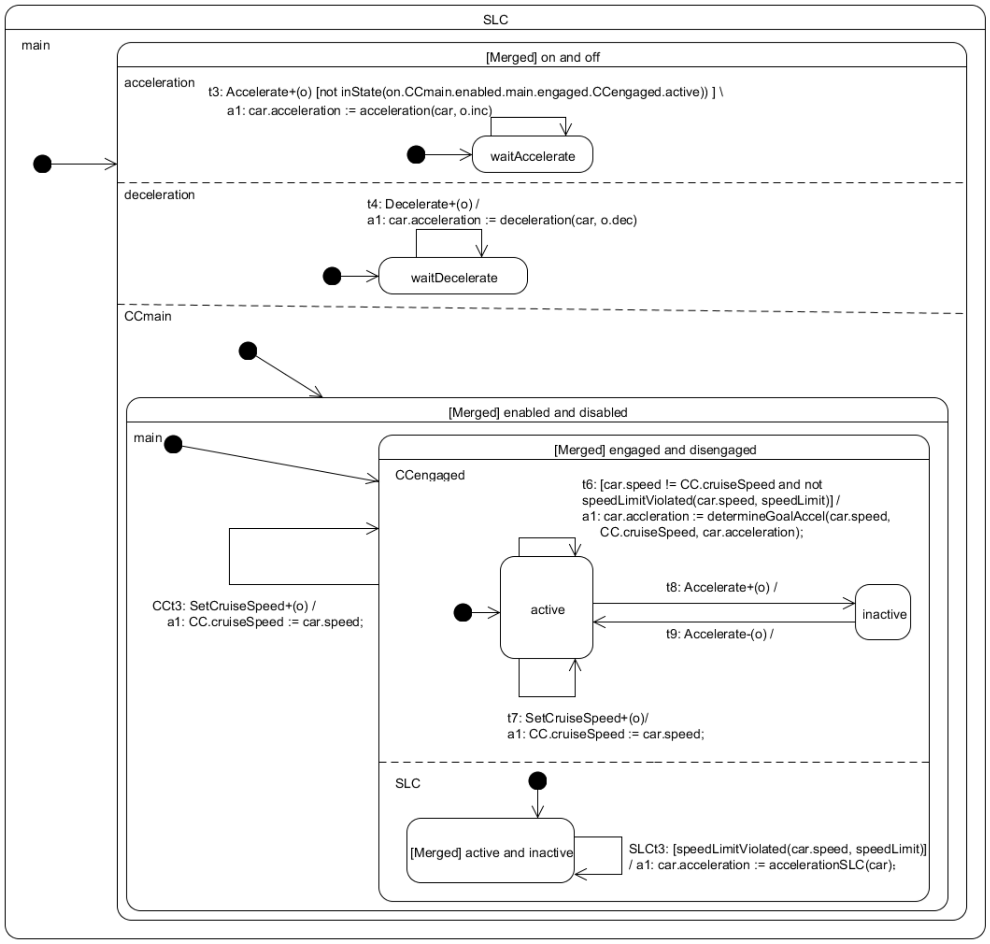

SLC
main
[Merged] on and off
acceleration
t3: Accelerate+(o) [not inState(on.CCmain.enabled.main.engaged.CCengaged.active)) ] \
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
deceleration
t4: Decelerate+(o) /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
CCmain
[Merged] enabled and disabled
main
[Merged] engaged and disengaged
CCengaged
t6: [car.speed != CC.cruiseSpeed and not speedLimitViolated(car.speed, speedLimit)] /
a1: car.accleration := determineGoalAccel(car.speed, CC.cruiseSpeed, car.acceleration);
active
t8: Accelerate+(o) /
inactive
t9: Accelerate-(o) /
t7: SetCruiseSpeed+(o)/
a1: CC.cruiseSpeed := car.speed;
CCt3: SetCruiseSpeed+(o) /
a1: CC.cruiseSpeed := car.speed;
SLC
[Merged] active and inactive
SLCt3: [speedLimitViolated(car.speed, speedLimit)]
/ a1: car.acceleration := accelerationSLC(car);

**Figure A.14:** 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 $\leq$ 1 then return paths;
    SortPaths (paths);
    set start := paths.lastIndex;
    set i : = start - 1;
    set targetIndices := $\emptyset$;
    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 := $\emptyset$;
            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] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

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

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

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

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

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

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

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

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

[10] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. A comparison of product-based, feature-based, and family-based type checking. In *Proceedings of the 12th International Conference on Generative Programming: Concepts &#38; Experiences*, GPCE '13, pages 115–124, New York, NY, USA, 2013. ACM.

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

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

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

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

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

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

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

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

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

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

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

[22] 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.

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

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

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

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

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

[28] 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.

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

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

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

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

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

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

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

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