NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

Improving State-Based Coverage Criteria Using Data Flow Information

By

Qing (Ching) Lin

A thesis submitted to the Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the

degree of

Master of Science in ISS

Information and Systems Science

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

September 2004

Copyright © 2004 by Qing Lin

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-97478-2

Our file Notre référence

ISBN: 0-612-97478-2

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

PAGINATION ERROR.

TEXT COMPLETE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

Abstract

Empirical evaluations have revealed limitations of existing state-based coverage criteria. As these criteria can be considered as executing the control flow structure of the statechart, we are attempting to investigate how data flow information can be used to improve these criteria. This thesis presents a comprehensive methodology to perform data flow analysis of UML statecharts, applies it to the transition tree criterion and uses it on two case studies. The results of the case studies show that data flow information can be used to select the best transition tree when more than one satisfies the transition tree criterion. We further propose a more optimal strategy for the transition tree criterion, in terms of cost and effectiveness. The improved tree strategy is evaluated through the two case studies and the results suggest that it is a cost-effective strategy that would fit into many practical situations.

Acknowledgements

I would like to thank my thesis supervisors, Dr. Lionel Briand and Dr. Yvan Labiche for their understanding, guidance and feedback.

Special thanks to my parents and my husband for their encouragement and support throughout my graduate study, and my son for putting up with not having mom around during the time I was working on this thesis.

Thanks also go to the SQUALL Research Group for making a friendly and enjoyable working environment.

Table of Contents

Chapter 1	Introduction.....	1
Chapter 2	Background Information.....	4
2.1	Software testing criteria	4
2.1.1	Program-based coverage criteria.....	5
2.1.2	Specification-based coverage criteria	7
2.1.3	Mutation operators	8
2.2	UML statecharts.....	8
2.2.1	States and transitions.....	9
2.2.2	Events.....	9
2.2.3	Guard conditions.....	12
2.2.4	Actions	12
2.2.5	Internal transitions	15
2.2.6	Activities.....	16
2.3	Object Constraint Language	18
Chapter 3	Related Works.....	20
3.1	Control flow coverage criteria based on UML statecharts	20
3.2	Data flow coverage criteria based on UML statecharts	24
3.3	Deriving data flow information from postconditions	26
3.4	Detailed objectives.....	27
Chapter 4	Data Flow Analysis of UML Statecharts	29
4.1	Transforming UML statecharts into event/action flow graphs	29
4.1.1	EAFG Metamodel.....	32
4.1.2	Call events and call actions.....	34
4.1.3	Signal events and send actions.....	36

4.1.4	Change events	37
4.1.5	Time events.....	37
4.1.6	Assignment actions	38
4.1.7	Create actions and destroy actions.....	39
4.1.8	Terminate actions, return actions, and uninterpreted actions	39
4.1.9	Guard conditions	40
4.1.10	Activities	40
4.1.11	Complementing EAFG metamodel with event and action types.....	41
4.2	Modified EAFGs.....	42
4.3	Determining definitions and uses in edges and nodes	48
4.3.1	Levels of precision in postconditions	49
4.3.2	Query methods	50
4.3.3	Local definitions	50
4.3.4	EAFG metamodel with data flow information	51
4.3.5	Instantiation examples	53
4.3.6	Model elements in edges.....	55
4.3.7	Model elements in nodes	57
4.3.8	Contracts with collection operations.....	64
4.3.9	Summary of rules for identifying definitions and uses.....	74
4.4	Identifying definition use pairs and definition clear paths in EAFG.....	75
4.5	Limitations and possible solutions.....	81
4.6	Data Flow Analysis of Transition Trees	85
4.6.1	Transforming transition trees into EAFGs.....	86
4.6.2	Modified transition tree EAFGs.....	87
Chapter 5	Data Flow Information Analyzer	89
5.1	Requirements for DFIA	89
5.2	System architecture.....	90
5.2.1	control subsystem	91
5.2.2	umlModel subsystem.....	92
5.2.3	EAFG subsystem	93

5.2.4	parser subsystem	93
5.3	Implementation	94
Chapter 6	Case Studies	95
6.1	Cruise Control.....	96
6.1.1	Transition trees.....	96
6.1.2	Mutation scores	99
6.1.3	Comparison of data flow information in each transition tree	100
6.1.4	Summary of results	110
6.2	VCR	111
6.2.1	Transition trees.....	113
6.2.2	Mutation scores	114
6.2.3	Comparison of data flow information in each transition tree	116
6.2.4	Summary of results	124
6.3	Conclusions from case studies	125
Chapter 7	Optimizing Transition Trees	127
7.1	Building an optimal tree.....	128
7.2	Results from case studies	130
7.2.1	Cruise Control.....	130
7.2.2	VCR	131
7.3	Conclusions.....	133
Chapter 8	Conclusions and Future Work	134
8.1	Summary and conclusions	134
8.2	Limitations and future work.....	136
References		138
Appendix A	Data Dictionary for EAFG Metamodel.....	141
A.1	Entity classes.....	141
A.2	Associations between entity classes.....	143
Appendix B	Algorithms to Derive Data Flow Information in EAFG.....	148
B.1	Algorithms to compute dominator trees	149
B.2	Algorithms to compute dominance frontiers	152

B.3	Algorithms to insert ϕ functions	153
B.4	Algorithms to rename variables and get def clear paths	154
Appendix C	Class Diagram for Cruise Control (Design Level)	157
Appendix D	Data Dictionary for Cruise Control.....	158
Appendix E	The EAFG for Cruise Control Statechart.....	161
Appendix F	Modified Transition Tree EAFGs (Cruise Control).....	162
Appendix G	Class Diagram for VCR (Analysis Level)	164
Appendix H	Data Dictionary for VCR.....	165
Appendix I	Transition Trees for VCR	176

List of Figures

Figure 1 CFG fragment for the <code>if-else</code> structure.....	6
Figure 2 Example of state transitions.....	9
Figure 3 Example of change events.....	11
Figure 4 Example of time events	11
Figure 5 Entry and exit actions	14
Figure 6 Attaching entry/exit actions to transitions.....	15
Figure 7 Internal transitions and self transitions.....	16
Figure 8 Internal transitions and self transitions with entry and exit actions attached to transitions.....	16
Figure 9 Example of activity.....	17
Figure 10 Example of infeasible transition pairs	21
Figure 11 (a) Excerpt of an EFSM (b) Corresponding flow graph.....	25
Figure 12 (a) Excerpt of a statechart (b) Corresponding EAFG.....	31
Figure 13 (a) Excerpt of a statechart with a cycle (b) Corresponding EAFG.....	31
Figure 14 (a) Excerpt of a statechart with an action sequence, entry/exit actions, and an internal transition (b) Entry and exit actions are attached to transitions (c) Corresponding EAFG	32
Figure 15 EAFG Metamodel	33
Figure 16 An abstract example	34
Figure 17 EAFG for Figure 16 (b).....	34
Figure 18 Representing call events in EAFGs.....	35
Figure 19 Representing change events in EAFGs	37
Figure 20 Representing time events in EAFGs.....	38
Figure 21 Representing assignment actions in EAFGs.....	39
Figure 22 EAFG metamodel with event/action types.....	41
Figure 23 Example of terms in a postcondition.....	44

Figure 24 Templates for connectives	46
Figure 25 Compound node for resume ()	46
Figure 26 Modified EAFG metamodel	48
Figure 27 Modified EAFG metamodel with data flow	52
Figure 28 Metamodel instantiation: an example without collection operations	54
Figure 29 Metamodel instantiation: an example with a collection operation	55
Figure 30 Du pairs vs. def clear paths	76
Figure 31 (a) Excerpt of an EAFG (b) EAFG in SSA	77
Figure 32 Example of join nodes	77
Figure 33 Example of a join node with a phi function	78
Figure 34 Pre and postcondition for event accelerate	80
Figure 35 Compound node for Running_accelerate_true	81
Figure 36 Example of decomposing the predicate part into basic nodes	82
Figure 37 Compound node for op ()	83
Figure 38 Compound node for op () after rearranging basic nodes	84
Figure 39 Example of iterative collection operations	85
Figure 40 Transition Tree 1 (Cruise Control)	86
Figure 41 EAFG for Transition Tree 1 (Cruise Control)	87
Figure 42 Modified EAFG for Transition Tree 1 (Cruise Control)	88
Figure 43 Overview of DFIA	90
Figure 44 Subsystem decomposition for DFIA	91
Figure 45 control subsystem	92
Figure 46 umlModel subsystem	93
Figure 47 parser subsystem	93
Figure 48 Statechart for CruiseController	97
Figure 49 Transition trees for Cruise Control	98
Figure 50 Mutants distributions across mutation operators (Cruise Control)	99
Figure 51 Def clear paths/du pairs vs. mutation scores (Cruise Control)	101
Figure 52 Definitions vs. mutation scores (Cruise Control)	109
Figure 53 Statechart for VcrController	113

Figure 54 Mutants distributions across mutation operators (VCR).....	115
Figure 55 Def clear paths/du pairs vs. mutation scores (VCR)	118
Figure 56 Definitions vs. mutation scores (VCR)	123
Figure 57 Building an optimal tree	129
Figure 58 Formation of a tree set.....	129
Figure 59 Optimal transition tree (Cruise Control).....	131
Figure 60 Optimal transition tree (VCR).....	132
Figure 61 Class diagram for dataFlowGenerator.....	148
Figure 62 Class diagram for Cruise Control (at design level)	157
Figure 63 EAFG for CruiseController statechart	161
Figure 64 Modified EAFG for Transition Tree 2 (Cruise Control).....	162
Figure 65 Modified EAFG for Transition Tree 3 (Cruise Control).....	163
Figure 66 Class Diagram for VCR (at analysis level)	164
Figure 67 Transition Tree 1 (VCR)	176
Figure 68 Transition Tree 2 (VCR)	177
Figure 69 Transition Tree 3 (VCR)	178
Figure 70 Transition Tree 4 (VCR)	179
Figure 71 Transition Tree 5 (VCR)	180
Figure 72 Transition Tree 6 (VCR)	181
Figure 73 Transition Tree 7 (VCR)	182
Figure 74 Transition Tree 8 (VCR)	183
Figure 75 Transition Tree 9 (VCR)	184
Figure 76 Transition Tree 10 (VCR)	185
Figure 77 Transition Tree 11 (VCR)	186
Figure 78 Transition Tree 12 (VCR)	187

List of Tables

Table 1 Summary of contributions and limitations of existing works.....	28
Table 2 Three levels of precision in postconditions, an example from [8].....	49
Table 3 Examples of obtaining a collection.....	64
Table 4 Type (1) non-iterative collection operations.....	66
Table 5 Type (2) non-iterative collection operations.....	68
Table 6 Iterative collection operations.....	70
Table 7 Summary of rules.....	75
Table 8 Def clear paths and du pairs in transition tree EAFGs (Cruise Control)	100
Table 9 Def clear paths/du pairs with respect to each attribute (Cruise Control).....	102
Table 10 Common and unique def clear paths (Cruise Control)	102
Table 11 Two def clear paths w.r.t <code>self.throttle</code> in EAFG1 and EAFG3	103
Table 12 Common and unique du pairs between EAFG1 and EAFG3 (Cruise Control).....	104
Table 13 Common and unique du pairs between EAFG1 and EAFG2 (Cruise Control).....	104
Table 14 Common and unique du pairs between EAFG3 and EAFG2 (Cruise Control).....	105
Table 15 Live mutants with transition trees (Cruise Control)	105
Table 16 Number of definitions with respect to each attribute (Cruise Control)	109
Table 17 Mutation scores for transition trees (VCR).....	115
Table 18 Live mutants with transition trees (VCR).....	116
Table 19 Def clear paths and du pairs in transition tree EAFGs (VCR).....	117
Table 20 Mutation scores and def clear paths/du pairs for each group (VCR).....	117
Table 21 Def clear paths/du pairs w.r.t. each model element (VCR)	118
Table 22 Live mutants in various classes (VCR).....	119
Table 23 Number of definitions for EAFGs (VCR)	123
Table 24 Mutation score and number of definitions for each group (VCR).....	123
Table 25 Definitions w.r.t. each variable (VCR)	124

CHAPTER 1 INTRODUCTION

In the context of object-oriented (OO) development, the Unified Modeling Language (UML) has become the de-facto standard language for analyzing and designing software systems. It includes a set of diagrams that represent different aspects of a system. UML statecharts are based on finite state machines with structures of hierarchy and concurrency, and mechanism of communication; they are commonly used to describe the state-dependent behavior of an object and other entities such as a subsystem by specifying its response to the events triggered by the system itself or the external environment. As the most formalized component of UML, statecharts have long been used as a basis for generating test data.

A number of papers have proposed coverage criteria for test data selection from UML statecharts. Some of the well established criteria include all transitions [24], all paths in transition trees [4], full predicates [24], and all transition pairs [24]. Empirical evaluations have revealed limitations of these criteria [6]. For example, the transition tree criterion sometimes results in several trees, which may significantly differ in fault detection rates. And the transition pairs criterion, though effective at detecting faults, is extremely expensive. The motivation of our research is, therefore, to propose strategies to refine these criteria and especially, the transition tree criterion that, despite limitations, appears to be a good compromise between all transitions and all transition pairs.

Researchers have pointed out, based on theoretical and empirical studies, that data flow testing strategies may be complementary to control flow testing strategies [13], [15]. As the abovementioned criteria can be considered as executing the control flow structure of the statechart, our overall approach is to derive data flow information that may occur in UML statecharts by examining event/action contracts and guard conditions, and to investigate how such information can be used to improve those criteria. Since Object Constraint Language (OCL) is the natural choice as a constraint language in the context of UML, we assume it is chosen as the formal language to express operation contracts and guard conditions.

In this research, we propose a comprehensive methodology to conduct data flow analysis of UML statecharts, which is based on examining event/action contracts and guard conditions and involves four steps: (1) Transforms a UML statechart into an event/action flow graph (EAFG) that explicitly specifies the control flow relationships among the events and actions in the statechart; (2) Modifies the EAFG according to OCL expressions; (3) Identifies definitions and uses from operation contracts and guard conditions with a set of automatable rules; (4) Derive data flow information from the modified EAFG using a set of algorithms adapted from [3].

Then we use the methodology on two case studies to empirically investigate how data flow information is related to fault detection effectiveness of transition trees. Based on the results of the case studies, we propose a strategy to build an optimal tree that improves fault detection results.

The rest of the thesis is organized as follows: Chapter 2 presents some background information. Chapter 3 discusses the related work. Chapter 4 presents our approach of data flow analysis of UML statecharts. Chapter 5 describes the prototype tool that aids the data flow analysis. Chapter 6 reports on two performed case studies. Chapter 7 describes the strategy to generate optimal trees. Chapter 8 presents conclusions and points out directions for future work.

CHAPTER 2 BACKGROUND INFORMATION

This Chapter presents some material that is needed to fully understand this thesis. Section 2.1 reviews software testing criteria. Section 2.2 introduces UML statecharts and its elements. This is followed by a brief introduction to OCL given in Section 2.3.

2.1 Software testing criteria

Software testing is the process of revealing defects in software systems. It is a critical element in software development and directly affects the quality of a software system. An important problem in software testing is to decide whether a piece of software has been tested enough. An *adequacy criterion* is a coverage measure to indicate the degree of thoroughness with which the software is tested and to guide any additional testing that might be required. Most adequacy criteria are based on graphs and can be broadly classified as *structural graph coverage criteria* and *data flow coverage criteria*. The former set of criteria exercises the control flow whereas the latter is based on the flow of data through the software artifact represented by the graph.

Graph coverage criteria can be either based on program implementations or specifications. The primary concern of this research is with a set of statechart-based coverage criteria, which is a subset of specification-based coverage criteria. Program-based coverage criteria, however, are most studied. Section 2.1.1 reviews program-based coverage criteria, followed by a brief discussion of specification-based coverage criteria in Section 2.1.2. A detailed discussion of various state-based coverage criteria, which is

the subject of this research, is presented in Chapter 3. As this research measures the effectiveness of a test set in terms of mutants being killed, Section 2.1.3 reviews the fundamental concepts of mutation operators.

2.1.1 Program-based coverage criteria

Program-based coverage criteria use the control flow or data flow properties of a program implementation to determine whether a test suite satisfies some criterion. They are defined on a *control flow graph* (CFG). In a control flow graph, nodes are basic blocks of program statements, where a *basic block* is defined as a maximal set of ordered statements such that if the first statement of the block is executed, the other statements are executed sequentially. Directed edges in a control flow graph represent the flow of control: an edge from node *i* to node *j* indicates that node *i* can be executed immediately after node *j*. The example in Figure 1 shows the CFG fragment for an *if* statement.

Structural graph coverage criteria for source code are those criteria that are defined in terms of the control flow properties of CFG: the all nodes (also called all statements) criterion requires that each node in the CFG be executed by some test case, the all edges (also called all branches) criterion requires that each edge in the CFG be traversed, and the all paths criterion requires that all program paths be traversed by some test case.

```

if ( x < y )
{
  y = 0;
  x = x + 1;
}
else
{
  x = y;
}

```

Figure 1 CFG fragment for the `if-else` structure

Data flow coverage criteria for source code focus on the flows of data values. This is done by tracking the definitions and uses of variables. A definition of a variable is the occurrence of the variable in the program where the value of the variable is stored into memory. For instance, a variable appearing on the left hand side of an assignment statement indicates a definition. A formal parameter of a procedure is considered defined at the beginning of the procedure. A *use* of a variable is the occurrence of the variable where its value is accessed. A use can be either a *computational use* (*c-use*) or *predicate use* (*p-use*). A *c-use* occurs when the variable appears on the right hand side of an assignment statement or as an argument for a procedure (the parameter being an *in* parameter); a *p-use* occurs when the variable appears in the predicate of a decision statement. A *definition use pair* (*a du pair*) is defined as a triplet (v, d, u) which consists of a variable v , a node d in the program's CFG where v is defined, and a node or edge u where v is used, and for which there is a path p from d to u such that v is not redefined on p [31]. Such a path p is called a *def clear path* with respect to (w.r.t.) variable v .

Based on the above concepts, data flow coverage criteria can be defined [26] : A test set satisfies the all-definitions criterion if for every variable v , the test set contains def clear paths from every node where v is defined to a use of v . A test set satisfies the all-uses criterion if for every variable v , the test set contains at least one def clear path from every node where v is defined to every reachable use of v . A test set satisfies the all-du-paths criterion if for every variable v , the test set contains all def clear paths from every node where v is defined to every reachable use of v , and these paths are either cycle free or single loops traversals.

2.1.2 Specification-based coverage criteria

Specification-based coverage criteria determine the adequacy of a test suite by checking whether the specification has been tested thoroughly. Specification-based testing has several advantages [24]. For instance, it is implementation-independent and thus is applicable to any implementation and testing activity can be shifted to an early stage of software development process. Finite state machines (FSMs) and UML statecharts, provide a precise definition of the expected behavior of a class, class cluster, or subsystems, and are therefore widely used as a specification to define test criteria. FSMs are essentially graphs, whose nodes represent states and edges represent transitions among states. UML statecharts are based on FSMs with structures of hierarchy and concurrency. Various control flow and data flow coverage criteria based on UML statecharts are discussed in Chapter 3.

2.1.3 Mutation operators

Mutation operators are commonly used to evaluate the fault detection effectiveness of different testing techniques [6]. Faulty versions of the Software Under Test (SUT) are created by systematically seeding faults into the SUT. Each program where a fault is seeded is referred to as a *mutant*. To avoid interaction effects between faults, each mutant program possesses one and only one fault. *Mutation operators* are used to automatically generate mutants in a systematic way. Mutation operators are predefined program modification rules, based on a fault model for a given programming language [16], [17], [19]. A test suite is executed over a mutant program, and the output is compared against the original program. If the output is different, the mutant is said to be *killed*, otherwise the mutant remains *live*. A mutant remains live either because it is functionally equivalent to the original program or the test suite is inadequate to kill the mutants. *Equivalent mutants* should not be counted when calculating the mutation score of a test suite; *mutation score* is defined as the number of mutants killed by a given test set divided by the total number of mutants. Mutation score is a typical measure of the effectiveness of a test suite.

2.2 UML statecharts

Our purpose in this research, as described in Chapter 1, is to analyze the data flow information in UML statecharts, and to investigate how such information may help improve existing state-based testing criteria. In the current Chapter, we closely look at elements of UML statecharts defined in [29], and will refer to them in Chapter 4 when we present a detailed data flow analysis.

2.2.1 States and transitions

A finite state machine consists of a finite number of states and transitions. A *state* represents a situation or a condition where the object satisfies some condition, performs some activity, or waits for some event [5]. The state of an object is defined by the values of the object's attributes and links to other objects [1]. A *transition* is a directed relationship between two states, and it indicates how an object changes state due to an event. A transition, in general, may have five parts: a source state, an event trigger, a guard condition, an action, and a target state. The *source state* is the state from which the transition departs, the *event trigger* specifies the event that fires the transition, the *guard condition* is a condition that must be true for the transition to be fired, the *action* is the object's reaction to the event, the *target state* is the state that results from the firing of the transition. A special type of action is the activity, which is an ongoing task that an object executes while it stays in a specific state. In Figure 2, the transition from source state A to target state B indicates that when the object which owns the state machine is in A, it will enter B and perform the action a if the event trigger e occurs and the guard condition g is satisfied.

Figure 2 Example of state transitions

2.2.2 Events

An *event* is defined as a noteworthy occurrence that takes place at some point in time [29]. A triggering event of a state transition may have parameters that can be used in the

guard condition and the actions on the transition. UML [6] classifies four kinds of events, namely, call events, signal events, change events, and time events.

A call event represents the receipt of a request to invoke an operation. It is usually modeled as an operation on the class of the receiver, with its name being the name of the operation. There are two special cases of call events, object creation event indicating that the instance receiving the event has just been created, and object destruction event indicating that the instance receiving the event is being destroyed.

A signal event represents the receipt of an asynchronous signal. The sender creates and initializes a signal instance and sends it to one or more objects. A signal is modeled in UML as a class. The attributes of a signal class are the parameters of the signal event. Moreover, signals are generalizable. A child signal inherits the attributes of its ancestors, and can trigger a transition that requires one of its ancestors.

Ultimately a signal event is realized by an operation, usually called the event handler. Different solutions exist to specify the handler of a signal event. One of the solutions is that an operation, with the same name as the signal, be declared in the class or interface that accepts the signal, thus explicitly indicating the handler (note that such an operation has the stereotype <<signal>>) [27]. Another solution, suggested in [5], is to consider that the handler of the signal event is the action out of the transition.

Call events and signal events have the same syntax: `event-name (parameterlist)`.

A change event triggers a transition when the value of a Boolean expression becomes true. It is modeled by the key word `when` followed by a Boolean expression: `when (a`

Boolean expression). Figure 3 shows an example of change events where `currentTemp` is an attribute of the state owning object [25].

Figure 3 Example of change events

A time event models the expiration of a specific deadline. Like change events, time events are not declared as named events. Time events are either absolute time or the elapsed period of time since entering the source state. The former can be notated with the keyword `when` and the latter `after`, followed by an expression involving time. Figure 4 shows an example of time events.

Figure 4 Example of time events

Time and change events also have handlers. In those cases, the only solution to indicate the handler is to consider that the action out of the transition is the handler.

A special kind of transitions, called completion transitions is defined in [30]. A completion transition is a transition that may have a guard condition and an action but no explicit trigger. The implicit trigger for a completion transition is the completion event

that is typically generated by an activity of the source state of the transition. Activity is further discussed in Section 2.2.6.

To sum up the above discussion, we may describe the trigger events of transitions using one of the following notations: `callEvent [guard]`, `signal [guard]`, `when (...)` or `after (...)`.

2.2.3 Guard conditions

A guard condition is a Boolean expression associated with a trigger event. It may reference attributes of the state owning object and its links with other objects, as well as the parameters of the trigger event. The guard condition is evaluated when the event occurs: if the expression evaluates to true, the transition fires, otherwise the transition doesn't fire. The same event can trigger more than one transition from a single state, with each transition having a different guard condition [27].

Note that a guard condition is different from a change event in that a guard condition is evaluated only once at the time the trigger event occurs whereas a change event is evaluated continuously until it becomes true.

2.2.4 Actions

An action on a transition is an atomic computation that is executed by the owning object as a response to the trigger event of the transition. The response can also be an action sequence, which is a list of actions that are executed one after another. The entire sequence is also atomic and must be executed completely before other events can be handled [27]. Actions in a transition action sequence are usually assumed to be

independent of one another, and as a result their order of execution does not matter. In cases when the order is relevant, it is assumed that modelers specify transition actions in the right sequence. Our research conforms to the assumption here and analyze transition actions according to their written order. An action expression may reference the attributes and associations of the object that owns the state machine, as well as the parameters of the trigger event of the transition. Eight types of actions can be used in UML statecharts: assignment actions, call actions, create actions, destroy actions, send actions, return actions, terminate actions, and uninterpreted actions.

A call action invokes an operation on an object or a set of objects. A call action is synchronous. A call action takes the form: `object-set.action-name(parameterlist)`, where the object-set is the target to which the message is being sent. Note that an object may invoke its own operations in which case `object-set` can be omitted.

A send action creates and initializes a signal instance, and sends it to an object or a set of objects. Each receiver gets its own copy of the signal. A send action has the same form as a call action.

An assignment action assigns a given value to an object's attribute. It is notated as `anObject.anAttribute:= OCL expression`.

A create action instantiates an object, specified as `new class-name(argumentlist)`. A destroy action destroys an object together with its links and composite parts. It is usually written as `object.destroy()`.

A return action is an action that returns a value to a caller. A terminate action is an action in which an instance causes itself to cease to exist. It is denoted by the keyword `terminate`. Uninterpreted actions are those actions that are not defined by UML, and have to be specified by the implementation language.

Actions can also be associated with states. These actions are referred to as entry and exit actions. An entry action is dispatched whenever the state is entered, and an exit action is dispatched whenever the state is left. On a transition which has its own trigger event and actions, and the source and target state with exit and entry actions, the execution order is: the triggering event, the exit action of the source state, the action attached to the transition, entry action of the target state, as illustrated in Figure 5:

Figure 5 Entry and exit actions

Entry and exit actions can be moved out of the state and attached to transitions without changing the semantics of a statechart. Entry actions can be added after the actions of each of the incoming transitions and exit actions can be added before the actions of each of the outgoing transitions, as shown in the example in Figure 6. Note that we assume that the statecharts in our research have already gone through such a conversion process. As a result, all actions are associated with transitions only.

Figure 6 Attaching entry/exit actions to transitions

2.2.5 Internal transitions

A state may also have a list of internal transitions. An internal transition executes without exiting or re-entering the state in which it is defined. In other words, an internal transition doesn't lead to change of states, and no entry or exit actions are dispatched during an internal transition. Note that a self transition (also called a reflexive transition) of a state is different from its internal transitions in that when a self transition fires, the source state is exited and reentered, that is, both exit actions and entry actions are dispatched. As an example, the state in Figure 7 (a) has an internal transition, $e2/a4$. Assume the event $e2$ is received while in state. The execution order is $e1, a1, a2, e2, a4, e3, a3, a5$. In Figure 7 (b), the execution order of the self transition $e1/a3$ is $e1, a2, a3, a1$.

Figure 7 Internal transitions and self transitions

Since we assume entry and exit actions are moved out of the state and attached to transitions (Section 2.2.4), the above two examples can be represented as in Figure 8.

Figure 8 Internal transitions and self transitions with entry and exit actions attached to transitions

2.2.6 Activities

In addition to actions, it is also possible to have an activity executed as a result of a state transition. An activity is a computation that executes for the duration of a state. Unlike an action, an activity is not atomic. An activity is enabled on entry into the state and disabled on exit from the state [14]. The cause of the state change may be due to some input event,

in which case the activity is interrupted, or the activity itself may generate the event that causes the state change. In this latter case, the event is a completion event. (Section 2.2.2).

An ongoing activity in a state is specified with the `do` transition [5]. A `do` transition can specify one or more activities (executed in sequence) that can be interrupted at any time. A `do` transition can also consist in a sequence of actions: actions are never interruptible, but sequences of actions are (when an action in the sequence terminates, the sequence can be interrupted). Note that an activity is not terminated by the firing of an internal transition [27].

Figure 9 shows an example where the change of state is triggered by some input event unrelated to the activity. The `sound alarm` activity is on as long as the system is in the `Sounding` state. When the `reset` event occurs, the transition fires and the system is taken back to the `Monitoring` state and the activity `sound alarm` is disabled [27].

Figure 9 Example of activity

2.3 Object Constraint Language

A UML diagram typically doesn't provide enough information for a precise and unambiguous specification. OCL has been designed as a formal language to specify constraints which supplement UML diagrams. Compared to other formal languages such as Z, OCL doesn't require as much mathematical background, and is more straightforward to read and write [28]. OCL can be used for a number of different purposes [30]. In our research we assume OCL is used to specify guard conditions and operation contracts.

Combining OCL operation contracts with UML models is advised by current and emerging software paradigms [20]. The principle behind using contracts to specify operations is referred to as the design by contract principle [22]. It views the relationship between a class and its clients as a formal agreement in terms of the rights and obligations of each party. The services offered by the class are operations. For each operation, the obligations of the client, which are also the rights of the class, are specified by preconditions. A precondition is defined as being true before the operation is executed. The obligations of the class, which are also the rights of the client are specified by postconditions. A postcondition is defined as being true after the operation has been executed.

OCL is a declarative language that has no side effects [32]. This means that the evaluation of OCL expressions doesn't change the state of a system. OCL is also a typed language. Each type has a set of predefined operations that can be applied on it. In general, OCL types can be divided into 3 categories: basic types, collection types and

user defined model types. The basic types include Integer, Real, String, and Boolean, supplemented by meta types OclType, OclAny, and OclExpression. The collection types include abstract type Collection and its concrete subtypes Set, Bag and Sequence. The user defined model types include the classes, interfaces, components and datatypes in a UML model.

In a postcondition, an OCL expression can refer to two sets of values for each model element, that is, the value of a model element at the start of the operation, referred to as *pre-value* and the value of a model element upon completion of the operation, referred to as *after-value* [29]. The value of a model element in a postcondition is the after-value of the model element. To refer to the pre-value of a model element, the *@pre* postfix has to be used after its name. Note that operations on collections are denoted in OCL expressions using the arrow operation. The pre-value of a property of a collection is denoted by appending the postfix *@pre* to the collection name, e.g., `collection->size= collection@pre->size+1`, meaning the after-value of the size of collection equals the pre-value of the size of collection plus one.

CHAPTER 3 RELATED WORKS

This Chapter looks at the state of the art in the fields related to our research. Section 3.1 and Section 3.2 review coverage criteria based on UML statecharts that focus on control and data flow information, respectively. Section 3.3 presents the approach taken by [33] to derive data flow information from postconditions. Section 3.4 states more precisely the objective of this research in light of existing works.

3.1 Control flow coverage criteria based on UML statecharts

Four well known coverage criteria for test data selection from state-based specifications have been defined and we adapt the definitions provided in [24]: The transition coverage criterion requires that all transitions in the statechart be tested. The full predicate coverage criterion requires that the test set include tests that cause each term of guard conditions to control the value of the guard condition. The transition pair coverage criterion requires that each pair of adjacent transitions be tested. The complete sequence criterion requires that all transition sequences that form a complete practical use of the system be tested.

Note that a transition pair on a statechart may turn out to be infeasible. The attribute values of states, pre- and postconditions of events and actions, and guard conditions may conflict with each other. To illustrate this, consider the following example: Figure 10 describes part of the statechart of a class, say class X , which has an attribute v . Suppose, the state invariant of state $s1$ is $v > 1$, where v is an attribute of the class, the state

invariant of state s_2 is $v < 0$, the guard condition g_3 is $[v < 0]$. The feasibility of transition pair (e_1, e_3) depends on the value of v at s_1 , the post-conditions of e_1 and/or a_1 , and the guard condition g_3 . If, for example, e_1 changes the value of v while a_1 doesn't, and the postcondition of e_1 is $v > 1$, then the pair (e_1, e_3) will be infeasible. Note that this also applies to the complete sequence criterion.

Figure 10 Example of infeasible transition pairs

The generation of test paths for three of the abovementioned criteria, namely, transitions, full predicates, and transition pairs coverage can be automated. As for the complete sequence criterion, the authors [24] argue that since the number of possible transition sequences may be infinite in many realistic applications, the selection of meaningful sequences of transitions depends on the experience and judgment of test engineers, and hence cannot be fully automated. The W-method, originally proposed in [11] is however a solution intended to automate the testing of transition sequences in a systematic and cost-effective way. The W-method was focused on finite state machines with no hierarchy or guard conditions on transitions. Binder adapted it to UML statecharts [4]. This method traverses the statechart to build a transition tree according to the following procedure: the initial state is the root of the tree, an edge is drawn for each transition out of the initial node, and the resultant state of this transition is represented by a node. If the state this node represents is already drawn somewhere in the tree, or is a final state, this

node is considered terminal and no more transitions are drawn from it. The set of paths (paths which start with the tree root and end with a tree leaf of the transition tree) are called round-trip paths by the author as they capture all the transition sequences which begin and end at the same state and simple paths from the initial state to the final state with no loops. To account for the guard conditions in UML statecharts, Binder proposes that when the guard is a simple predicate or only contains AND operators, then one tree branch is drawn; if the guard contains one or more OR operators, then each true combination of the guard needs a separate branch. Since the statechart can be traversed either depth first or breadth first, different tree structures can be generated, but are all considered equivalent in a sense that they all cover the round-trip paths. Each tree path represents a test case starting from the root and ending at a leaf node.

It's worth mentioning that the state-based testing criteria discussed so far are based on flat statecharts without hierarchical or concurrent structures, though several papers such as [9] generate test cases directly from hierarchical statecharts. Steps for flattening statecharts are provided in [15]. Note that flattening a statechart may need OCL constraints to be transformed as well. For example, the OCL constraints that refer to composite states in a hierarchical statechart need to be transformed so that the composite states are replaced by their substates. Our research assumes that the statecharts have already been flattened and OCL constraints are transformed accordingly. It may be argued that this "flattening" process leads to a highly complex statechart and fails to provide a direct mapping to the original design model. Such a process, however, can be fully automated, and the resultant flattened statechart can be regarded as an intermediate

form, being used by the test case generation algorithms only rather than being visualized by modelers or testers.

As one of the most established state-based testing strategies, the W-method has been widely used in protocol testing. Several empirical studies [2], [6], however, have revealed the limitations of the strategy. Antoniol et al. applied the round-trip strategy to a typical state-dependent container class [2]. They found certain kinds of faults can't be detected by the strategy and proposed that black-box testing such as the *category-partition* techniques can play a supplementary role. As they generalized their findings, many statecharts contain methods whose behavior depends on parameters and data values, to ensure a higher fault-detection ratio, the round-trip strategy needs, in many cases, to be augmented with black-box testing such as category-partition; the role of control or data flow testing criteria to complement the round-trip strategy, however, remains to be further investigated.

Another experimental evaluation study used simulation experiments to compare the cost-effectiveness of criteria: round-trip transition trees, all transitions, all transition pairs, and full predicates criteria [6]. The round-trip strategy is observed to yield mixed fault detection ratio. Traversing the statechart according to Binder's algorithm results in several different trees in one case study, all of which are equivalent in the sense that they all cover the round-trip paths in the statechart. These "equivalent" trees, however, are observed to have quite different fault-detection effectiveness. The all transitions criterion is observed to be not sufficient to achieve an adequate level of fault detection as the adequate test sets which satisfy the criterion tend to miss a significant number of faults. The transition-pairs criterion offers a very strong guarantee to detect nearly all faults, but

the cost is extremely high. The full predicate criterion, when applied to an example that contains complex guard conditions, is more expensive than transition pairs, but not quite as effective. Though these results are preliminary and need to be confirmed by additional experiments, the conclusion of the experiment was that the round-trip path strategy appears to be a good compromise between all transition and all transition pairs, provided that there exists a mechanism to select effective transition trees when alternative trees are possible.

3.2 Data flow coverage criteria based on UML statecharts

All the criteria we discussed in Section 3.1 are defined in terms of states and transitions. Hong et al. [15] argue that since these criteria are based on the control flow of the finite state machines or statecharts, they should be complemented with data flow analysis. Kim et al. [18] generate test cases based on data flow information in a UML statechart. Recall that a number of data flow criteria were defined in [26], [13] for program-based testing which examine the associations between the definition of a variable and its uses. To apply these criteria to statechart-based testing, Kim et al. perform two transformations on the statechart:

Step 1: First the statechart is transformed into an Extended Finite State Machine (EFSM) which contains all the possible transition sequences of the statechart, but with no hierarchy, concurrency or communication mechanism.

Step 2: Then the EFSM is transformed into a flow graph, where the nodes can be either s-nodes or t-nodes. An s-node represents a state in the EFSM, and a t-node represents a transition. There exists an edge from an s-node to a t-node if the state that the s-node

represents is the source state of the transition represented by the t-node. And there exists an edge from a t-node to an s-node if the state that the s-node represents is the target state of the transition represented by the t-node. One example is given in Figure 11, where A and B are s-nodes, and the transition $e[g]/a$ is a t-node.

Figure 11 (a) Excerpt of an EFSM (b) Corresponding flow graph

It is worth noting that the data flow criteria proposed by [26], [13] are based on implementation code. When used with specifications expressed in UML statecharts, modifications have to be made to reflect the semantic differences between statecharts and programming languages. Kim et al. [18] consider a variable as being defined when the variable is assigned a value in an assignment action, and a variable as being used when it is referenced in an assignment action or in a guard condition. This approach, however, is not complete as definitions and uses caused by events are ignored (e.g. a call event may assign a value to a variable, as described in the postcondition of the call event), and only assignment actions are handled whereas UML statecharts may have 8 different kinds of actions (Section 2.2).

To capture a complete set of variable definitions and uses in UML statecharts, one can examine the pre- and postconditions of events and actions as well as guard conditions. The work done by Zweben et al. [33], with the focus of testing ADT module specifications, was one of the earliest attempts to examine the function postconditions to determine variable definitions and uses. The specification that the author worked on, however, was meant to illustrate the general approach of model-based specification, rather than being specific to UML statecharts. This is discussed in detail in the next section.

3.3 Deriving data flow information from postconditions

Zweben et al. [33] focused on applying control flow and data flow testing criteria to ADT modules which are specified in terms of operation pre- and postconditions. Edwards [12] extended it to object-oriented software components. The approach involves generating a *specification flow graph* to depict the operations of a component and the control flow relationships between them. Each node in the graph represents one operation, a directed edge from node i to node j indicates that the operation in node j may be invoked following the operation in node i .

Zweben et al. [33] determined whether there is a definition or a use of a variable by examining the postcondition of an operation, based on the assumption that a postcondition describes whether a variable is changed or not during the operation. A set of rules to identify definitions and uses are defined in [33], and definitions and uses from the postconditions of each node are derived accordingly. Then program-based criteria such as all definitions, all uses, and all du paths were applied to the ADT module.

Note that this approach has two limitations. First, precondition of an operation is not taken into account. Second, as mentioned at the end of Section 3.2, this approach is not based on a state model.

3.4 Detailed objectives

As discussed in Section 3.1, empirical studies have revealed the limitations of some of the state-based criteria [6]. For example, the transition pairs criterion is very effective but extremely expensive. The transition tree criterion may sometimes result in several trees, which may significantly differ in terms of fault-detection effectiveness. The objective of this research is to propose strategies to refine and improve these state-based criteria. Recall that a number of researchers have pointed out, based on theoretical and/or empirical studies, that data flow criteria may be complementary to control flow criteria. As most of the well established state-based testing criteria are defined in terms of states and transitions of the statechart under test and can be therefore considered as control flow oriented (Section 3.1), we are attempting to investigate how data flow information can be used to improve these criteria in the specific context of UML statecharts.

In the case of the transition tree criterion, when more than one transition tree can be obtained by traversing the statechart with Binder's algorithm, we intend to explore the relationship between the mutation score of a tree and its data flow information, i.e. we want to investigate whether the tree which exercises the most data flow is the most effective in terms of fault detection. If this is the case, we can conclude that data flow information can be used to select a tree when alternative trees exist.

Three different pieces of work are relevant to our objectives, as discussed in Section 3.2 and Section 3.3. Their contributions and limitations are summarized in Table 1. Our intent is to extend and combine them for the purpose of improving existing criteria based on UML statecharts, especially the transition tree criterion.

Existing works	Statechart notation	Guard	Event	Action	Operation contract
[18]	Yes	Yes	No	Only assignment	No
[33], [13]	No	N/A	N/A	N/A	Only postcondition

Table 1 Summary of contributions and limitations of existing works

CHAPTER 4 DATA FLOW ANALYSIS OF UML STATECHARTS

Our data flow analysis of UML statechart consists of four steps. The first step is to transform a UML statechart into an Event/Action Flow Graph (Section 4.1). The purpose of such a transformation is to explicitly represent the control flow relationship among the events and actions of UML statecharts. Next, the Event/Action Flow Graph (EAFG) is modified to account for alternatives in OCL expressions (Section 4.2). Once the pre and postconditions of an event/action are modeled by the flow graph, definitions and uses from these constraints can be identified with the rules defined in Section 4.3. Then, definition-use pairs and definition clear paths are derived from the EAFG by using one of the well-established algorithms proposed for compiler optimization [3] (Section 4.4). Section 4.5 summarizes the limitations of our approach and discusses possible solutions. Section 4.6 applies the approach to the transition tree criterion.

4.1 Transforming UML statecharts into event/action flow graphs

Our analysis of data flow information in statecharts is based on an event/action flow graph. An *event/action flow graph (EAFG)* is defined as a directed graph G where nodes denote postconditions of events or actions in the statechart, and edges indicate that the successor node may be invoked after the predecessor node. Note that edges are associated with predicates indicating that the flow is only possible when the predicate is evaluated to be true. This predicate is a conjunction of the precondition of the event or the action represented by the node, and the guard condition of the transition. Since actions don't

have guard conditions, the incoming edges for actions are only associated with preconditions.

A fundamental difference between our EAFG and the flow graph representations in [33] and [12] is that EAFGs are transformed directly from statecharts whereas flow graphs in [33] and [12] are built from operation contracts.

As for traditional control flow graphs, EAFGs may contain paths that are infeasible. Theoretically speaking, determining infeasible paths in EAFGs can not be fully automated. Currently, we rely on user input to remove infeasible paths.

Note that an event may be a trigger for multiple transitions. In this case, each occurrence of the event is represented by a separate node in the EAFG. Hence a node is uniquely identified by a triplet: name of the event/action it represents, name of the source state of the transition where the even/action occurs, and the guard condition of the transition. The label of a node takes the format `sourceState_eventName_guard`. In the example shown in Figure 12, `pre(e1)` denotes the precondition of `e1` and `post(e1)` denotes the postcondition of `e1`. The event `e2` triggers two transitions: one from the state `s3` to the state `s5` and one from the state `s2` to the state `s4`. So in the corresponding EAFG, `post(e2)` occurs twice, in nodes `s2_e2_true` and `s3_e2_true` respectively. Note that the EAFG has a start node and an end node. A start node is always required since an EAFG is defined to have a single entry. An end node, however, is optional, and is only present when the statechart has a final state.

Figure 12 (a) Excerpt of a statechart (b) Corresponding EAFG

An EAFG can have cycles when there are cycles in the statechart, as illustrated in Figure 13.

Figure 13 (a) Excerpt of a statechart with a cycle (b) Corresponding EAFG

Recall that this research analyzes actions in an action sequence according to their written order and that entry and exit actions are moved to transitions (Section 2.2.4). Each action in an action sequence in a statechart is denoted by a separate node, and is connected to other actions according to their written order. Figure 14 shows an example where the

Figure 1 consists of three diagrams labeled (a), (b), and (c), illustrating the transformation of a Petri net into a Petri net with places.

(a) A Petri net with three places: s_1 , s_2 , and s_3 . Place s_2 is a double circle. Transitions are represented by arrows. Transition e_1 has input s_1 and output s_2 , labeled $e_1[q/a_1, a_2]$. Transition e_4 has input s_2 and output s_3 , labeled e_4/a_7 .

(b) A Petri net with three places: s_1 , s_2 , and s_3 . Place s_2 is a double circle. Transition e_1 has input s_1 and output s_2 , labeled $e_1[q/a_1, a_2, a_3]$. Transition e_4 has input s_2 and output s_3 , labeled $e_4/a_4, a_7$. There is a self-loop transition on s_2 labeled e_2/a_5 .

(c) A Petri net with 10 places, each represented by a box labeled $post(\text{event})$. The places are: $post(e_1)$, $post(a_1)$, $post(a_2)$, $post(a_3)$, $post(e_2)$, $post(a_5)$, $post(e_4)$, $post(a_4)$, $post(a_7)$, and $post(a_7)$. The transitions are represented by arrows with labels: g and $pre(e_1)$ into $post(e_1)$; $pre(a_1)$ from $post(e_1)$ to $post(a_1)$; $pre(a_2)$ from $post(a_1)$ to $post(a_2)$; $pre(a_3)$ from $post(a_2)$ to $post(a_3)$; $pre(e_2)$ from $post(a_3)$ to $post(e_2)$; $pre(a_5)$ from $post(e_2)$ to $post(a_5)$; $pre(e_4)$ from $post(a_5)$ to $post(e_4)$; $pre(a_4)$ from $post(e_4)$ to $post(a_4)$; $pre(a_7)$ from $post(a_4)$ to $post(a_7)$. There are also labels $s_1_e1_g$, $s_1_a1_g$, $s_1_a2_g$, $s_1_a3_g$, $s_2_a7_true$, $s_2_a4_true$, $s_2_e4_true$, and $s_2_a5_true$ associated with the places.

4.1.1 EAFG Metamodel

32

Figure 15 EAFG Metamodel

The metamodel defines precisely the form an EAFG can take. An EAFG consists of Nodes and Edges. A Node can be a StartNode, an EndNode, an EventNode, or an ActionNode. Each Node has zero or more Incoming edges and zero or more Outgoing edges, whereas each Edge has one Head node and one Tail node. A Node contains zero or one Postcondition and an Edge may contain zero or one Precondition and zero or one Guard. A Node may have zero or more predecessors and successors, both of which are Nodes.

Recall that a UML statechart may contain four kinds of events and eight kinds of actions. We enumerate in the following sections each kind of events and actions and discuss how they are represented in an EAFG and the possible definitions and uses that may occur in them. To ease the discussion, we use the abstract example in Figure 16 and Figure 17 (the

EAFG corresponding to Figure 16) whose elements are discussed in the following sections.

Figure 16 An abstract example

Figure 17 EAFG for Figure 16 (b)

4.1.2 Call events and call actions

In an EAFG, the postcondition of a call event is represented by a node and its precondition is associated with the incoming edge of the node, as shown in Figure 18.

This is also the case of events $m1(n1)$ and $m3()$ in Figure 16.

Figure 18 Representing call events in EAFGs

The invoked operation of a call event may reference attributes and links of the owning object of the state machine. For example, operation m1 may define and use the attributes of x1, (a1, a2), and/or define and use the links between x1 and x2. Such information can be obtained by analyzing the pre and postconditions of m1. Indeed, if a model element is constrained in a precondition, it is likely because that the element is used in the operation. A postcondition states what is true upon the completion of an operation, and in particular what the values of the model elements are. Section 4.3 discusses the rules to classify each model element in pre and postconditions as a definition, c-use, or p-use.

During state transitions, the owning object of the statechart may invoke an operation on an object or a set of objects. This kind of action is a call action. Recall that a call action takes the form: `object-set.action-name(parameterlist)` (Section 2.2.4). Note that in the case when the target is a set of objects, a copy of the action with its arguments is sent to every object in the set and each object handles this action copy independently, thus constituting a complex concurrent system. Since concurrency raises difficulties in tracking down data flow, and it is rarely encountered in most statecharts, this research doesn't address this issue and only handles the case when a call action has one single target object.

Pre- and postconditions of a call action are denoted in the same way as that of call events: see actions m2 (n2) and m4 in Figure 16. As in the case of call events, the information as to which attributes and links are defined and used during a call action can be obtained from the pre and postconditions in OCL (Section 4.3).

4.1.3 Signal events and send actions

Signal events and send actions denote asynchronous executions. Recall from Section 2.2.2 that the handler operation of a signal event can be specified in two different ways: In the first situation, there exists an operation in the class that accepts the signal with stereotype <<signal>> and with the same name as the signal. For our purpose, this is equivalent to a call event (the operation called is the handler) and the discussion of the previous section applies. In the second situation, there is no such operation with stereotype <<signal>> and it is assumed that the action out of the transition is the handler. For our purpose, this is equivalent to a call event where the pre- and postcondition of the operation are empty.

A send action occurs as a result of state transition, and results in a signal being sent. Its effect is a priori non-deterministic. This is because a send action is an asynchronous one-way communication, and the owning object of the state machine, which is the sender, may not be able to know whether the signal is received and processed by the receiver. To account for as many data flow interactions as possible, we assume that the signal will always be received and processed by the receiver. The receiver provides information on how the signal is dealt with by specifying an action handler operation. The pre and

postcondition of the action handler corresponding to the send action are added to the EAFG as discussed above.

4.1.4 Change events

Recall that change events occur when the value of a Boolean expression becomes true and that the handler of the event is the action out of the transition. It is modeled in the form of `when (a Boolean expression)`. The Boolean expression in the `when` expression is the condition under which the transition fires and can be considered as part of the precondition for a change event to occur. So a change event is represented in the EAFG as a node with an empty OCL condition and its incoming edge is associated with the Boolean expression, as shown in Figure 19.

Figure 19 Representing change events in EAFGs

As an example, the transition between `S3` and `S4` in Figure 16 is triggered by the change event `when a2=5`. It is represented in the EAFG (Figure 17) as a node `S3_(when a2=5)_true` which has an empty OCL expression and its incoming edge is associated with `a2=5`.

4.1.5 Time events

Recall that a time event triggers a state transition when a specific deadline expires. A time event is represented in an EAFG in a way similar to change events: a node with an

empty OCL expression and its incoming edge associated with the condition in the time event (referred to as time expression), as illustrated in Figure 20

Figure 20 Representing time events in EAFGs

4.1.6 Assignment actions

Recall that an assignment action sets the value of an object's attribute. It is specified as `aTargetObject.anAttribute:=expression` (Section 2.2.4). An assignment action can be considered as an operation with an empty precondition, and a postcondition that is the assignment expression itself in which the `@pre` postfix is placed (Section 2.3) after each model element occurring in the right-hand-side of the expression. The addition of the `@pre` postfix will find a justification when we discuss the identification of definitions and uses in OCL expressions in Section 4.3. In an EAFG, an assignment action is then modeled by a node labeled with the transformed assignment expression whose incoming edge doesn't have any guards or preconditions (Figure 21). For instance in Figure 16, the action of the transition between `S1` and `S2` is an assignment action, `a1=a2+1`. It is represented in the corresponding EAFG with the node `S2_(a1=a2+1)_true`, which contains the postcondition of the assignment action, `a1=a2@pre+1`.

Figure 21 Representing assignment actions in EAFGs

4.1.7 Create actions and destroy actions

Recall that a create action instantiates and initializes an instance of a class and a destroy action deletes an existing instance as well as all the links between the destroyed object and other objects. In an EAFG, create actions and destroy actions are modeled in the same way as a call or a send action: a node is used to denote the postcondition of the action handler operation and the incoming edge of the node is associated with the precondition of the action handler. The handler of a create action (resp. destroy action) is the corresponding constructor (resp. destructor).

4.1.8 Terminate actions, return actions, and uninterpreted actions

A terminate action results in the self-destruction of the owning object of the state machines. Since the owning object doesn't exit after a terminate action, we are not interested in such kind of actions. We don't need to consider return actions since a return action simply returns the control to the caller and no flow of data is involved;

uninterpreted actions are not accounted for either because their semantics can only be completely specified by a specific implementation language.

4.1.9 Guard conditions

As described in Section 4.1.1, a guard condition in a statechart belongs in an EAFG to the incoming edge of the node that denotes the postcondition of the trigger event that the guard condition is associated with. An instance in Figure 16, $a2 > 0$ is the guard condition for the trigger event $m3()$. It is associated with the incoming edge of the node $S2_m3_a2 > 0$.

4.1.10 Activities

Recall that an activity is an ongoing non-atomic execution while an object is in a state. An activity either completes its execution and then sends a completion event or is interrupted by the arrival of an event. If the `do` transition section of a state specifies an activity (or a sequence of activities), there is no simple way to specify its effect if it is interrupted by the occurrence of an event. For instance, there is no way to specify that in the OCL postcondition of the activity. If the `do` transition section specifies a sequence of actions (each action is uninterruptible but the sequence is), we can use their OCL postconditions. However, since we cannot foresee the occurrence of an event and thus when the sequence will be interrupted, we have to model every possible interruption in the EAFG. If the sequence has n actions and can be interrupted by the firing of m transitions, we have to model $(n-1) * m$ possible interruptions. Considering that `do` transitions are not used often and when used, represent ongoing activities that do not change the state and do not participate in any data flow within the statechart (e.g. the

sound alarm activity in Figure 9), we ignore them in the construction of the EAFG. Future work will address this limitation.

4.1.11 Complementing EAFG metamodel with event and action types

We've seen from the above discussion that an EAFG may model four types of events, which are call events, signal events, change events, and time events, and five types of actions, which are call actions, send actions, assignment actions, create actions and destroy actions. The EAFG metamodel defined in Figure 15 (Section 4.1.1) is therefore complemented with such information, as shown in Figure 22. Note that an ActivityNode is included for future extensions.

Figure 22 EAFG metamodel with event/action types

4.2 Modified EAFGs

Recall that we assume that pre- and postconditions, as well as guard conditions in statecharts are specified with OCL expressions. These OCL expressions are associated with edges and nodes in the EAFG: guard conditions and preconditions are associated with edges and postconditions are associated with nodes. As described in Section 4.1, those OCL expressions can be used to identify definition and uses. Actually, nodes (i.e., postconditions) entail definitions and uses, whereas edges (i.e., guards and preconditions) only entail uses (Section 4.3). For instance, without entering into too many details, the following operation postcondition can be used to determine that attributes `attribute1` and `attribute2` are defined, i.e., given a value, during the execution of the operation:

```
attribute1 > 0 and attribute1 = 1 implies attribute2 = 2
```

However, OCL constructs in postconditions can be used to refine the analysis of definitions and uses as they may suggest some control flow in the execution of the operation. For instance, in the example above, `attribute2` is given a value under a specific condition (i.e., `attribute1=1`).

We thus modify the EAFG, and more specifically the nodes (they hold postconditions), to take advantage of this information and then refine the data flow analysis.

In OCL postconditions, four OCL operators, namely `implies`, `if-then-else`, `or` and `xor`, suggest some control flow in the operation execution. Indeed, they are used to state alternative operation results. This is obvious for operators `implies` and `if-then-else`: In addition to the alternative results, they state the conditions under which those

results are obtained (Boolean expression on the left of the `implies` operator, predicate in the `if-then-else` expression). Operators `or` and `xor` can also be used to state alternative results, though, not necessarily along with the conditions under which those results are obtained. For instance, `postcondition state=#On xor state=#Off` states that the operation may result in two different states, without providing a condition that would lead to `#On` or `#Off`. Similarly, `postcondition attribute1=1 or attribute2=10` states that the operation may results in two different modifications of the class's state. Though the last two examples show postconditions described with few (one would say, too few) details, we have to account for such situations. Note that the level of details of OCL expressions in postconditions is further discussed in Section 4.3.1.

To account for possible control flow in operations, as suggested by their postconditions, we first acknowledge that OCL postconditions are conjunctions of *conjuncts*. Those *conjuncts* are OCL expressions that can contain connectives `or`, `xor`, `implies` and `if-then-else`. We further decompose the *conjuncts* into *terms*, that is, OCL expressions without connectives. For instance, the postcondition of the event `resume` in the Cruise Control case study (Section 6.1) has two *conjuncts* and four *terms*, as illustrated in Figure 23.

```

resume()
pre: ...
post: if (
    self.cruiseSpeed@pre > 0
)
    then
        self.throttle = self.cruiseSpeed / CruiseController.st
        + (self.cruiseSpeed - self.speed) / CruiseController.diff
    else
        true
    endif
    and
    self.state =#cruising

```

Conjunct one
 Term1
 Term2
 Term3
 Conjunct two
 Term4

Figure 23 Example of terms in a postcondition

Note that the Boolean expression on the left of the `implies` expression and the predicate part of the `if-then-else` expression are always considered as one term even when they contain disjuncts or `if-then-else` or `implies` expressions. The reason is that we consider those OCL expressions only contain uses of model elements (if a model element is constrained in the Boolean expression on the left of the `implies` expression or the predicate part of the `if-then-else` expression, it is likely because that element is used in the operation): definitions only appear in the `then` or `else` parts of `if-then-else` operations or on the right part of the `implies` expressions. This is however an assumption as a designer is allowed by the OCL syntax to describe how a model element is set a value in the predicate part of an `if-then-else` expression, thus suggesting a definition in that predicate. This limitation is further discussed in Section 4.5, where possible solutions are investigated.

The decomposition into conjuncts and terms is then used to transform nodes. We introduce the notion of *compound node* and *basic node*. A node in the EAFG is either a compound node or a basic node. An operation postcondition is associated with a compound node, and a compound node is composed of basic nodes with a start and end

nodes. Basic nodes associated with a compound node are linked by edges, thus representing the control flow suggested by the postcondition. When decomposing a compound node into basic nodes, conjuncts in the OCL postcondition are considered in sequence, as they appear in the OCL expression, and are transformed into basic nodes linked by edges: the starting (resp. ending) basic node for the compound node corresponds to the first (resp. last) conjunct in the OCL expression. If a conjunct contains any of the four connectives `or`, `xor`, `implies` and `if-then-else`, the corresponding basic node is further decomposed according to the four different templates in Figure 24.

Note that in these templates, nodes are associated with the terms connected by `xor`, `or` operators, and terms in the `then` or `else` parts of `if-then-else` operations or on the right part of the `implies` expressions. The predicate parts of `if-then-else` or `implies` expressions are associated with edges. In case those nodes also contain the four connectives, they are further decomposed using the same templates. For instance, with postcondition `if c then a or b else true`, first the `if then else` is decomposed and then `a or b` is also decomposed.

Figure 24 Templates for connectives

As an example, the compound node (with basic nodes) for `resume()` (Figure 23) is shown in Figure 25.

Figure 25 Compound node for `resume()`

Note that, though a conjunction in an OCL postcondition does not specify any order (it is simply a Boolean expression), our transformation explicitly creates one: the order of appearance in the OCL postcondition expression. This artificial order has an impact on the layout of the EAFG, and as a consequence on the identification of definition use pairs and def clear paths. This issue is further discussed in Section 4.5, where possible solutions are investigated.

The EAFG metamodel (Figure 22) is thus modified to account for the structure of compound nodes (Figure 26). A Node is classified as being either a BasicNode or a CompoundNode. A CompoundNode can be an EventNode, ActionNode, or ActivityNode. To avoid cluttering the diagram, the types of events and actions that a node can model are not shown. A CompoundNode consists of a start BasicNode and an end BasicNode, which are sufficient to determine all the basic nodes the compound node contains (by transitivity), thanks to the inherited association predecessor-successor between basic nodes. Note that an abstract class Constraint is created as the superclass of Postcondition, Precondition and Guard, each of which contains one or more than one Term.

Figure 26 Modified EAFG metamodel

4.3 Determining definitions and uses in edges and nodes

In this section, we describe how definitions and uses of model elements can be identified from OCL expressions and how this information can be added to the EAFG. Sections 4.3.1 to 4.3.3 discuss several important issues with postconditions written in OCL. Section 4.3.4 presents the metamodel of EAFGs with data flow information. Section 4.3.5 provides instantiation examples for the metamodel, followed by detailed descriptions of the rules for determining definitions and uses in nodes and edges (Section 4.3.6 and Section 4.3.7). Additional rules are defined for OCL expressions with

collection operations (Section 4.3.8). Section 4.3.9 summarizes the rules discussed in previous sections.

4.3.1 Levels of precision in postconditions

One issue is that postconditions can be specified at different levels of precision. Three levels of precision for writing postconditions were defined in [8]. The lowest level of precision only defines the ranges/enumerations of values expected upon the completion of the method. The intermediate level of precision distinguishes the standard situations from exceptional situations, while with the highest level of precision, every distinct condition, possibly resulting from a different set of inputs or system state, is distinguished in the postcondition. To illustrate this, an abstract example is provided in Table 2.

Highest level	Intermediate level	Lowest level
if A1 then B1 else if A2 then B2 else B3	if A1 then B1 else B2 or B3 // A1 is assumed to be the standard situation	B1 or B2 or B3

Table 2 Three levels of precision in postconditions, an example from [8]

Because we want to capture all the data flow information reflected by postconditions, ideally we would prefer postconditions to be as precise as possible. However, our strategy is still applicable to the intermediate and lowest levels of precision, though with lower levels of precision, predicate uses of a model element may be missed, which will lead to less precise data flow analysis. The impact of the precision of postcondition on the data flow analysis results will be the subject of future work.

4.3.2 Query methods

Since OCL is a declarative language, OCL expressions only contain query operations, that is, operations that do not have any impact on the system state [32]. A query operation simply returns a value and its postcondition describes how that value is constrained using the keyword `result`. Since we are attempting to reveal all the data flow information contained in postconditions, we need to replace every occurrence of a query with the expression assigned to `result`.

4.3.3 Local definitions

`let` expressions in OCL allow modelers to define a variable or operation that can be used (possibly several times) in a constraint, for example, an invariant or a pre- or postcondition [29]. The scope of such a definition is the constraint in which it is defined. The stereotype `<<definition>>` can be used to extend the scope of such definitions to all the constraints of a class. To simplify the analysis, every occurrence of variables or operations defined by `let` expressions is replaced by the defining expressions.

Local variables may also be defined and used in a transition's action sequence [7]. For instance, a transition can trigger the following two actions in sequence: `v=op1()` and `op2(v)`. In such a case, every use of the variable is replaced with the `result` part of the postcondition of the operation that defines it. In our example, `v` is replaced with the `result` part of the postcondition of `op1()`.

4.3.4 EAFG metamodel with data flow information

Before we define the rules for identifying definitions and uses in pre and postconditions, we first present the modified EAFG metamodel with data flow information (Figure 27). It is built from the metamodel in Figure 26. Classes that model data flow information and the elements of a term are added to facilitate the definition of rules presented in the following sections. A `Term` consists of `ModelElements` and `Operations`. A `ModelElement` refers to the `Attributes` and `Links` of an instance of a class, and `Parameters` of a context operation since `let` expressions and query operations are removed. An `Operation` can be either an operation on `OCL Basic Type` or `OCL Collection Type`, which is further classified into `IterativeOperation` and `Non-iterativeOperation` (Section 4.3.8). A collection operation in OCL takes the form `collection->op(parameter)`. The collection of a `CollectionTypeOperation` refers to the link of an instance of a class. The `ModelElements` involved in the parameter of a `CollectionType` operation can be the parameters of the context operation or refer to the links and attributes of the context object (Section 4.3.8). For instance, in the OCL expression `self.reservation->includes(reserve)`, `self.reservation` is an instance of link, and `includes` is an instance of a subclass of `CollectionTypeOperation`, and `reserve` is an instance of the `ModelElement` involved in the parameter of the collection operation. The association class `ValueInTerm` between `Term` and `ModelElement` consists of the `Values` of a model element in a `Term`. The kind of `Value` appearing in a term can be either a `PreValue` or an `AfterValue`. Both the `PreValue` and `AfterValue` of a model element may occur in a `Term`, thus the multiplicity of 1..2. Note that in case the

4.3.5 Instantiation examples

This section uses two examples to illustrate the instantiation of the metamodel in Figure 27.

The first operation contract (shown below) only contains basic type operations. Figure 28 shows an excerpt of the corresponding metamodel instance.

<code>Class1::op1 (param: int)</code>	
<code>Pre: param < 10</code>	<code>Term0 (T0)</code>
<code>Post: att1 = att2 + param</code>	<code>Term1 (T1)</code>
<code> and</code>	
<code> att2 = att2@pre + 1</code>	<code>Term2 (T2)</code>

The Edge of the CompoundNode is associated with T0, which has an in parameter param, an in parameter is a PreValue which is a Puse when associated with an edge (Section 4.3.6). The start basic node of the compound node is associated with T1 and the end basic node with T2. T1 contains the AfterValues of att1 and att2, and in parameter param, which is a PreValue: the AfterValue of att1 is a Def, the AfterValue of att2 is Cuse, and Param is a Cuse (Section 4.3.7). T2 contains the PreValue and the AfterValue of att2: the PreValue is a Cuse whereas the AfterValue is a Def.

The second operation contract has collection operations, as shown below.

<code>Class2::op2 (param: A)</code>	
<code>Pre: not self.roleName->includes (param)</code>	<code>Term0 (T0)</code>
<code>Post: self.roleName->includes (param)</code>	<code>Term1 (T1)</code>

The Edge of the CompoundNode is associated with T0, which has an in parameter param, and the link self.roleName. They are both PreValues and are Puses. The start basic node and the end basic node are the same node which is associated with T1. T1

contains the AfterValue of the link `self.roleName`, which is a def, and the in parameter `param`, which is a c-use (Section 4.3.8).

Figure 28 Metamodel instantiation: an example without collection operations

The way definitions, c-uses and p-uses are identified in the above two examples is discussed in the following sections. Section 4.3.6 and Section 4.3.7 do not consider collection operations, i.e., model elements not associated with `CollectionTypeOperation`. Section 4.3.8 focuses on terms that involve collection operations.

Figure 29 Metamodel instantiation: an example with a collection operation

4.3.6 Model elements in edges

Recall that an edge in an EAFG is associated with a precondition, a guard, the Boolean expression on the left of the `implies` operator, or the predicate part of an `if-then-else` expression (Section 4.2). Since these constraints are typically implemented as conditional statements, the model elements that appear in an edge are classified as `p-uses`. In the following example from the Cruise Control case study (Section 6.1), `self.state` is `p-used` in the precondition:

```
engineOn()
pre: self.state = #idle
```

Both pre-values and after-values of a model element may occur in the edges within a compound node. Now we look at two examples where edges within compound nodes are associated with the pre-value or after-value of a model element. In the postcondition of the event Resume, the pre-value of `self.cruiseSpeed` in Term1 is a p-use of `self.cruiseSpeed`.

```

resume()
post:
if (self.cruiseSpeed@pre > 0)                                Term1
    then
        self.throttle = self.cruiseSpeed / CruiseController.st
        + (self.cruiseSpeed - self.speed) / CruiseController.diff    Term2
    else true                                                  Term3
endif
and
self.state =#cruising                                         Term4

```

In the following example, the after-value of `a` appears in Term2, it is considered as a p-use.

```

Op2()
post:
a = b@pre + 1                                                  Term1
if a = c@pre                                                  Term2
    then d = d@pre + 1                                         Term3
    else true                                                  Term4
endif                                                         Term5

```

Note that operations may carry a list of parameters. Based on the direction of information flow of a parameter, parameters can be divided into four kinds: in, out, inout and return parameters. An in parameter is an input parameter that is passed by value and may not be modified. An out parameter is an output parameter whose final value is available to the caller. An inout parameter is an input parameter that may be modified and its final value is available to the caller. A return parameter is semantically equivalent to an out parameter, but the result is available for use in an inline expression [27].

When parameters of an operation appear in a precondition or a guard condition, they can be either in parameters or inout parameters, and can be considered as pre-values. Postconditions, however, may contain any kind of the abovementioned parameters. An in parameter in a postcondition can always be considered as a pre-value since an in parameter may not be modified during the operation. For an inout parameter, the postcondition of the operation may contain both its pre and after-values since it may be modified in an operation. An out or return parameter are modified in an operation. So its appearances in postconditions can be considered as after-values.

Like other model elements, parameters appearing in an edge are considered as p-uses. One example is given below where the in parameter of `addReservation(v:Reservation)` is p-used in the precondition.

```
addReservation(v:Reservation)
pre:
not self.reservation->includes(v)
```

Rule 1 states that the model elements associated with an edge are identified as p-uses.

Rule 1 Model elements in edges	
context	Edge
<code>self.term.valueInTerm.value.dataFlow->forAll(d: DataFlow d.kind = #Puse)</code>	

4.3.7 Model elements in nodes

Recall that a node contains those OCL expressions in a postcondition that are not in the predicate part of an if-then-else expression or on the left part of an implies expressions (Section 4.2). A postcondition can describe what has been changed during the operation, as well as those conditions that have not been changed and remained true

after the operation. The former was referred to as *change specifications* and the latter as *frame rules* [23]. Section 4.3.7.1 and Section 4.3.7.2 discuss data flow information in frames rules and change specifications respectively.

4.3.7.1 Frame rules in nodes

A frame rule specifies what does not change during an operation [23]. A term T , is a frame rule if either of the following two conditions holds:

1. T takes the format `aModelElement = aModelElement@pre`.
2. T is semantically equivalent to a term appearing in the precondition of the operation.

An example for condition 1 is given below. The operation `Op2()` assigns the pre-value of attribute `c` to attribute `a` and doesn't change the value of attribute `b`. `Term1` in the following postcondition is a frame rule, whereas `Term2` is a change specification.

```
Op2()
post:
self.b = self.b@pre                                Term1
and
self.a = self.c@pre                                Term2
```

OCL provides a rich set of operations on collection types, many of which are semantically equivalent. If a term occurring in the postcondition of an operation is semantically equivalent to a term occurring in the precondition of the operation, then the former is a frame rule. The following postcondition is a frame rule because the postcondition `collection->exists(expr)` is semantically equivalent to the precondition `collection->select(expr)->size>0`.

```
Op3 ()
Pre: collection->select(expr)->size>0
Post: collection->exists(expr)
```

Term1
Term2

Because frame rules specify what does not change, they do not really provide any data flow information: if a variable or link is defined or used during an operation, the change specification will certainly assert such information. Therefore frame rules don't contain any definitions or uses.

Rule 2 states that model elements appearing in frame rules in a node don't have any data flow information. Note that edges don't have terms that are frame rules.

Rule 2 Frame rules in nodes

```
context BasicNode
self.term.isFrameRule = true
implies
self.term.valueInTerm.value.dataFlow->forAll(d:DataFlow|d.kind = #Null)
```

Note that frame rules may help the identification of definitions and uses in postconditions. Frame rules are usually needed when a postcondition compares the after-value of a changed model element with the after-values of other model elements. For example, an operation `op4()` assigns the pre-value of attribute `b` plus one to attribute `a` and it doesn't change the value of attribute `b`. The following is an ambiguous postcondition of `op4()` because we can't decide whether `a` and `b` are defined or used:

```
Op4 ()
Post:
self.a = self.b + 1
```

Term1

It could be written as `self.b = self.a - 1`. When adding a frame rule to the postcondition, stating that `b` is not changed, definitions and uses can be determined:

```

Op4 ()
Post:
self.b = self.b@pre                                Term1
and
self.a = self.b + 1                                Term2

```

Now it is clear that `b` is not changed during the operation, and `Term1` doesn't have any definitions or uses. And in `Term2`, the occurrence of `self.b` is a use and the occurrence of `self.a` is a definition. This is further discussed in Section 4.3.7.2.2.

Our research assumes that if an element appears in a postcondition, whether or not it is changed during an operation has to be specified. That is to say, if a model element is not changed during the operation, there must be a frame rule saying so, otherwise there's a change specification describing what the change is. On the other hand, if an element doesn't appear anywhere in a postcondition, then we assume it is not changed. As the above example shows, this assumption guarantees that definitions and uses are correctly determined for after-values in postconditions.

4.3.7.2 Change specifications

A change specification can contain both pre-values and after-values of model elements. Section 4.3.7.2.1 and Section 4.3.7.2.2 define rules for pre-values and after-values in sequence.

4.3.7.2.1 Pre-values in nodes

The occurrence of a pre-value in a change specification is considered as a c-use, as this states that the operation uses the value before the execution to set another element's value. For example, in the postcondition of `op2()` (Section 4.3.7.1), `self.a =`

`self.c@pre` shows that attribute `self.a`'s after-value equals `self.c`'s pre-value. The occurrence of `self.c@pre` is a c-use of `self.c`.

The following rule states that the pre-values of model elements appearing in a node are considered as c-uses provided that the terms in which these pre-values occur are not frame rules.

Rule 3 Pre-values in nodes

```
context BasicNode
self.term.isFrameRule = false
implies
self.term.valueInTerm.value->select(v:Value|v.kind = #PreValue).dataFlow->
                                forall (d:DataFlow|d.kind = #Cuse)
```

Recall that the occurrence of an `in` parameter in a node can always be considered as a pre-value. So Rule 3 applies, that is, an `in` parameter in a node is a c-use. One example is given below where in the postcondition of `Rent (r: Rental)`, `Term2` has a c-use of `r`.

```
Rent (r:Rental) :: post
self.state = #Rented                                Term1
and self.currentRental = r                            Term2
```

Both the pre- and after-values of an `inout` parameter may occur in a node. When a pre-value of an `inout` parameter appears in a node, Rule 3 applies, that is, the pre-value is identified as a c-use. For an `out` or `return` parameter, its appearances in postconditions can be considered as after-values. Identifying definitions and uses of after-values in postconditions are discussed in the next section.

4.3.7.2.2 After-values in nodes

Two cases are identified when an after-value, say `v`, occurs in nodes:

Case 1: v is the only after-value in a term. In this case, other values in the term are either literals, e.g., $v=1$, or pre-values of model elements, e.g., $v=u@pre+1$. Since the term shows the change of v during the operation, v is a definition. Below is an example where Term2 contains a definition of `self.state`.

```

Off()
pre: self.state = #cruising           Term1
post: self.state = #standby           Term2

```

Case 2: v is not the only after-value in a term. As discussed in Section 4.3.7.1, if v is not changed by the operation, then there must be a frame rule with respect to v . Otherwise, there is a change specification that defines v . Therefore if v appears in a frame rule in the postcondition or if v is defined in a change specification somewhere else in the postcondition, then the occurrence of v in this term is a c-use of v ; otherwise this term is a change specification that defines v .

Below are two examples. In the postcondition of `Op5()`, Term2 has two after-values, `self.a` and `self.b`. `self.b` also appears in Term1, which is a frame rule. So `self.b` in Term2 is a c-use whereas `self.a` in Term2 is a definition. In the postcondition of `Op6()`, `self.b` is defined in Term1. Hence its appearance in Term2 is a c-use and the occurrence of `self.a` is a definition.

```

Op5()
post:
self.b = self.b@pre           Term1
and
self.a = self.b + 1           Term2

Op6()
post:
self.b = self.c@pre           Term1
and
self.a = self.b + 1           Term2

```

Rule 4 summarizes the above two cases when an after-value of a model element appears in a node. Note that terms in a compound node are all the terms of the basic nodes in the compound node, i.e., all the basic nodes that belong to paths starting with the start basic node of the compound node and ending with the end basic node of the compound node. `fTerm` refers to all the terms in the compound node that are frame rules, and `nfTerm` refers to all the terms in the compound node that are not frame rules. `dVal` refers to those values in `nfTerm` that are definitions. In Rule 4, Line 1 to 4 correspond to the case when `v` is the only after-value in a term and `v` is identified as a definition. Line 5 to 11 correspond to the case when `v` is not the only after-value in a term: `v` is c-used if it appears in a frame rule of the compound node (Line 7 and 8), or defined in another term in the compound node (Line 9 and 10), otherwise it is a definition (Line 11).

Rule 4 After-values in nodes

```

context BasicNode
let fTerm:Set(Term)=self.compoundNode.term->select(t:Term|t.isFrameRule =
                                                    true)
let nfTerm:Set(Term)=self.compoundNode.term->select(t:Term|t.isFrameRule =
                                                    false)
let dVal:Set(Value)=nfTerm.valueInTerm.value->
                    select(v:Value|v.dataflow.kind =#Def)

self.term.isFrameRule = false
implies (
self.term.valueInTerm.value->select(kind = #AfterValue)->
  forall(
1    v:Value|self.term.valueInTerm.value->
2      select(kind = #AfterValue)->size=1
3    implies
4    v.dataFlow.kind = #Def
5    and
6    self.term.valueInTerm.value->select(kind = #AfterValue)->size>1
7    implies
8    if fTerm.valueInTerm.value->includes(v)
9    then v.dataFlow.kind = #Cuse
10   else if dVal->includes(v)
11   then v.dataFlow.kind = #Cuse
12   else v.dataflow.kind = #Def
13   endif
14   endif
)
)

```

Note that the rules presented so far for identifying definition and uses in pre and postconditions can be fully automated (through parsing of OCL expressions). These rules are sufficient when an OCL expression doesn't contain any collection operations. However, when collection operations are involved, these rules need to be extended. The next section addresses this issue.

4.3.8 Contracts with collection operations

A collection operation in OCL takes the form `collection->op(parameter)`. A collection can be defined explicitly by a literal, obtained by navigation or operations on collections [30], as exemplified in Table 3:

Ways to obtain a collection	Examples
Explicitly defined by a literal	Set {1, 2, 3, 5}
Obtained by navigation	<code>context Title</code> <code>self.copy</code>
Obtained by collection operations	<code>bag->union(bag2)</code>

Table 3 Examples of obtaining a collection

Collection operations can be broadly divided into two categories: Iterative collection operations and non-iterative collection operations. Iterative collection operations in this work refer to those operations that iterate over collection elements and take `OclExpression` as parameter. We make this distinction because iterative and non-iterative collection operations involve different data flow information and hence require separate rules. Section 4.3.8.1 and 4.3.8.2 discuss non-iterative and iterative collection operations respectively.

4.3.8.1 Non-iterative collection operations

Among non-iterative collection operations, two types of operations are identified: Type (1) operations that return a new collection and Type (2) operations that return a Boolean, Integer or an element in the collection. Some of those operations require a parameter and we first justify that those parameters are c-used. Then Section 4.3.8.1.1 and Section 4.3.8.1.2 discuss the rules for the collection itself in Type (1) operations and Type (2) operations respectively.

The parameter of a non-iterative collection operation can refer to the parameters of the context operation, and attributes and links of the context object. When the parameter appears on an edge, the model elements involved in it are identified as p-uses, that is, Rule 1 (Section 4.3.6) applies as it is valid for any type of model elements.

When a non-iterative collection appears in nodes, both pre-values and after-values of model elements may occur in its parameter. Their occurrences are considered as c-uses since they don't suggest any change of model elements, e.g., creation of a new object.

Below is an example where, both pre and postcondition uses `reserve` that is a parameter of the context operation.

```
Member::addReservation(reserve: Reservation)
Pre: not self.reservation->includes(reserve)
Post: self.reservation ->includes(reserve)
```

`Term1`
`Term2`

Rule 5 is defined for model elements in parameters of non-iterative collections when they are in nodes. Note that we can't simply use Rule 3, which is applicable for pre-values only, as after-values may also appear in the parameter.

Rule 5 Parameters of non-iterative collection operations in nodes

```

context BasicNode
self.term.isFrameRule = false
implies
self.term.operation->select
(o:Operation|o.oclIsTypeOf (NonIterative)).involvedInOpParam.dataflow
->forAll (d:DataFlow|d.kind = #Cuse)

```

4.3.8.1.1 Type (1) non-iterative collection operations

Table 4 lists Type (1) collection operations:

Set	Set->union(set2) set->union(bag) set->intersection(set2) set->intersection(bag) set-(set2) set->symmetricDifference(set2) set->including(object) set->excluding(object) set->asSequence() set->asBag()
Bag	Bag->union(bag2) bag->union(set) bag->intersection(bag2) bag->intersection(set) bag->including(object) bag->excluding(object) bag->asSequence() bag->asSet()
Sequence	sequence->union(sequence2) sequence->including(object) sequence->excluding(object) sequence->append(object) sequence->prepend(object) sequence->asBag() sequence->asSet() sequence->subsequence(integer, integer)

Table 4 Type (1) non-iterative collection operations

When a set/bag/sequence is with an edge or its pre-value is in a node, i.e., there is an instance of link for this set/bag/sequence, Rule 1, Rule 2, and Rule 3 apply (Section 4.3.6 and Section 4.3.7). That is, a set/bag/sequence is considered as a p-use if it appears in an edge, and a c-use if its pre-value is in a node provided that the term in

which it occurs is not a frame rule. For instance, in the following postcondition of Op7 (),
sequence@pre is c-used to define sequence.

```
Op7 ()
post:
sequence = sequence@pre->append(object)
```

When the after-values of a set/bag/sequence in Type (1) operations appear in nodes, they are identified as c-uses. This is because these operations do not modify the collection on which they are applied but generate new collections from them. They are typically used in OCL contracts to define a constraint on the resulting collection (e.g., self.roleName2->union(param)->size()=10) or to define the value of model elements (e.g., self.roleName1=self.roleName2->asSet()). In both cases, the collection on which the operation is applied (i.e., self.roleName2) is used only.

So, instead of Rule 4 (Section 4.3.7), Rule 6 is defined for this particular situation: it states that the after-values of a collection in a Type (1) operation are considered as c-uses provided that the collection doesn't appear in a frame rule.

Rule 6 After-values of collections in Type (1) non-iterative operations

```
Context BasicNode
self.term.isFrameRule = #false
implies
self.term.operation->select
  (o:Operation|o.ocliIsTypeOf(Type1)).collection->select
    (v:Value|v.kind = #AfterValue).dataflow->forAll
      (d:DataFlow|d.kind = #Cuse)
```

4.3.8.1.2 Type (2) non-iterative collection operations

Type (2) non-iterative collection operations refer to those operations that return a Boolean value, an integer or an element of the collection. Table 5 lists the Type (2) non-iterative collection operations.

```

collection->isEmpty()
collection->notEmpty()
collection->includes(object)
collection->excludes(object)
collection->includesAll(collection2)
collection->excludesAll(collection2)
collection->size()
collection->count(object)
sequence->at(i)
sequence->first
sequence->last

```

Table 5 Type (2) non-iterative collection operations

Note that two of the collection operations above test the numeric properties of a collection: `collection->size()` returns the number of elements in `collection`, and `collection->count(object)` counts the number of times the object occurs in `collection`.

One question is when these operations appear in contracts, what should be considered as being defined (or used): `collection` only or both `collection` and `collection->size/count(object)`. Our approach is to consider `collection` only. Since `collection->size` and `collection->count(object)` are all properties of a collection, the use or change of properties of `collection` implies the use or change of `collection`. Therefore, test cases that traverse `du pairs/def clear paths` (Section 4.4) with respect to `collection->size /collection->count(object)` will be covered by test cases that traverse `du pairs/def clear paths` with respect to `collection`. But the converse is not true. Also, considering `collection` only eases the automatic identification of definitions and uses in contracts.

When a Type (2) non-iterative collection operation appears in an edge, Rule 1 applies (Section 4.3.6), i.e., the collection is p-used. When a Type (2) non-iterative collection operation applies on the pre-value of a collection and appears in a node, the collection is

considered as c-used provided that the term in which the collection occurs is not a frame rule (Rule 3 Section 4.3.7). Unlike Type (1) non-iterative collection operation, Type (2) non-iterative collection operations may modify the collection on which they are applied. For instance, suppose `self.roleName->includes(param)` is part of a postcondition and is not a frame rule. It represents a modification to `self.roleName` during the execution of the operation: `param` is not an element of `self.roleName` before the execution but it is after the execution. Therefore, Rule 4 (Section 4.3.7) is used to determine definitions and uses of the collection when its after-value appears in a node.

Below are two examples to illustrate how the rules are applied.

<code>Reservation::cancel()</code>	
<code>Pre: self.state=#Pending or self.state = #Outstanding</code>	Term1
<code>Post: self->whenCancelled = TimePoint.currentTime()</code>	Term2
<code>and self.state =#Cancelled</code>	Term3
<code>and if self.copy@pre->notEmpty</code>	Term4
<code>then self.copy@pre.state=#ForRent</code>	Term5
<code>else true</code>	Term6
<code>and self.copy->isEmpty</code>	Term7
 <code>Member::addReservation(reserve: Reservation)</code>	
<code>Pre: not self.reservation->includes(reserve)</code>	Term1
<code>Post: self.reservation ->includes(reserve)</code>	Term2

In the postcondition of `Reservation::cancel()`, Term4 contains a p-use of the pre-value of `self.copy` since it appears on an edge (Rule 1). Term7 contains a definition of `self.copy` because `self.copy->isEmpty` is not a frame rule and `self.copy` is the only after-value in the term (Rule 4).

In the postcondition of `Member::addReservation(reserve:Reservation)`, the precondition Term1 contains a p-use of `self.reservation` (Rule 1). And the postcondition Term2 contains a definition of `self.reservation`, since Term2 is not a frame rule and `self.reservation` is the only after-value in this term.

4.3.8.2 Iterative collection operations

Iterative Collection Operations in this work refer to those operations that iterate over the elements of a collection and take `OclExpression` as parameter. Table 6 lists the iterative collection operations.

<code>collection->select</code> <code>collection->collect</code> <code>collection->reject</code> <code>collection->forAll</code> <code>collection->exist</code> <code>collection->isUnique</code> <code>collection->sortedBy</code> <code>collection->any</code> <code>collection->one</code> <code>collection->iterate</code>

Table 6 Iterative collection operations

These operations all take an `OclExpression` as parameter and can follow either of the following syntax [30], where `v` is an optional iterator:

```
collection->op(v: Type|OclExpression with v)
collection->op(v|OclExpression with v)
collection->op(OclExpression)
```

Note that `OclExpression` may refer to the elements of the collection on which the operation is applied, attributes/links of the context object or any element in the collection, and parameters of the context operation. For example:

- `self.title.reservation->select(state=#Pending)` refers to attribute `state` of elements in `self.title.reservation`.
- In `Copy.allInstances->exists(c:copy|c.barcode=barcode)`, `c.barcode` refers to the attribute `barcode` of `Copy` objects, and `barcode` is a parameter of the context operation.

- In `self.reservation->selects`
`(r:Reservation|r<>self.oldestPending)`, the elements of
`self.reservation` are referred to.

Note that we interpret the definition of an object as the definitions of the attributes and links of the object, and the c-use (or p-use) of the object as the c-use (or p-use) of the attributes and links of the object. This is based on the convention for structured variables such as records in procedural languages [13]. So in the last example, occurrences of objects of `Reservation` are considered as the occurrences of the attributes and links of these objects.

The above examples suggest that because of `OclExpression`, not only the collection on which the operation is applied but also the attributes, links, and parameters involved in `OclExpression` may be defined (or used) in contracts. However, it is difficult to characterize the exact objects whose attributes and links are defined (or used). This would require (likely very complex) semantic analysis of OCL expressions. Our approach is to consider that all the objects in the class are defined (or used). This is a conservative approach as it will not miss any definitions (or uses) but may lead to the identification of definitions (or uses) that do not actually exist. This approach has an impact on the identification of def clear paths and du pairs as discussed in Section 4.5.

The following two sections deal with iterative collection operations in edges and nodes respectively.

4.3.8.2.1 Iterative collection operations in edges

When an iterative collection operation appears in an edge, Rule 1 (Section 4.3.6) applies: the model elements that are considered as p-uses are the collection on which the operation is applied, the attributes/links that are referred to in `OclExpression` and parameters of the context operation that are referred to in `OclExpression`. Below is an example where iterative collection operations appear in a precondition.

```
Title::setForSale()
pre: self.state = #ForRent                                Term1
    and
    not self.reservation->exists
        (r:Reservation|r.state = #Pending
        or
        r.state = #Outstanding)                            Term2
    and
    not self.copy->exists
        (c:copy|c.state = #OnHold
        or
        c.state = #Rented)                                Term3
```

In the precondition of `Title::setForSale()`, `Term2` contains a p-use of `self.reservation` and p-uses of attribute `state` of all `Reservation` objects. `Term3` contains a p-use of `self.copy` and p-uses of attribute `state` of all `Copy` objects.

Now we look at an example where an iterative collection operation appears in an edge within a compound node. In the following postcondition, `Term2` is associated with an edge (since it is on the left part of an `implies` expression). So it contains a p-use of `self.title.reservation` and p-uses of attribute `state` of all `Reservation` objects.

```
Copy::makeAvailable()
Pre: self.state = #OnHold                                Term1
Post: self.title.reservation@pre-> select(state=#Pending) ->isEmpty Term2
    implies
    self.state = #ForRent                                Term3
```

4.3.8.2.2 Iterative collection operations in nodes

When an iterative collection operation appears in a node, and is applied on the pre-value of a collection, the collection and the model elements involved in the parameter of the collection operation (i.e., `OclExpression`) are c-used provided that the term in which the collection operation occurs is not a frame rule. Note that after-values of attributes/links may be referred to in `OclExpression`; they are considered as c-uses as they are typically used to define another collection. For instance, in the following postcondition, the after-value of `att1` in the parameter of `self.roleName@pre` is used.

```
Post:
self.roleName->select(att1 = param)->size
  = self.roleName@pre->select(att1 = param)->size + 1
```

So we can't simply apply Rule 3, which is for pre-values only. Rule 7 is defined for this particular situation.

Rule 7 Pre-values of collections in iterative collection operations in nodes

```
context BasicNode
let op:Set(Iterative) = self.term.operation->
    select(o:Operation|o.oclIsTypeOf(Iterative)
        and o.collection.kind = #PreValue)

self.term.isFrameRule = false
implies (
op.collection.dataFlow->forAll(d:DataFlow|d = #Cuse)
and
op.involvedInOpParam.dataFlow->forAll(d:DataFlow|d = #Cuse)
)
```

When an iterative collection operation is applied on the after-value of a collection, the collection itself and the attributes/links referred to in `OclExpression` are considered as definitions provided that the term in which the operation occurs is not a frame rule. This is because iterative collection operations are typically used in postconditions to specify

the change to a collection and the creation of objects (through defining the attributes and links of the objects). Below is an example:

```

Title::setForSale()
pre: self.state=@ForRent                                Term1
    and not Self.reservation->exists
        (r:Reservation|r.state = #Pending
        or r.state =#Outstanding)                        Term2
    and not self.copy->exists
        (c:copy|c.state = #OnHold or c.state = #Rented)   Term3
Post: state = #ForSale                                    Term4
    and self.copy->forAll(c: Copy|c.state = #ForSale)     Term5

```

In the postcondition for `Title::setForSale()`, `Term5` is not a frame rule. Hence `Term5` contains definitions of `self.copy` and attribute state of all `Copy` objects.

Note that pre-values (e.g., an in parameter) may also appear in `OclExpression` and they are c-used.

Rule 8 After-values of collections in iterative collection operations in nodes

```

context BasicNode
let op:Set(Iterative) = self.term.operation->
    select(o:Operation|o.ocIsTypeOf(Iterative)
        and o.collection.kind = #AfterValue)

self.term.isFrameRule = false
implies (
op.collection.dataFlow->forAll(d:DataFlow|d = #Def)
and
op.involvedInOpParam->select(v:Value|kind = #AfterValue).dataFlow->
    forAll(d:DataFlow|d = #Def)
and
op.involvedInOpParam->select(v:Value|kind = #PreValue).dataFlow->
    forAll(d:DataFlow|d = #CUse)
)

```

4.3.9 Summary of rules for identifying definitions and uses

We have defined in the previous sections an exhaustive set of rules for identifying definitions and uses. Table 7 summarizes these rules, according to where a model element occurs (i.e., in an edge or a node), the operation type a model element is

associated with, and the kind of value that occurs. Note that Rule 2, which deals with model elements in frame rules, is not included in Table 7.

Where	Operation type	Value	Data Flow
Edge (Rule 1)	Any	Any	P-use
Node (Rule 3)	Any	Pre-value	C-use
Node (Rule 4)	Basic type and collections in type (2) non-iterative collection operations	After-value	Def or c-use
Node (Rule 5)	Parameters of non-iterative collection operations	After-value	C-use
Node (Rule 6)	Collections of type (1) non-iterative collection operations	After-value	C-use
Node (Rule 7)	Parameters in pre-values of iterative collection operations	After-value	C-use
Node (Rule 8)	Collection and parameters in after-values of iterative collection operations	After-value	Def

Table 7 Summary of rules

4.4 Identifying definition use pairs and definition clear paths in EAFG

A definition clear path (def clear path) with respect to (w.r.t.) a model element e is a path in a flow graph (in our case the EAFG) that it starts at a node where e is defined and ends at a node where e is used and e is not redefined on the path. A def clear path w.r.t. e is denoted by $(node_0, node_1, \dots, node_n)$. A definition use pair (du pair) with respect to a model element e is represented by a triplet (e, d, u) where d is a node that defines e , u is a node that uses e , and there is at least one def clear path from d to u .

Note that the number of def clear paths is always equal to or larger than the number of du pairs since a du pair may be traversed by at least one def clear path. As shown in the following example (Figure 30), the number of du pairs w.r.t. a is one, i.e., (a, n_0, n_2) , whereas the number of def clear paths w.r.t. a is two, i.e. (n_0, n_2) and (n_0, n_1, n_2) , both of which traverse the du pair (a, n_0, n_2) .

Figure 30 Du pairs vs. def clear paths

We now briefly describe the algorithms used to derive def clear paths and du pairs in an EAFG. Detailed algorithms are presented in Appendix B.

Our approach of deriving du pairs in EAFGs is to first obtain def clear paths in an EAFG, using a set of well-established algorithms proposed in the literature for compiler optimization [3]. The second step is to derive du pairs from def clear paths.

To derive def clear paths in an EAFG, the overall approach is to convert an EAFG into Single Statement Assignment Form [3]. In data flow analysis, the *Single Statement Assignment Form (SSA form)* is an intermediate representation in which each variable has only one definition in the program. As we apply it to an EAFG, an EAFG in SSA form has only one definition for a given model element. That is, each new definition of a model element, say a , is modified such that a fresh new model element, a_i , is defined and each use of the variable is modified to use the most recently defined version. For instance, in Figure 31, the EAFG fragment in (a) can be transformed into the SSA form in (b).

Note that an EAFG may have join nodes. A join node is a node that has multiple predecessors. When the use of a variable at the join node can be reached by different definitions, we can't decide which the newly defined version of the element is. For example in Figure 32, n_3 has two predecessors n_1 and n_2 , both of which define variable a . So, the version of a used in n_3 can be either of them.

Figure 31 (a) Excerpt of an EAFG (b) EAFG in SSA

Figure 32 Example of join nodes

To solve this problem, a notational fiction called ϕ -function is inserted [3]. Figure 33 shows that we can combine a_1 (defined in n_1) and a_2 (defined in n_2) using the function

$a_3 \leftarrow \phi(a_1, a_2)$. It yields a_1 if control reaches n_3 along the edge $n_1 \rightarrow n_3$, and yields a_2 if control comes in along the edge $n_2 \rightarrow n_3$.

Figure 33 Example of a join node with a phi function

But, where should ϕ -functions be inserted? We could add a ϕ -function for model elements at each join node, but this is not necessary for those model elements used at the join node but reached by the same definition. An efficient algorithm for placing ϕ -functions is to use the dominator tree (Appendix B.1) of a flow graph [10]. After ϕ -functions are inserted, all the definitions and uses are renamed using subscripts. Def clear paths can then be derived by simply traversing the dominator tree. More specifically, deriving def clear paths includes four steps: (1) The dominator tree is constructed from the EAFG; (2) The dominance frontier mapping (Appendix B.2) is constructed from the EAFG and its dominator tree; (3) Using the dominance frontiers, the location of the ϕ -function for each variable in the EAFG is determined; (4) The variables are renamed by replacing each occurrence of an original variable v with an appropriate mention of a new variable v_i . Def clear paths are derived as we traverse the dominator tree and rename the different definitions.

By applying these algorithms, def clear paths in an EAFG are derived. This complete set of def clear paths may contain duplicate du pairs since the same du pair may be traversed

by multiple def clear paths, as discussed earlier in this section. The algorithm for generating du pairs from def clear paths is described in Appendix B. It simply identifies du pairs from the set of def clear paths and removes duplicates.

However, some of the def clear paths may not be feasible. Recall (Section 4.1) that EAFGs may have infeasible paths due to incompatible sequences of transitions in statecharts. Also when EAFGs are modified to include the paths within a node due to OCL expressions (Section 4.2), not all paths would be evaluated to be true after the execution of a sequence of operations. Take the EAFG for Cruise Control statechart (Appendix E) as an example. Consider the `Running_accelerate_true` node. The contracts for event `accelerate` is in Figure 34, based on which the compound node for `Running_accelerate_true` in the modified EAFG can be created (Figure 35). Suppose the node `Running_accelerate_true` is reached from the node `Idle_engineOn_true`. Because the postcondition for `Idle_engineOn_true` is `self.brakepedal=0 and self.throttle=0 and self.speed=0 and self.state=#running`, only those terms along the highlighted path in Figure 35 would be evaluated to be true. Therefore, def clear paths located on other paths within this compound node are infeasible. A compound node with multiple predecessors may have several feasible paths. In this research, for each path in which a compound node occurs, the user input is required to provide the information on which path is taken within a compound node. Automatically identifying infeasible paths would require (possibly complex) semantic analysis of OCL expressions and this will be the subject of future research.

accelerate()	
Pre: self.state!= #idle	Term1
Post: if (self.brakepedal@pre > 0)	Term2
then self.brakepedal = 0	Term3
else true	Term4
endif	
and	
if(self.throttle@pre < CruiseController.maxThrottle - 5.0)	Term5
then self.throttle = self.throttle@pre + 5.0	Term6
else self.throttle = CruiseController.maxThrottle	Term7
endif	
and	
if(self.speed@pre < self.maxSpeed)	Term8
then	
if (((self.throttle -	
self.speed@pre/CruiseController.airRes)/CruiseController.speedRate)	
< CruiseController.maxSpeed-self.speed@pre)	Term9
then self.speed = self.speed@pre + (self.throttle -	
self.speed@pre/CruiseController.airRes)/	
CruiseController.speedRate	Term10
else self.speed = CruiseController.maxSpeed	Term11
endif	
else true	Term12
endif	
and	
if (self.state@pre = #cruising)	Term13
then self.state = #standby	Term14
else true	Term15
endif	

Figure 34 Pre and postcondition for event accelerate

Figure 35 Compound node for Running_accelerate_true

4.5 Limitations and possible solutions

As discussed in Section 4.2, the advantage of transforming an EAFG to a modified EAFG is to obtain more precise data flow information. There are, however, two limitations related to this approach. First, as mentioned in Section 4.2, the predicate parts of `if-then-else` and `implies` expressions are considered as one term which is associated with an edge. This is based on the assumption that the predicate parts are implemented as conditions and therefore only entail p-uses. However, the OCL syntax allows a designer to specify the change of a model element in the predicate part of an `if-then-else` or `implies` expression, thus suggesting a definition in that predicate. This can be done by embedding an `if-then-else` or an `implies` expression in the predicate. For instance,

in the following postcondition, b is defined in the predicate part of the if-then-else expression.

```

post:
if(a@pre>0  $\Rightarrow$  b=1)                                Term1
    then c = 2                                       Term2
    else true                                       Term3
endif

```

Though defining a variable in the predicate parts of if-then-else and implies expressions is not common, we have to account for such situations. One possible solution is to decompose the if-then-else or the implies expression embedded in the predicate part into basic nodes, thereby modeling the definition of a model element in the predicate part. Note that the edge, which is associated with the predicate part being false, does not need to be decomposed since it contains uses only. So the above example is represented by the compound node in Figure 36.

Figure 36 Example of decomposing the predicate part into basic nodes

Another limitation related to transforming an EAFG into a modified EAFG is, as mentioned in Section 4.2, that basic nodes within a compound node have an order, which

is the order of appearance in the OCL postcondition. This artificial order has an impact on the identification of du pairs and def clear paths (Section 4.4). For instance, the following postcondition of `op()` is represented in the compound node in Figure 37.

<code>Op</code>	
<code>post:</code>	
<code>(a = b@pre)</code>	Term1
<code>implies c = 2</code>	Term2
<code>and</code>	
<code>a = 1</code>	Term3

Figure 37 Compound node for `op()`

The p-use of the after-value of `a` at Term1 should be reached by the definition of the after-value of `a` at Term3. In Figure 37, however, Term1 occurs before Term3. This will lead to the after-value of `a` at Term1 reached by a definition of `a` in a compound node before this compound node.

One possible solution is to rearrange the order of basic nodes in a compound node such that the following two conditions are satisfied: (1) The appearance of a pre-value of a model element always precedes the appearance of an after-value; (2) The definition of a model element always occurs before the use of the after-value of the model element. The basic nodes in Figure 37 can then be rearranged as in Figure 38. This step of rearranging

basic nodes can be automated once the feasible paths within a compound node are determined with the help from the user (Section 4.4).

Figure 38 Compound node for `op()` after rearranging basic nodes

Recall (Section 4.3.8.2) that when identifying definitions and uses in terms with iterative collection operations, we consider not only the collection on which the operation is applied to be defined (or used), but also the attributes, links, and parameters (of the context operation) involved in the parameter (i.e., `OclExpression`) of the operation to be defined (or used). If an attribute `a` of class `C` appears in the parameter of an OCL collection operation, we consider that the attribute `a` of all the objects of class `C` is defined (or used). For instance, the following postcondition has a definition of `self.copy`, and definitions of attributes `barcode` and `state` of all `Copy` objects.

```

Title::AddCopy(barcode:String)
pre: not Copy.allInstances->exists(c:Copy|c.barcode = barcode)      Term1
Post: self.copy->exists
      (c:Copy|c.barcode = barcode and c.state = #ForRent)          Term2

```

Such an approach eases the automation since refining the analysis and characterizing the exact objects that are defined (or used) would require (likely very complex) semantic analysis of OCL expressions. Our approach guarantees that no definition (or use) be missed. But on the other hand, it may lead to the identification of definitions (or uses) that do not actually exist. This has an impact on the identification of def clear paths and

du pairs. For instance, in Figure 39, *a* is an object of class *A*, suppose that the definition of *a.att* at *n4* doesn't actually exist but is accounted for because attribute *att* of class *A* appears in the parameter of a collection operation which applies to a collection of *A* instances. As a consequence, du pair (*a.att*, *n4*, *n5*) will be identified whereas in fact du pair (*a.att*, *n1*, *n5*) should be.

Figure 39 Example of iterative collection operations

There is, however, no easy way to solve this problem. Future work will address this limitation.

4.6 Data Flow Analysis of Transition Trees

Recall that our objective is not to apply data flow criteria alone to UML statecharts. Rather, we aim to use data flow information to refine and improve existing state-based criteria and we are particularly interested in the transition tree criterion. This section describes how the approach proposed in previous sections can be applied to the transition tree criterion. Section 4.6.1 describes the step of transforming a transition tree to a

transition tree EAFG. Section 4.6.2 discusses the step of modifying a transition tree EAFG to represent the feasible path within a node, based on which definitions and uses are determined using the rules in Section 4.3 and du pairs are derived using the algorithms in Section 4.4.

4.6.1 Transforming transition trees into EAFGs

First, each transition tree is transformed into an EAFG using the approach in Section 4.1. The resulting EAFG H is also a tree, i.e., it is a connected acyclic graph. Besides, H is a subgraph of the EAFG G of the statechart from which the transition tree is derived. The node set of H equals the node set of G , i.e., $N(H) = N(G)$, and the edge set of H is a subset of the edge set of G , i.e., $E(H) \subseteq E(G)$. As an example, we present in Figure 40 one of the transition trees (Transition Tree 1) in the Cruise Control case study (Section 6.1) and its corresponding EAFG in Figure 41: To avoid cluttering the diagram, preconditions are not shown.

Figure 40 Transition Tree 1 (Cruise Control)

i_eOn: idle_engineOn_true
 r_eOff: running_engineOff_true
 r_ac: running_ac_true
 r_br: running_brake_true
 c_br: cruising_brake_true
 c_on: cruising_on_true
 c_ac: cruising_accelerate_true
 c_eOff: cruising_engineOff_true
 c_off: cruising_off_true
 s_on: standby_on_true
 s_ac: standby_accelerate_true
 s_br: standby_brake_true
 s_eOff: standby_engineOff_true
 s_re: standby_resume_true

Figure 41 EAFG for Transition Tree 1 (Cruise Control)

4.6.2 Modified transition tree EAFGs

Recall that once a statechart is transformed into an EAFG, the EAFG is modified to include the branches within a node (Section 4.2). Each node in a modified transition tree EAFG contains only the feasible path and in fact there's only one feasible path within each node of a transition tree EAFG. As mentioned in Section 4.4, determining the feasible path within a compound node requires the user input.

The modified transition tree EAFG for Cruise Control Transition Tree 1 (Figure 41) is presented in Figure 42. Others are presented in Appendix F. Note that the term number is used to label the basic node that contains the term. To avoid cluttering the diagram, terms associated with edges are not shown.

Figure 42 Modified EAFG for Transition Tree 1 (Cruise Control)

After the modified transition tree EAFGs are generated, definitions and uses are derived with the rules in Section 4.3. Then algorithms described in Section 4.4 are used to generate def clear paths and du pairs contained in the modified transition tree EAFGs.

CHAPTER 5 DATA FLOW INFORMATION ANALYZER

Data Flow Information Analyzer (DFIA) is developed to aid the data flow analysis of UML statecharts. This Chapter is organized as follows: Section 5.1 specifies the system requirements for DFIA, Section 5.2 describes the system analysis and design for DFIA, and Section 5.3 summarizes the current implementation.

5.1 Requirements for DFIA

The purpose of DFIA is to perform data flow analysis of UML statecharts. According to the methodology presented in Chapter 4, DFIA has the following functional requirements:

- Import and parse the files that define the UML statechart, the class diagram (e.g. to know whether a parameter is an in parameter), operation contracts, and transition trees generated from the UML statechart as well as helper files that provide information on infeasible paths.
- Transform the UML statechart into an EAFG.
- Modify the EAFG to include paths within a compound node.
- Identify definitions and uses from the modified EAFG.
- Derive def clear paths and du pairs from the modified EAFG.

Figure 43 Overview of DFIA

Figure 43 describes DFIA in terms of its main functionalities, inputs and outputs. The inputs to DFIA include the UML statechart specification file, the UML class specification file, the transition tree file, and operation contract file that defines the precondition and postcondition of each operation involved in the UML statechart. The outputs of DFIA are def clear paths and du pairs for the modified EAFG.

5.2 System architecture

DFIA is decomposed into four subsystems, namely: `control`, `EAFG`, `umlModel`, and `parser`, as shown in Figure 44.

Figure 44 Subsystem decomposition for DFIA

5.2.1 control subsystem

The `control` subsystem is decomposed into four subsystems, as shown in Figure 45. The `DFIAController` subsystem is responsible for the overall control flow of DFIA. The `EAFGBuilder` subsystem transforms a statechart or a transition tree into an EAFG or a transition tree EAFG. The `defUseIdentifier` subsystem implements the rules defined in Section 4.3. The `dataFlowGenerator` subsystem implements the `du` pair derivation algorithms described in Section 4.4.

Figure 45 control subsystem

5.2.2 umlModel subsystem

The `umlModel` subsystem is a UML metamodel adapted to the requirement of DFIA. This metamodel is based on the official UML metamodel [29]. As shown in Figure 46, it has two subsystems: `foundation` and `behaviourElements`. The `foundation` subsystem is the language infrastructure that specifies the static structure of models [29], which is further decomposed into `core` and `dataTypes` subsystems. The `core` subsystem defines the basic abstract and concrete metamodel constructs needed for the development of object models [29]. The `dataTypes` subsystem specifies the different data types that are used to define UML [29]. The `behaviourElements` subsystem is the language superstructure that specifies the dynamic behaviour of models [29]. It is further decomposed into `commonBehavior` and `stateMachines` subsystems. The `commonBehavior` specifies the core concepts required for dynamic elements [29]. The `stateMachines` subsystem defines behaviour using finite-state transition systems [29].

Figure 46 umlModel subsystem

5.2.3 EAFG subsystem

The EAFG subsystem contains the EAFG metamodel, which is shown in Figure 27 (Section 4.3.4).

5.2.4 parser subsystem

As shown in Figure 47, the parser subsystem is decomposed into three subsystems, namely, xmiParser, oclParser and textFileParser. xmiParser parses XMI files that describe the UML models, oclParser parses OCL expressions, and textFileParser parses the helper file with information on infeasible paths.

Figure 47 parser subsystem

5.3 Implementation

The current version of DFIA implements the `dataFlowGenerator`, `DFIAController` and `textFileParser` subsystems. The implementation consists of eleven classes and 1864 lines of java source code. Currently, the `EAFGBuilder` and the `defUseIdentifier` subsystems are not implemented: building modified EAFGs (transforming statecharts to EAFGs, then to modified EAFGs), and identifying definitions and uses are manually performed. Definitions and uses of model elements in nodes and edges of a modified EAFG are stored in a text file, which is parsed by `TextFileParser`. `dataFlowGenerator` then generates def clear paths and du pairs for the modified EAFG. A class diagram for the current implementation is presented in Appendix B.

CHAPTER 6 CASE STUDIES

We've selected two case studies where multiple transition trees can be generated by following Binder's algorithm [4]. The first case study is a cruise control system [6] that has three transition trees and the second case study is a VCR system that can generate twelve transition trees with Breadth First Search. As mentioned in Section 2.1.3, to determine the effectiveness of a transition tree, we seeded faults into the code of the two case studies, using the mutation operators provided in [16], [17], [19]. Our goal is to cover all the mutation operators that were applicable in the code under test and to seed the faults in a balanced way across operators given the characteristics of the code of each case study. To derive data flow information contained in a transition tree, we apply the approach proposed in Chapter 4: First a transition tree is transformed into an EAFG, then definitions and uses in each node are derived based on the rules discussed in Section 4.3, and the number of def clear paths and du pairs are derived using the algorithms described in Section 4.4.

The rest of this chapter is divided into three sections: Section 6.1 and Section 6.2 describe the results for the Cruise Control study and the VCR system respectively. Section 6.3 summarizes the results and discusses practical implications of the results.

6.1 Cruise Control

The Cruise Control system is implemented in Java and contains six classes, namely `CruiseControl`, `CruiseDisplay`, `CarSpeed`, `Controller`, `CarSimulator`, and `SpeedControl` (Appendix C). Among them `Controller`, `CarSimulator`, and `SpeedControl` form a class cluster whose behavior is described by a statechart. Such a class cluster can be, at analysis level, represented by one class `CruiseController` (Appendix D). The statechart describing the behavior of `CruiseController` is shown in Figure 48. Note that in this statechart, transitions have no guard conditions and events have no parameters.

6.1.1 Transition trees

Recall that according to Binder's Algorithm, the termination criterion for a transition tree is when the state is already drawn somewhere in the transition or is a final state. From the `CruiseController` statechart, once states `Idle`, `Running` and `Cruising` are traversed, states `Idle`, `Cruising`, and `Standby` are added as the children of the state `Cruising`. And in fact, the `Standby` state is added three times since there are three outgoing transitions from the `Cruising` state to the `Standby` state triggered by events `brake`, `accelerate` and `off` respectively. Because the `Idle` state and the `Cruising` state are already drawn in the tree, they are considered terminal. The `Standby` state is the new state in the transition tree and its three occurrences produce three different transition trees, namely `TT1` (the tree is further developed after the event `brake` as in Figure 49 (a)), `TT2` (the tree is further developed after event `accelerate` as in Figure 49 (b)), and `TT3` (the tree is further developed after event `off` as in Figure 49 (c)).

Recall that the three transition trees are equivalent in the sense that they all cover the round-trip paths in the statechart.

Figure 48 Statechart for CruiseController

Figure 49 Transition trees for Cruise Control

6.1.2 Mutation scores

Six mutation operators were used in the case study as an analysis of the code reveals that some mutation operators turned out not to be applicable and some operators caused compilation errors. The six mutation operators are Arithmetic Operator Replacement (AOR), Constant Replacement (CRP), Method Name Replacement (MNR), Relational Operator Replacement (ROR), Return Statement Replacement (RSR) and Statement Deletion (SDL). 91 faults were seeded into the three core classes of the system. The histogram of fault distributions across mutation operators is shown in Figure 50.

Figure 50 Mutants distributions across mutation operators (Cruise Control)

Important variations are observed in terms of fault detection ratio among the three possible transition trees. The mutation scores for TT1, TT2 and TT3 are 91%, 96% and 85% respectively (Section 2.1.3). The next section explains such differences using the data flow information contained in each transition tree EAFG.

6.1.3 Comparison of data flow information in each transition tree

Modified EAFGs for the three transition trees are derived, as shown in Appendix F. Based on the rules discussed in Section 4.3, definitions and uses in each node are obtained. Using the algorithms described in Section 4.4, information on the number of def clear paths and du pairs in each of the transition trees is obtained, as shown in Table 8. To ease discussion, the rest of the thesis uses EAFG1 to refer to the EAFG for TT1, EAFG2 for TT2, and EAFG3 for TT3.

EAFG	Def clear paths	Du pairs
EAFG1 (TT1)	81	81
EAFG2 (TT2)	88	88
EAFG3 (TT3)	81	81

Table 8 Def clear paths and du pairs in transition tree EAFGs (Cruise Control)

Table 8 shows that for each EAFG, the number of def clear paths is the same as the number of du pairs. This is explained by the fact that these EAFGs are in essence trees that have no cycles and there are no alternative branches in compound nodes as infeasible paths have been removed. So each du pair is traversed by one def clear path. Referring to the mutation scores in Section 6.1.2, the mutation score for TT2 is 5% higher than that for TT1, and TT1 7% higher than TT3. EAFG1 and EAFG3 have the same number of def clear paths/du pairs, and EAFG2 has seven more def clear paths/du pairs than EAFG1 and EAFG3. The relationship between mutation scores and numbers of def clear paths/du pairs is depicted in Figure 51.

Figure 51 Def clear paths/du pairs vs. mutation scores (Cruise Control)

We can see from Figure 51 that the mutation score of a transition tree is related to the number of def clear paths/du pairs it covers: TT2, which has the highest mutation score, covers the largest number of def clear paths/du pairs. For TT1 and TT3, however, they contain the same number of def clear paths/du pairs although the mutation score for TT1 is 7% higher than that for TT3. This result motivates us to further examine the specific sets of def clear paths/du pairs traversed by each transition tree and to investigate whether the differences between TT1 and TT3 in terms of the specific sets of def clear paths/du pairs covered can explain their differences in mutation scores.

Table 9 shows the number of du pairs w.r.t. each attribute. Note that the operation contracts for the Cruise Control system don't involve any parameters or links. EAFG1 and EAFG3 have exactly the same number of def clear paths/du pairs w.r.t. each attribute. For attribute `self.state` and `self.throttle`, all three EAFGs have the

same number of def clear paths/du pairs, whereas for the other attributes, EAFG1 and EAFG3 have the same number of def clear paths, which are smaller than that of EAFG2.

EAFG	EAFG1	EAFG2	EAFG3
self.state	23	23	23
self.speed	18	22	18
self.cruisespeed	10	11	10
self.throttle	18	18	18
self.brakepedal	12	14	12
Total	81	88	81

Table 9 Def clear paths/du pairs with respect to each attribute (Cruise Control)

Though two EAFGs may have the same number of def clear paths w.r.t. attributes, the sets of def clear paths are not identical. That is, for each attribute, there exist def clear paths/du pairs that are uniquely covered by only one EAFG. Table 10 lists for each attribute, the number of def clear paths that are traversed by all three EAFGs and the number of def clear paths that are uniquely traversed by each EAFG. There are no def clear paths covered by only two EAFGs.

Attribute	Common def clear paths	Uniquely traversed by		
		EAFG1	EAFG2	EAFG3
self.state	15	8	8	8
self.speed	12	6	10	6
self.cruisespeed	6	4	5	4
self.throttle	14	4	4	4
self.brakepedal	8	4	6	4

Table 10 Common and unique def clear paths (Cruise Control)

Table 10 shows that for instance there is a total number of 39 def clear paths w.r.t. `self.state` (15+8+8+8) in the three trees together, and that only 23 of them (15 + 8) are covered by each tree.

Note that two different def clear paths may traverse one du pair. For instance, Table 11 shows two def clear paths w.r.t. `self.throttle` traversed by EAFG1 and EAFG3 respectively. They traverse the same du pair w.r.t. `self.throttle` that is defined in node `Idle_engineOn`¹, and used in node `Standby_engineOff_6`, or (`self.throttle`, `Idle_engineOn`, `Standby_engineOff_6`) using the triplet defined in Section 4.4. Table 12 shows the common and unique du pairs covered by EAFG1 and EAFG3. Table 13 and Table 14 show the common and unique du pairs between EAFG1 and EAFG2, EAFG3 and EAFG2.

EAFG1	Idle_engineOn, Running_on_1, Running_on_2, Running_on_4, Running_on_6, Running_on_7, Cruising_brake_1, Cruising_brake_2, Cruising_brake_4, Cruising_brake_5, Cruising_brake_7, Cruising_brake_11, Cruising_brake_12, Standby_engineOff_1, Standby_engineOff_2, Standby_engineOff_4, Standby_engineOff_6
EAFG3	Idle_engineOn, Running_on_1, Running_on_2, Running_on_4, Running_on_6, Running_on_7, Cruising_off_1, Standby_engineOff_1, Standby_engineOff_2, Standby_engineOff_4, Standby_engineOff_6

Table 11 Two def clear paths w.r.t `self.throttle` in EAFG1 and EAFG3

We see that EAFG1 and EAFG3 differ in the sets of du pairs w.r.t. `self.state` and `self.brakepedal`, while all other sets of du pairs are identical. EAFG1 and EAFG3

¹ A node in the Cruise Control EAFG is labeled with `sourceState_eventName_true` since transitions in the statechart don't have guard conditions. For ease of presentation, the rest of this paper uses `sourceState_eventName` instead.

have less du pairs in common with EAFG2. The only difference between Table 13 and Table 14 is on the attribute `self.brakepedal`: all the du pairs w.r.t. `self.brakepedal` covered by EAFG3 are traversed by EAFG2.

Attribute	Common du pairs	Uniquely traversed by	
		EAFG1	EAFG3
<code>self.state</code>	15	8	8
<code>self.speed</code>	18	0	0
<code>self.cruisespeed</code>	10	0	0
<code>self.throttle</code>	18	0	0
<code>self.brakepedal</code>	8	4	4

Table 12 Common and unique du pairs between EAFG1 and EAFG3 (Cruise Control)

Attribute	Common du pairs	Uniquely traversed by	
		EAFG1	EAFG2
<code>self.state</code>	15	8	8
<code>self.speed</code>	12	6	10
<code>self.cruisespeed</code>	9	1	2
<code>self.throttle</code>	14	4	4
<code>self.brakepedal</code>	8	4	6

Table 13 Common and unique du pairs between EAFG1 and EAFG2 (Cruise Control)

Attribute	Common du pairs	Uniquely traversed by	
		EAFG3	EAFG2
<code>self.state</code>	15	8	8
<code>self.speed</code>	12	6	10
<code>self.cruisespeed</code>	9	1	2
<code>self.throttle</code>	14	4	4
<code>self.brakepedal</code>	12	0	2

Table 14 Common and unique du pairs between EAFG3 and EAFG2 (Cruise Control)

In conclusion, though EAFG1 and EAFG3 cover the same number of def clear paths and du pairs (Table 8), the sets of def clear paths covered are different (Table 10) and the sets of du pairs covered are different too (Table 12).

Next, we look at the live mutants missed by the transition trees, to find out whether those differences explain the different mutation scores in Figure 51, i.e., whether covering certain du pairs guarantees that certain mutants be detected. As Table 15 shows, TT1 misses seven mutants that are killed by TT2. TT2 misses three mutants that are killed by TT1. TT3 misses all the mutants that are missed by TT1 and TT2, plus two more mutants that are killed by TT1 and TT2.

Transition Tree	Live Mutant#
TT1	8, 16, 24, 25, 26, 59, 69
TT2	63, 64, 71
TT3	4, 8, 16, 22, 24, 25, 26, 59, 63, 64, 69, 71

Table 15 Live mutants with transition trees (Cruise Control)

We first look at the three mutants missed by both TT2 and TT3 but killed by TT1 (Mutant 63, 64 and 71). They are related to the variable `brakepedal` in class `CarSimulator`. For these three mutants to be detected, `brakepedal` needs to be

changed to a value other than zero before event `accelerate` or `engineOff` is called. The four du pairs that are uniquely traversed by EAFG1 define `self.brakepedal` at `Cruising_brake_5` where the value of `self.brakepedal` is increased by one, and use `self.brakepedal` at `Standby_engineOff_8`, `Standby_brake_4`, `Standby_brake_5`, and `Standby_accelerate_2`. This means that TT1 includes test cases that increase `brakepedal` to a value larger than zero before event `accelerate` or `engineOff`. Hence, TT1 kills these three mutants thanks to these four covered du pairs. On the other hand, all the du pairs w.r.t. `self.brakepedal` in EAFG3 define `self.brakepedal` at `Idle_engineOn` where the value of `self.brakepedal` is set to zero. This explains why TT3 fails to detect these mutants. EAFG2 covers two du pairs in addition to those covered by EAFG3, w.r.t. `self.brakepedal`. They are defined and used within the same compound node `Standby_brake`. Though `self.brakepedal` is defined to be larger than zero, it is not used in `accelerate` nodes, or `engineOff` nodes since `Standby_brake` is the terminal compound node of a path in EAFG2, thus explaining why TT2 does not kill these three mutants.

Next we take a look at the mutants killed by TT2 but missed by TT1 and TT3. Mutant 8 and 16 are related to the variable `throttle` in class `CarSimulator`. For these mutants to be detected, the value of `throttle` must be equal to a value other than zero before event `brake` or event `on` is called. All the du pairs w.r.t. `self.throttle` covered by EAFG1 and EAFG3 define `self.throttle` to be zero at `Idle_engineOn`. That is to say, TT1 and TT3 don't contain any path where `throttle` is set to a value larger than zero and subsequently used. EAFG2, however, in addition to the du pairs covered by EAFG1 and EAFG3, has four more du pairs w.r.t `self.throttle`, where `self.throttle` is

redefined to be larger than zero at node `Cruising_accelerate`. This means that TT2 has test cases that set the value of throttle to be larger than zero and subsequently used this definition.

Mutants 24 to 26 were seeded into the `run` method that implements the speed control function in class `SpeedControl` when cruise control is turned on. For these three mutants to be detected, the speed of the car has to be changed to a value other than zero before the method `on` gets executed. EAFG1 and EAFG3 have equivalent sets of du pairs w.r.t. `self.speed`, which define `self.speed` to be zero in node `Idle_engineOn`. This means that TT1 and TT3 don't contain any test cases that use car speed with a value larger than zero. On the other hand, TT2 have 11 du pairs which define `self.speed` at `Cruising_accelerate_9`, where `self.speed` is increased to a value larger than zero. Therefore, only TT2 is able to kill these mutants. Mutant 59 and 69 were seeded into class `Controller` and class `carSimulator`. They can be detected if the value of the speed is greater than zero before event `engineOff` is executed. As in the case of Mutant 24 to 26, only EAFG2 has du pairs w.r.t. `self.speed` where `self.speed` is assigned to be a value greater than zero. Hence, only TT2 is able to kill these mutants.

Lastly, Mutant 4 and 22 are live with TT3 but killed by both TT1 and TT2. Mutant 4 was seeded into class `CarSimulator` and Mutant 22 was seeded into class `SpeedControl`. These mutants can only be detected if either `self.brakepedal` or `self.speed` is set to a non-zero value before event `engineOff`, `on` or `resume`. As discussed previously, this is the case with TT1 and TT2. However, the du pairs for `self.brakepedal` or

`self.speed` in EAFG3 all have `self.brakepedal` and `self.speed` defined to be zero. Hence Mutant 4 and 22 were not killed by TT3.

From the above analysis of the mutants missed by the transition trees, we can observe that covering certain du pairs guarantees that certain mutants be detected. This observation not only confirms the relationship between mutation scores and the number of du pairs, but also suggests that different sets of du pairs may lead to different mutation scores.

As the live mutants for the transition trees seem to be related to the definitions of attributes, we report the number of definitions in each EAFG: EAFG1 has 25 definitions, EAFG2 has 30 and EAFG3 23. The relationship between mutation scores and numbers of definitions is depicted in Figure 52. We can see from Figure 52 that numbers of definitions exhibit a near linear relationship to the mutation scores, that is, the more definitions a transition tree has, the higher its mutation score will be. This result suggests that the number of definitions covered by a transition tree EAFG is a good indicator of its fault detection power. When multiple transition trees can be generated from one statechart, choosing the one which has the largest number of definitions guarantees the highest effectiveness.

Figure 52 Definitions vs. mutation scores (Cruise Control)

With the purpose of investigating how the definitions covered by one EAFG but not covered by another may contribute to mutant detection, we determine the number of definitions w.r.t each attribute, as shown in Table 16.

EAFG	EAFG1	EAFG2	EAFG3
self.state	10	10	10
self.speed	4	6	4
self.cruisespeed	1	2	1
self.throttle	4	8	4
self.brakepedal	6	4	4
Total	25	30	23

Table 16 Number of definitions with respect to each attribute (Cruise Control)

A closer look at these sets of definitions reveals that with all other definitions being the same, EAFG1 has two more definitions w.r.t self.brakepedal and EAFG2 has two more definitions w.r.t self.speed, one more definition w.r.t self.cruiseSpeed, and 4 more definitions w.r.t self.throttle. And all these definitions occur on paths

which has du pairs that guarantee certain mutants to be killed. For example, TT1 has 2 more definitions for `self.brakepedal`: `Standby_acceleate_3` and `Standby_engineOff_9`. The first definition of `self.brakepedal` occurs on the path that has the du pair `(self.brakepedal, Cruising_brake_5, Standby_accelerate_2)`. And the second definition of `self.brakepedal` occurs on the path that has the du pair `(self.brakepedal, Cruising_brake_5, Standby_engineOff8)`. As we've already seen in previous discussion, traversing these two du pairs guarantees the mutants related to the variable `brakepedal` be killed.

As discussed previously, the set of du pairs exercised by a transition tree seem to contribute to mutant detection, i.e., covering certain du pairs guarantees that certain mutants be detected. Since each of the extra definitions in EAFG1 and EAFG2 occur along the same path as a du pair which leads to detecting certain mutants, we can draw the conclusion that covering certain definitions also guarantees that certain mutants be detected.

6.1.4 Summary of results

The results of the Cruise Control case study suggest that both du pairs and definitions are good indicators of the fault detection power of a transition tree. The transition tree with the highest mutation score has the largest number of both du pairs and definitions. The analysis of live mutants shows that covering certain du pairs or definitions guarantees some mutants be detected.

The result that definitions can be used to determine the fault detection effectiveness of a transition tree has important practical implications as identifying definitions is less costly

than identifying du pairs: First identifying definitions requires that only after-values in nodes in an EAFG be considered since edges and pre-values in nodes don't contain any definitions (Section 4.3). Consequently less complex and fewer rules will be needed. This would certainly ease the automation of these rules. Second, definitions can be easily collected by simply traversing the graph whereas deriving du pairs from an EAFG needs sophisticated algorithms (Section 4.4).

6.2 VCR

The VCR system is implemented in Java using the state design pattern. It contains 29 classes, among which fourteen are state classes², and eleven are event classes³. The remaining classes are `Vcr`, `Tape`, `TimedTask`, and `VcrDriver`. At the analysis level, one single class `VcrController` can be used to model the state-dependent behavior of VCR. The class diagram at analysis level is shown in Appendix G, and operation contracts are provided in Appendix H. The statechart of the VCR system at analysis level is presented in Figure 53. Note that the statechart is hierarchical and its flattened equivalent is not shown since it's too large and complex.

The statechart model (Figure 53) and contracts at analysis level (Appendix H) identify states `Rewinding` and `Forwarding` without further refining the states into states `FastFF`, `SlowFF`, `FastRew` and `SlowRew`, while the implementation does. However, the attribute `isTapePulledToDrum` in the class diagram at analysis level models the speed

² The state classes are `VcrState`, `Off`, `TapeAbsent`, `TapePresentMode`, `Paused`, `RecPaused`, `Stopped`, `FastFF`, `FastRew`, `Playing`, `TapePositionAlteringState`, `Recoring`, `SlowFF`, and `SlowRew`

³ The event classes are `VcrEvent`, `insertingTape`, `EjectButton`, `FFButton`, `PauseButton`, `PlayButton`, `PowerButton`, `RecButton`, `RewButton`, `StopButton`, and `EndOfTapeEvent`.

of forwarding/rewinding: it is set to true if the speed of forwarding/rewinding is slow, false if the speed of forwarding/rewinding is fast. When the VCR is in states Playing, Recording, Paused, or RecPaused, `isTapePulledToDrum` is set to true. When the VCR is in the state Stopped, it is set to false. When a transition is triggered by `FFButton/RewButton`, the VCR is fast forwarding/rewinding if the source state has `isTapePulledToDrum` set to false, slow forwarding/rewinding if the source state has `isTapePulledToDrum` set to true. When a transition is triggered by `EndOfTapeEvent`, the VCR is fast rewinding if the source state is Playing, Recording, or Forwarding. This information is detailed in the operation contracts (Appendix H).

Figure 53 Statechart for VcrController

6.2.1 Transition trees

Recall that the termination criterion for a transition tree is when a state is already drawn somewhere in the transition or is a final state. From the state `Off`, the breadth first search adds states `Playing`, `TapeAbsent`, `Off`, and `Stopped` (twice because of transitions `PowerButton[self.tape<>null]` and `insertingTape[self.tape.isProtected()=false]`). Among these states, only the state `Off` becomes a leaf since it's already drawn in the tree. Because the state `Stopped` appears twice, transition trees can be further developed after either of the

events leading to Stopped mentioned above. Let's say transition trees are developed after event `PowerButton[self.tape<>null]`. Among the children of states Playing, TapeAbsent, and Stopped, the state Rewinding are added three times (twice as the children of the state Playing after events `RewButton` and `EndOfTapeEvent`, and one time as the child of the state Stopped after event `RewButton`), and the state Forwarding is added twice (as the child of the state Stopped and Playing after event `FFButton`). This yields six possible combinations. A transition tree can be developed along a path formed by any of these combinations of nodes. Similarly, six transitions can be developed along the state Stopped after event `insertingTape[self.tape.isProtected()==false]`. This leads to twelve possible transition trees listed in Appendix I.

6.2.2 Mutation scores

After analyzing the code, six applicable mutation operators were used in this case study. In addition to the AOR, ROR, SDL, and CRP operators used in the Cruise Control case study, Class Instance Creation Expression Changes (ICE), and Overriding Method Removal (OMR) were also applied. A total of 131 mutants were seeded into the system. The histogram of fault distributions across mutation operators is shown in Figure 54.

Mutation scores for the transition trees are listed in Table 17. Note that some of the transition trees have the same mutation scores. When looking at live mutants for each transition tree, it appears that some of the transition trees have exactly the same set of live mutants, as shown in Table 18.

Figure 54 Mutants distributions across mutation operators (VCR)

Transition tree	1	2	3	4	5	6	7	8	9	10	11	12
Mutation score (%)	76	74	74	71	74	71	76	74	74	71	74	71

Table 17 Mutation scores for transition trees (VCR)

Group	Transition tree	Mutation score (%)	Live mutants
1	TT1, TT7	76	3, 4, 5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 104, 105, 106, 107, 108, 109
2	TT2,TT5, TT8, TT11	74	3, 4, 5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 104, 105, 106, 127, 128, 129, 130, 141
3	TT3, TT9	74	20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 68, 69, 70, 72, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 107, 108, 109, 123, 124, 125, 126, 140
4	TT4,TT6, TT10, TT12	71	68, 69, 70, 72, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 123, 124, 125, 126, 127, 128, 129, 130, 140, 141

Table 18 Live mutants with transition trees (VCR)

We can see from Table 18 that Group 1 and Group 4 are complementary in the sense that Group 1 kills all the mutants that are missed by Group 4 whereas Group 4 kills all the mutants that are missed by Group 1. Similarly, Group 2 is complementary to Group 3. The next section discusses how data flow information is related to the mutation scores.

6.2.3 Comparison of data flow information in each transition tree

Once transition trees are transformed into EAFGs, definitions and uses in each node and edge are determined based on the rules in Section 4.3. Using the algorithms described in Section 4.4, information on the number of def clear paths and du pairs in each of the transition trees is obtained. Similar to the Cruise Control case study, for each EAFG transition tree, the number of def clear paths equals the number of du pairs. Table 19 lists the def clear paths/du pairs information for each EAFG.

EAFG	1	2	3	4	5	6	7	8	9	10	11	12
Du pairs	188	185	185	182	185	182	188	185	185	182	185	182

Table 19 Def clear paths and du pairs in transition tree EAFGs (VCR)

We can see from Table 19 that some EAFGs have the same number of def clear paths/du pairs. In fact, transition trees with the same mutation scores have the same number of def clear paths/du pairs. In other words, the transition trees within one group have exactly the same number of def clear paths/du pairs. This is shown in Table 20. The relationship between mutation scores and def clear paths for each group is depicted in Figure 55.

Group	Def clear paths	Mutation scores (%)
1	188	76
2	185	74
3	185	74
4	182	71

Table 20 Mutation scores and def clear paths/du pairs for each group (VCR)

Figure 55 Def clear paths/du pairs vs. mutation scores (VCR)

Figure 55 shows a linear relationship between the number of def clear paths/du pairs and mutation scores. Since the first case study suggests a relationship between the covered du pairs and mutation scores, we look at the number of du pairs w.r.t. each model element in the VCR system for the twelve EAFGs (Table 21).

Group	1		2				3		4			
EAFG	1	7	2	5	8	11	3	9	4	6	10	12
numTimesTapePulled	25	25	22	22	22	22	22	22	19	19	19	19
isTapePulledToDrum	12	12	12	12	12	12	12	12	12	12	12	12
isTapePresent	14	14	14	14	14	14	14	14	14	14	14	14
eventStartTime	40	40	40	40	40	40	40	40	40	40	40	40
state	48	48	48	48	48	48	48	48	48	48	48	48
tape.position	46	46	46	46	46	46	46	46	46	46	46	46
tape.isWriteProtected	3	3	3	3	3	3	3	3	3	3	3	3

Table 21 Def clear paths/du pairs w.r.t. each model element (VCR)

We can see from Table 21 that the twelve EAFGs have exactly the same number of def clear paths/du pairs w.r.t. every attribute except the attribute numTimesTapePulled.

We then look at the specific sets of def clear paths/du pairs covered. The results show that for EAFGs within the same group, their sets of def clear paths/du pairs are not identical, but they tend to have more common def clear paths and du pairs than EAFGs from different groups. Take EAFG1, EAFG7 and EAFG4 as an example. EAFG1 and EAFG7 belong to Group 1, whereas EAFG1 and EAFG4 are from different groups. Over 87% def clear paths and du pairs traversed by EAFG1 are also traversed by EAFG7 whereas EAFG1 only has 59% of its def clear paths and du pairs in common with EAFG4.

Now, we look at the live mutants missed by at least one group of EAFGs and try to find out whether this can be explained by the absence of certain du pairs. Table 22 shows the live mutants are located in classes `FastFF`, `FastRew`, `SlowFF`, and `SlowRew`. Group 1 missed the mutants in classes `FastFF` and `FastRew`, Group 2 missed the mutants in classes `FastFF` and `SlowRew`, Group 3 missed the mutants in classes `FastRew` and `SlowFF`, and Group 4 missed the mutants in classes `SlowFF` and `SlowRew`.

Group	FastFF.java	FastRew.java	SlowFF.java	SlowRew.java
1	3, 4, 5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 104, 105, 106	20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 107, 108, 109		
2	3, 4, 5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 104, 105, 106			86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 127, 128, 129, 130, 141
3		20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 107, 108, 109	68, 69, 70, 72, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 123, 124, 125, 126, 140	
4			68, 69, 70, 72, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 123, 124, 125, 126, 140	86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 127, 128, 129, 130, 141

Table 22 Live mutants in various classes (VCR)

To kill the live mutants in class `FastFF` (Mutants 3 to 17, 104, 105, and 106), VCR needs to be fast forwarding before `EndOfTapeEvent`, `RewButton` and `PlayButton` are executed. That is, `isTapePulledToDrum` needs to be set to false before event `Forwarding_EndOfTapeEvent`, `Forwarding_Rew`, and `Forwarding_Play` gets executed. All transition trees in Group 3 and Group 4 contain du pairs that define `isTapePulledToDrum` to be false at `Off_power2_4`⁴ (EAFG3, EAFG4, EAFG6) or `Off_insertingTape2_9` (EAFG9, EAFG10, EAFG12) when the VCR enters state `Stopped` and use it in `Forwarding_EndOfTapeEvent`, `Forwarding_Rew`, or `Forwarding_Play`. This means that transition trees in Group 3 and 4 have test cases that put the VCR in a fast forwarding state before `EndOfTapeEvent`, `RewButton` and `PlayButton` are executed, and are thus able to detect these mutants. On the other hand, all the du pairs w.r.t. `isTapePulledToDrum` in Group 1 set `isTapePulledToDrum` to be true at `Off_insertingTape1_8` when the VCR enters state `Playing`. Though some of the du pairs w.r.t. `isTapePulledToDrum` in Group 2 define `isTapePulledToDrum` to be false, the uses don't occur in `Forward_EndOfTapeEvent`, `Forward_Rew`, or `Forward_Play`. So transition trees in Group 1 and Group 2 miss these mutants.

To kill the live mutants in class `FastRew` (Mutants 20 to 32, 107, 108 and 109), VCR needs to be in a `FastRew` state before `endOfTapeEvent`, `FFButton`, and

⁴ For ease of presentation, `off_power1` is used for `off_power_self.tape=null`, `off_power2` is used for `off_power_self.tape<>null`, `off_insertingTape1` for `off_insertingTape_self.tape.isProtected=true`, and `off_insertingTape2` for `off_insertingTape_self.tape.isProtected=false`. For those events without guard conditions, `sourceState_eventName` is used.

playButton get executed. So isTapePulledToDrum has to be set to false before event Rewinding_EndOfTapeEvent, Rewinding_Rew, and Rewinding_Play. Group 2 and 4 kill these mutants since they have du pairs w.r.t isTapePulledToDrum that define isTapePulledToDrum to be false at Playing_endOfTape_3 (EAFG5, EAFG12) when the VCR is fasting forwarding, Off_power2_4 (EAFG2, EAFG4, EAFG6) or Off_insertingTape2_9 (EAFG8, EAFG11) when the VCR enters state Stopped, and use it at Rewinding_EndOfTapeEvent, Rewinding_FF, or Rewinding_Play. Group 1 and Group 3 fail to detect these mutants since Group 1 only contain du pairs w.r.t. isTapePulledToDrum where isTapePulledToDrum is set to true and Group 3, though has du pairs that define isTapePulledToDrum to be false, doesn't use it in Rewinding_EndOfTapeEvent, Rewinding_FF, or Rewinding_Play.

To kill the live mutants in class SlowFF (Mutants 68 to 83, 123 to 126, and 140), the VCR needs to be in a SlowFF state before endOfTapeEvent, RewButton, and playButton are executed. So isTapePulledToDrum has to be set to true before Forwarding_endOfTapeEvent, Forwarding_rew, and Forwarding_play. Group 1 and Group 2 are able to kill these mutants since they have du pairs that define isTapePulledToDrum to be true at Off_insertingTape1_8 (EAFG1, EAFG7, EAFG2, EAFG5, EAFG8, EAFG11) when the VCR is entering state Playing and use it at Forwarding_endOfTapeEvent, Forwarding_rew, and Forwarding_play. However, in Group 4 the du pairs w.r.t. isTapePulledToDrum all define isTapePulledToDrum to be false. Group 3 has du pairs where isTapePulledToDrum

is defined to be true, but it is not used in `Forwarding_endOfTapeEvent`, `Forwarding_rew`, or `Forwarding_play`.

Lastly, to kill the live mutants in class `SlowRew` (Mutants 86 to 99, 127 to 130, and 141) VCR needs to be in a `SlowRew` state before `endOfTapeEvent`, `FFButton`, and `playButton` are executed. So `isTapePulledToDrum` has to be set to true before `Rewinding_endOfTapeEvent`, `Rewinding_FF`, and `Rewinding_play`. Group 1 and Group 3 contain du pairs that define `isTapePulledToDrum` to be true at `Off_insertingTape1_8` (EAFG1, EAFG7, EAFG3, EAFG9) when the VCR is entering state `Playing` and use it at `Rewinding_endOfTapeEvent`, `Rewinding_FF`, and `Rewinding_play`. In Group 4, as discussed previously, the du pairs w.r.t. `isTapePulledToDrum` all define `isTapePulledToDrum` to be false. Some of the du pairs w.r.t. `isTapePulledToDrum` in Group 2 define `isTapePulledToDrum` to be true, but use it at `Forwarding` nodes rather than `Rewinding` nodes. So transition trees in Group 4 and Group 2 miss these mutants.

From the above analysis of live mutants, we can conclude that traversing certain du pairs ensures that certain mutants be detected. This confirms the result in the Cruise Control case study that mutation scores are related to not only the number of du pairs but also the specific sets of du pairs.

Since definitions also appear to be a good indicator of fault detection effectiveness in the Cruise Control case study, we look at the number of definitions in each EAFG, as shown in Table 23.

EAFG	1	2	3	4	5	6	7	8	9	10	11	12
Definitions	166	160	160	154	160	154	166	160	160	154	160	154

Table 23 Number of definitions for EAFGs (VCR)

As in the case of def clear paths/du pairs, EAFGs within the same group have the same number of definitions (Table 24). The relationship between the number of definitions and mutation scores is depicted in Figure 56.

Group	Definitions	Mutation Scores
1	166	76
2	160	74
3	154	74
4	154	71

Table 24 Mutation score and number of definitions for each group (VCR)

Figure 56 Definitions vs. mutation scores (VCR)

Figure 56 confirms the result in the Cruise Control case study that the number of definitions has a linear relationship to mutation scores. We further investigate the specific

sets of definitions in each EAFG. Table 25 shows that EAFGs within the same group have the same number of definitions w.r.t. each model element. And in fact, EAFGs within the same group have an identical set of definitions.

Group	1		2				3		4			
EAFG	1	7	2	5	8	11	3	9	4	6	10	12
numTimesTapePulled	26	26	23	23	23	23	23	23	20	20	20	20
isTapePulledToDrum	29	29	26	26	26	26	26	26	23	23	23	23
isTapePresent	14	14	14	14	14	14	14	14	14	14	14	14
eventStartTime	19	19	19	19	19	19	19	19	19	19	19	19
state	48	48	48	48	48	48	48	48	48	48	48	48
tape.position	25	25	25	25	25	25	25	25	25	25	25	25
tape.isWriteProtected	5	5	5	5	5	5	5	5	5	5	5	5

Table 25 Definitions w.r.t. each variable (VCR)

We compare the sets of definitions w.r.t each model element among the four groups. The results show that many of the definitions that are present in one group, say Group A, but absent in another group, say Group B, belong to those paths that have du pairs that guarantee some mutants to be detected by Group A but missed by Group B.

6.2.4 Summary of results

In the VCR case study, twelve transition trees are generated by following Binder's algorithm using Breadth First Search. Based on mutation scores and the specific set of missed mutants, the twelve transition trees can be divided into four groups. EAFGs within one group are found to contain the same number of du pairs/def clear paths and an identical set of definitions.

Both the number of du pairs and the number of definitions show a linear relationship to mutation scores. In line with the results of Cruise Control study, traversing certain du pairs or definitions guarantees some mutants be detected.

6.3 Conclusions from case studies

The two case studies suggest that by following Binder's traversal algorithm multiple transition trees with different mutation scores may be generated from one single statechart. Transition trees with higher mutation scores tend to have more du pairs and definitions. We therefore can conclude that both du pairs and definitions can be used to determine the effectiveness of a transition tree. When alternative trees exist, the tree with the largest number of du pairs or definitions should be selected. From a practical point of view, using definitions is less expensive than du pairs as identifying definitions involves fewer rules and less costly algorithms, as discussed in Section 6.1.4.

In both of the two case studies, we've looked at the live mutants with each transition tree. It is shown that the live mutants with one transition tree may be a complement set of the live mutants with another transition tree. For instance, in the Cruise Control case study, the mutants missed by TT1 are all detected by TT2, and the mutants missed by TT2 are all detected by TT1. In the VCR case study, the live mutants with Group 1 are all killed by Group 4 and the live mutants with Group 4 are all killed by Group 1. The analysis of the live mutants shows that covering certain du pairs guarantees that some mutants be detected. In other words, the effectiveness of a transition tree could be improved by adding paths that cover the du pairs that are not already covered by the tree. Additionally, this result appears to hold for uncovered definitions too: covering the definitions that are

not already covered by a transition tree would increase the transition tree effectiveness.

This leads us to an optimized transition tree strategy, as detailed in the next Chapter.

CHAPTER 7 OPTIMIZING TRANSITION TREES

The motivation is to produce an optimal tree which has maximal fault detection power and minimal cost. Cost is typically considered as being proportional to test set size [6]. In our context, we measure size as the number of method invocations in the test set, which is the sum of the lengths of all tree paths. This work refers to the sum of the length of all tree paths as *the cumulative length* of a transition tree [6].

As suggested by the results of the case studies (Chapter 6), the more du pairs or definitions covered by a tree, the more effective the tree. So we could require either *all du pairs* or *all definitions* as the data flow information to be covered by the optimal tree. Results from the VCR case study, however, shows that transition trees with the same fault detection power (the same mutation score and an identical set of live mutants) tends to have an identical set of definitions while different sets of du pairs. This suggests that covering all du pairs may lead to more expensive trees and identical effectiveness. Therefore, we intend to find an optimal tree that covers all the definitions in all transition trees and is minimal in terms of cost.

The rest of this Chapter consists of two sections: Section 7.1 describes the strategy of building the optimal tree. Section 7.2 reports the results of applying the strategy on the Cruise Control and VCR case studies, and Section 7.3 concludes this Chapter by summarizing the strategy and results from case studies.

7.1 Building an optimal tree

Building an optimal tree takes a sequence of steps. These steps are shown in Figure 57 by means of an activity diagram. First, transition trees with an identical set of definitions are grouped together (Step 1). Then, we select from each group a transition tree with minimal cumulative length to form a tree set, say τSet (Step 2). These two steps are illustrated in Figure 58. Next, we choose a transition tree with the largest number of definitions from τSet as the initial tree (Step 3). Step 4 uses an exhaustive search to select a set of tree paths from τSet that cover all the definitions not already covered by the initial tree and has minimal cumulative length. To reduce the set of paths that are exhaustively searched, we eliminate the paths of the initial tree, duplicate paths and paths that don't contain any additional definitions to the initial tree. Lastly, these paths are added to the initial tree.

Note that the above strategy uses an exhaustive search to find all sets of tree paths that cover all the definitions not already covered by the initial tree, and then chooses the one with minimal cumulative length. The exhaustive search looks at every path (there are p paths, i.e., C_1^p), every pair of paths (i.e., C_2^p), every triplet of paths (i.e., C_3^p), and so on

and so forth. The number of searches needed is $\sum_{i=1}^p C_i^p$. Since $\sum_{i=1}^p C_i^p = 2^p - 1$, the

complexity of the exhaustive search is $O(2^p)$. Other search strategies are discussed in Section 7.3.

Figure 57 Building an optimal tree

Figure 58 Formation of a tree set

7.2 Results from case studies

This section reports on the use of the optimal tree strategy on Cruise Control and VCR case studies.

7.2.1 Cruise Control

For the cruise control case study, TT2 has the largest number of definitions, so TT2 is selected as the initial tree. The set of paths to be searched (pathSet3 in Figure 57) consists of five tree paths, so 31 searches are needed and only one set of tree paths is found to be able to cover all the definitions. This set is added to TT2, which forms an optimal tree (Figure 59).

The optimal tree killed all the mutants in Table 15 and thus generated 100% mutation score. This is achieved at the expense of adding two tree paths to the test set, which amounts to 13% increase in cumulative length.

Recall that we could have selected du pairs as the data flow information to be covered by the optimal tree. Covering all du pairs, in the cruise control system, also leads to kill all the mutants, but requires 10 tree paths to be added to the test set. This is an 89% increase in cumulative length.

Figure 59 Optimal transition tree (Cruise Control)

7.2.2 VCR

For the VCR system, the twelve transition trees are divided into four groups according to the sets of definitions they contain. Since all the twelve transition trees have the same cumulative length, we randomly select one transition tree from each group to form the $tSet$ (Figure 57), which is $\{TT1, TT2, TT3, TT4\}$. As TT1 has the largest number of definitions, it is taken as the initial tree. $pathSet3$ (Figure 57) has 18 tree paths, so 262143 searches are needed. Like the Cruise Control case study, only one set of paths is able to cover all the definitions. Adding this set of paths to TT1 forms an optimal tree (Figure 60). This leads to kill all the mutants in Table 18 and thus reaching 100% mutation score, and is achieved at the expense of adding 12 tree paths to the test set, amounting to 29% increase in cumulative length.

Building a tree to cover all the du pairs in all transition trees, in the VCR system, would result in adding 41 tree paths to the test set, which is a 106% increase in cumulative length.

Figure 60 Optimal transition tree (VCR)

7.3 Conclusions

This Chapter defines an optimal tree to be a tree which covers all the definitions in all transition trees, and proposes a strategy to build the optimal trees. The results of applying the strategy to two case studies suggest that the optimal tree is highly effective at detecting faults while doesn't incur significant increase in cost. On the other hand, a tree which covers all du pairs in all transition trees, though achieves 100% mutation score, leads to a much more expensive set of test cases.

Recall that to find an optimal tree, an exhaustive search is used to find a set of paths that cover all the definitions not already covered by an initial tree, with minimal cost. The complexity of the exhaustive search is exponential. As discussed in Section 7.1, the set of paths to be searched is reduced by removing duplicate paths and paths that doesn't contain any additional definitions to the initial tree. This significantly reduces the number of paths to be searched since transition trees generated from one statechart typically have a large number of tree paths in common. Therefore, it seems likely that exhaustive search would fit in many practical cases.

For more complex statecharts, like the VCR, other search techniques such as Genetic Algorithm could be considered and this will be done in future research.

CHAPTER 8 CONCLUSIONS AND FUTURE WORK

This Chapter is divided into two sections: Section 8.1 provides an overview of the work done in this thesis and Section 8.2 discusses the limitations of this research and outlines the directions for future work.

8.1 Summary and conclusions

The objective of this research is to investigate how data flow information can be used to improve state-based testing criteria. To this end, we provide a comprehensive methodology to conduct data flow analysis of UML statechart, apply it to the transition tree criterion, and use it on two case studies. Based on the results of two case studies, we propose an optimal tree strategy to optimize transition trees and validate it through the two case studies.

Our data flow analysis methodology consists of four steps: the first step is to transform a UML statechart into an Event/Action Flow Graph (EAFG), where a node represents the postcondition of an event/action and an edge is associated with the precondition of an event/action and the guard condition of a transition. By doing so, the control flow relationship among the events and actions of UML statechart are explicitly modeled. In this step, we have strived to support the UML statechart notation as much as possible. The second step is to modify the EAFG such that control flow paths in event/action implementation, as suggested by the postcondition are accounted for. The third step is to determine definitions and uses in nodes and edges in the modified EAFG with a set of

automatable rules. Lastly, def clear paths and du pairs are derived from the modified EAFG by using a set of algorithms adapted from [3]. Note that this is a general approach of data flow analysis of UML statecharts, which can be used in many different ways. In particular it does not assume any technique used to produce test cases. In our case, we apply this approach to one of the state-based testing criteria, i.e. the transition tree criterion: a transition tree is first transformed into a transition tree EAFG, then modified to include only feasible paths; determining definitions and uses and deriving def clear paths and du pairs in the transition tree EAFG use the general rules and algorithms we defined. We applied this procedure to two case studies to investigate whether data flow information can be used to select more effective transition trees.

The results of the case studies suggest that both du pairs and definitions are good indicators of the fault detection effectiveness of a transition tree: when multiple transition trees can be generated from one statechart, the transition tree which is the most effective at detecting mutants tends to cover the largest number of du pairs and definitions. We examine the live mutants for at least one transition tree in both of the case studies. The results show that certain du pairs and definitions guarantee that some mutants be detected. From these results, we can draw the conclusion that data flow information contained in a transition tree can be used to select a tree with greater fault detection rates, i.e., the transition tree which contains the largest number of du pairs or definitions would be most effective at detecting mutants. We see that both du pairs and definitions can be used to select trees. Using definitions, however, may be a better choice in practice since identifying definitions is easier and less costly than identifying du pairs, though in both

cases, the user input is needed to determine the feasible paths within a compound node in a transition tree EAFG.

It is also shown in the case studies that even the most effective tree may miss a significant number of mutants (e.g. 25% in the VCR case study). We then propose the optimal tree strategy to optimize transition trees. The optimal tree covers all the definitions in all transition trees and is minimal in terms of cost. The results of applying the strategy to the two case studies suggest that the optimal tree is highly cost-effective at detecting mutants. We could have chosen `du` pairs as the data flow information to be covered by all transition trees. This yields the same mutation score as our optimal tree, but its cost is significantly higher. Though these results need to be confirmed by additional case studies, it is likely that the optimal tree strategy would fit many practical situations.

8.2 Limitations and future work

Several methodology limitations and possible solutions are discussed in Section 4.5. These limitations are the following: (1) The predicate parts of `if-then-else` and `implies` expressions are considered as one term which is associated with an edge; (2) An artificial order is created within a compound node; (3) When an attribute (or link) of a class appears in the parameter of an OCL iterative collection operation, we consider the attribute (or link) of all the objects of the class to be defined (or used). Future work will investigate the possible solutions proposed for these limitations.

Currently we rely on the user input to determine the incompatible sequences of operations in EAFGs and infeasible paths within a compound node in a modified EAFG.

Though these can not be fully automated from a theoretical point of view, future work will explore practical heuristics that can provide approximate solutions. As discussed in the previous section, the number of definitions in a transition tree is shown to be a good indicator of its fault detection effectiveness. To obtain the exact number of definitions in a transition tree, the user input is required to determine the feasible path within each compound node in the transition tree EAFG. A more practically applicable approach that would not require that the user identify infeasible paths is to obtain, for example, the maximum, minimum and/or median number of definitions as a good enough approximation.

For the optimal tree strategy, an exhaustive search is used to select a set of tree paths with minimal cost. Future work will address the issue of optimizing the search algorithm using other techniques such as Genetic Algorithms.

Additional case studies should be performed to evaluate the generality of our approach and to confirm our results. An interesting case study would be one which has a rich set of collection operations in its operation contracts. This will help to assess the rules we defined for determining definitions and uses in collection operations. Also, since our results are based on transition trees generated using Breadth First Search, transition trees which are obtained by Depth First Search should be investigated to confirm our results.

REFERENCES

- [1] S. S. Alhir, *Guide to Applying the UML*, Springer-Verlag, 2004.
- [2] G. Antoniol, L. C. Briand, M. D. Penta, and Y. Labiche, "A Case Study Using the Round-Trip Strategy for State-Based Class Testing," *Proc. 13th IEEE International Symposium on Software Reliability Engineering (ISSRE'02)*, Annapolis, Maryland, USA, pp. 269-279, November 12-15, 2002.
- [3] A. Appel, *Modern Compiler Implementation in Java*, Cambridge University, 2002.
- [4] R. V. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 2000.
- [5] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
- [6] L. C. Briand, Y. Labiche, and Y. Wang, "Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statecharts," *Proc. 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, United Kingdom, pp. 86-95, May 23-28, 2004.
- [7] L. C. Briand, J. Cui, and Y. Labiche, "Towards Automated Support for Deriving Test Data from UML Statecharts," *Proc. ACM/IEEE International Conference on the Unified Modeling Language (UML'03)*, San Francisco, USA, pp. 249-164, October, 2003.
- [8] L. C. Briand, Y. Labiche, and H. Sun, "Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code," *Software - Practice and Experience*, vol. 33 (7), pp. 637-672, 2003.
- [9] P. Chevalley and P. Thevenod-Fosse, "An Empirical Evaluation of Statistical Testing Designed from UML State Diagrams: the Fight Guidance System Case Study," *Proc. 12th IEEE International Symposium on Software Reliability Engineering (ISSRE'01)*, Hong Kong, China, pp. 254-263, November 27 - 30, 2001.
- [10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek, "An Efficient Method of Computing Static Single Assignment," *Proc. 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Austin, Texas, pp. 25-25, January, 1989.

- [11] T. S. Chow, "Testing Software Design Modeled by Finite State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4 (3), pp. 178-186, 1978.
- [12] S. H. Edwards, "Black-Box Testing Using Flowgraphs: An Experimental Assessment of Effectiveness and Automation Potential," *Software Testing, Verification and Reliability*, vol. 10 (4), pp. 249-262, 2000.
- [13] P. G. Frankl and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, vol. 14 (10), pp. 1483-1498, 1988.
- [14] H. Gomaa, *Designing Concurrent, Distributed, and Real-time Applications with UML*. Addison-Wesley, 2000.
- [15] H. S. Hong, Y.G. Kim, S. D. Cha, D. H. Bae, and H. Ural, "A Test Sequence Selection Method for Statecharts," *Software Testing, Verification and Reliability*, vol. 10 (4), pp. 203 –227, 2000.
- [16] S. Kim, J. A. Clark, and J.A. McDermid, "The Rigorous Generation of Java Mutation Using HAZOP," *Proc. 12th International Conference on Software & Systems Engineering and Their Applications (ICSSEA'99)*, CNAM - Pairs, France, pp. 9-10 (11), December 8-10, 1999.
- [17] S. Kim, J. A. Clark, and J. A. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs," *Proc. Net.ObjectDays*, Erfurt, Germany, October 9-12, 2000.
- [18] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha, "Test Cases Generation from UML State Diagrams," *IEE proceedings Software*, vol. 146 (4), pp. 187-192, 1999.
- [19] K. N. King and A.J. Offutt, "A Fortran Language System for Mutation-based Software Testing," *Software - Practice and Experience*, vol. 21 (7), pp. 686-718, 1991.
- [20] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture - Practice and Promise*, Addison-Wesley, 1st Edition, 2003.
- [21] T. Lengauer and R.E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *ACM Transactions On Programming Languages and Systems*, vol. 1 (1), pp. 121-141, 1979.
- [22] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 2nd Edition, 1997.

- [23] R. Mitchell and J. McKim, *Design by Contract, by Example*, Addison-Wesley, 2002.
- [24] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating Test Data from State-based Specifications," *Software Testing, Verification and Reliability*, vol. 13 (1), pp.25-53, 2003.
- [25] Meilir Page-Jones, *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley, 2000.
- [26] S. Rapps and E. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, vol. 11 (4), pp. 367-375. 1985.
- [27] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.
- [28] S. Bennett, J. Skelton, and K. Lunn, *Schaum's Outlines of UML*, McGraw-Hill, 2001.
- [29] OMG, "Unified Modeling Language (UML)," Object Management Group V1.4, www.omg.org/technology/uml/, 2001.
- [30] OMG, "Unified Modeling Language (UML)," Object Management Group V1.5, www.omg.org/technology/uml/, 2003.
- [31] E. Weyuker, "More Experience with Data Flow Testing," *IEEE Transactions on Software Engineering*, vol. 19 (9), pp. 912-919, 1993.
- [32] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.
- [33] S. H. Zweben, W. D. Heym, and J. Kimmich, "Systematic Testing of Data Abstractions Based on Software Specification," *Software Testing, Verification and Reliability*, vol. 1(4), pp. 39-55, 1992.

Appendix A Data Dictionary for EAFG Metamodel

A.1 Entity classes

`Node` is an abstract class. It can be either a `BasicNode` or a `CompoundNode`.

`Edge` represents an edge in an EAFG.

`Constraint` is an abstract class. It can be either a `Precondition`, a `Postcondition` or a `Guard`.

`Precondition` represents the precondition of an operation.

`PostCondition` represents the postcondition of an operation.

`Guard` represents the guard condition of an event.

`CompoundNode` is associated with the postcondition of an operation. It can be an `EventNode`, an `ActionNode`, or an `ActivityNode`.

`EventNode` is a `CompoundNode` representing an event.

`ActionNode` is a `CompoundNode` representing an action.

`ActivityNode` is a `CompoundNode` representing an activity.

`CallEvent` is an `EventNode` representing a call event.

`SignalEvent` is an `EventNode` representing a signal event.

`ChangeEvent` is an `EventNode` representing a change event.

`TimeEvent` is an `EventNode` representing a time event.

`CallAction` is an `ActionNode` representing a call action.

`AssignmentAction` is an `ActionNode` representing an assignment action.

`SendAction` is an `ActionNode` representing a send action.

`CreateAction` is an `ActionNode` representing a create action.

`DestroyAction` is an `ActionNode` representing a destroy action.

`Term` is an OCL expression without connectives.

`BasicNode` contains zero or one term.

Start is a basic node representing the entry of an EAFG and the start of a compound node.

End is a basic node representing the exit of an EAFG and the end of a compound node.

ModelElement is an abstract class. It can be either an attribute, a link or aParameter.

Attribute represents an attribute of an instance of a class.

Link represents a link of an instance of a class.

Parameter represents a parameter of a context operation.

ValueInTerm is an association class representing the values of a model element in a Term.

Value represents the value of a model element in an OCL expression. A value can be either a pre-value or an after-value

DataFlow represents the data flow information of a pre-value or an after-value, which can be def, c-use, p-use or null.

Operation refers to the predefined OCL operations. An operation can be either a **BasicTypeOperation** or **CollectionTypeOperation**.

BasicTypeOperation refers to the operations defined on OCL basic types.

CollectionTypeOperation refers to the operations defined on OCL collection types.

Iterative represents the operations that iterate over collection elements and take **OclExpression** as parameter.

NonIterative represents all the collection operations that are not classified as iterative operations.

Type1 represents those noniterative operations that return a new collection.

Type2 represents those noniterative operations that return a Boolean value, an integer or an element of a collection.

A.2 Associations between entity classes

Source:	Node
Target:	Node
Description:	A node has an association with itself.
Role source:	predecessor
Role target:	successor
Cardinality source:	*
Cardinality target:	*

Source:	Node
Target:	Edge
Description:	A tail node has an association with its outgoing edge.
Role source:	tail
Role target:	outgoing
Cardinality source:	1
Cardinality target:	*

Source:	Node
Target:	Edge
Description:	A head node has an association with its incoming edge.
Role source:	head
Role target:	incoming
Cardinality source:	1
Cardinality target:	*

Source:	Node
Target:	Postcondition
Description:	A node has a composition relationship with the postcondition.

Role source:	
Role target:	postcondition
Cardinality source:	1
Cardinality target:	0..1
Source:	Edge
Target:	Precondition
Description:	An edge has a composition relationship with the precondition.
Role source:	
Role target:	precondition
Cardinality source:	1
Cardinality target:	0..1
Source:	Edge
Target:	Guard
Description:	An edge has a composition relationship with the guard.
Role source:	
Role target:	guard
Cardinality source:	1
Cardinality target:	0..1
Source:	Constraint
Target:	Term
Description:	A constraint has an association with its terms.
Role source:	
Role target:	
Cardinality source:	*
Cardinality target:	*

Source:	CompoundNode
Target:	BasicNode
Description:	A compound node has a composition relationship with its start basic nodes.
Role source:	
Role target:	start
Cardinality source:	0..1
Cardinality target:	0..1
Source:	CompoundNode
Target:	BasicNode
Description:	A compound node has a composition relationship with its end basic nodes.
Role source:	
Role target:	end
Cardinality source:	0..1
Cardinality target:	0..1
Source:	Term
Target:	ModelElement
Description:	A term has an association with the model elements involved in it.
Role source:	
Role target:	
Cardinality source:	*
Cardinality target:	*
Source:	Term
Target:	Operation
Description:	A term has an association with the operations involved in it.

Role source:	
Role target:	
Cardinality source:	*
Cardinality target:	*
Source:	Node
Target:	Term
Description:	A node has an association with the terms appearing in it.
Role source:	
Role target:	
Cardinality source:	*
Cardinality target:	*
Source:	Edge
Target:	Term
Description:	An edge has an association with the terms appearing on it.
Role source:	
Role target:	
Cardinality source:	*
Cardinality target:	*
Source:	ValueInTerm
Target:	Value
Description:	A valueInTerm has a composition relationship with its values.
Role source:	
Role target:	
Cardinality source:	1
Cardinality target:	1..2

Source:	Value
Target:	DataFlow
Description:	A value has an association with the dataflow information it contains.
Role source:	
Role target:	
Cardinality source:	1
Cardinality target:	1
Source:	CollectionTypeOperation
Target:	Value
Description:	A collectionTypeOperation has an association with the values involved in its parameter.
Role source:	
Role target:	involvedInOpParam
Cardinality source:	*
Cardinality target:	*
Source:	CollectionTypeOperation
Target:	Value
Description:	A collectionTypeOperation has an association with the collection on which it is applied.
Role source:	
Role target:	collection
Cardinality source:	*
Cardinality target:	1

Appendix B Algorithms to Derive Data Flow Information in EAFG

Figure 61 Class diagram for dataFlowGenerator

B.1 Algorithms to compute dominator trees

A node d *dominates* a node n if every path of directed edges from the entry node to n must go through d . Every node dominates itself [3]. In an EAFG, every node except the initial node is dominated by at least one node other than itself.

An *immediate dominator* of node n , denoted as $\text{idom}(n)$, dominates n and doesn't dominate any other dominator of n , and it's not the same node as n . In an EAFG, every node except the initial node has exactly one immediate dominator. The initial node has no immediate dominator.

A *dominator tree* is a graph containing every node of the flow graph, and for every node n an edge from $\text{idom}(n)$ to n . To compute dominator trees, we use Lengauer and Tarjan Algorithm [21], which runs in $O(E \cdot \log(N))$ execution time, where E is the number of edges and N is the number of nodes in the EAFG.

The algorithm consists of three steps:

Step 1: Compute depth-first spanning trees. A *depth-first spanning tree* is a recursion tree obtained by traversing the graph using the depth-first search (DFS) algorithm and numbering each node with a depth-first number (dfsNum). The algorithm for computing depth-first spanning tree is described below:

Algorithm DFS (DomTreeNode p, DomTreeNode n)

Input:

p: DomTreeNode

n: DomTreeNode

Output:

Function DFS (DomTreeNode p, DomTreeNode n)

var: i: int

```

        dfsNum: int
        nodes: Sequence (DomTreeNode)
begin
if n.index = 0 then
    n.index := dfsNum;
    n.parent := p;
    nodes := nodes->union(n);
    dfsNum := dfsNum + 1;
    for i := 1 to n.successor->size do
        DFS(n, n.successor->at(i));
    endif;
end.

```

Step 2: compute for each node v that is not the root of the tree, a semidominator of v . A node s is called a *semidominator* of n if s is the highest possible ancestor of n at which there exists a path that departs from the tree and rejoins the tree at n . To find the semidominator of a node n , we need to consider all predecessors v of n in the EAFG using the following semidominator theorem [3]:

- If v is a proper ancestor of n in the spanning tree (so $v.index < n.index$), then v is a candidate for the semidominator of node n ($n.semi$).
- If v is a nonancestor of n (so $v.index > n.index$), then for each u that is an ancestor of v (or $u = v$), let $u.semi$ be a candidate for $n.semi$.

Of all these candidates, the one with lowest $dfsNum$ is the semidominator of n .

Step 3: The immediate dominator is computed by using the following Dominator Theorem [3] : on the spanning-tree path below $n.semi$ and above or including n , let y be the node with the smallest-numbered semidominator (minimum $y.semi.index$).

```

if y.semi = n.semi
    then n.idom = n.semi
else n.idom = y.idom

```

The algorithm for step 2 and step 3 is described below:

Algorithm dominators (DomTreeNode root)

Input: root: DomTreeNode

Output:

```
Function dominators(DomTreeNode root)
var:      i, j: int
          dfsNum: int
          nodes: sequence(DomTreeNode)
          p, s, v, s', y, m, n': DomTreeNode

function: DFS
          AncestorWithLowestSemi
          link
begin
  DFS (null, root)
  for i := nodes->size downto 2 do //skip over the root node
  begin n := nodes->at(i)
    p := n.parent;
    s := p
    // calculate the semidominator of n, based on the
    // semidominator theorem
    for j := 1 to n.predecessor->size do
    begin v := n.predecessor->at(j)
      if v.index ≤ n.index then
        s' := v;
      else s' := AncestorWithLowestSemi(v).semi
      if s'.index < s.index then
        s := s';
    end;
    n.semi := s;
    s.bucket := s.bucket->union(n);
    //link the path from s to n into the spanning forest
    Link(p, n)
    for k := 1 to p.bucket->size do
      m:= p.bucket->at(k);
      y := AncestorWithLowestSemi(p.bucket->at(k))
      //calculate the dominator of p.bucket->at(k) if y.semi = p.bucket->
      //at(k).semi
      if y.semi = p.bucket->at(k).semi then
        p.bucket->at(k).idom := p;
      //if y.semi ≠ p.bucket->at(k).semi, defer the calculation until y's
      //dominator is known
      else p.bucket->at(k).samedom := y;
      endif;
    end;
    p.bucket->isEmpty
  for i := 1 to dfsNum - 1 do
  begin n' := nodes->at(i);
    //perform the deferred dominator calculations
    if n'.samedom->notEmpty then
      n'.idom := n'.samedom.idom
    end
  end.
```

Algorithm link (DomTreeNode p, DomTreeNode n)

Input: p, n: DomTreeNode

Output:

```
Function link (DomTreeNode p, DomTreeNode n)
begin
  n.ancestor := p;
  n.best := n;
end.
```

Algorithm ancestorWithLowestSemi (DomTreeNode v)

Input: v: DomTreeNode

Output: v.best: DomTreeNode

```
Function ancestorWithLowestSemi (DomTreeNode p)
var: a: DomTreeNode
begin
  a := v.ancestor
  if a.ancestor->notEmpty then
    b := AncestorWithLowestSemi(a);
    v.ancestor := a.ancestor;
    if b.semi.index < v.best.semi.index then
      v.best := b
    endif;
  endif;
  return v.best;
end.
```

B.2 Algorithms to compute dominance frontiers

The *dominance frontier* of a node x , $x.DF$, is the set of all nodes w such that x dominates a predecessor of w , but does not strictly dominate w (x strictly dominates w if x dominates w and $x \neq w$). We need $x.DF$ for every node in an EAFG because if the node x contains a definition of a variable, say a , then any node z in the dominance frontier of x needs a ϕ -function for a , refer to [3] for justification.

Algorithm computeDF (DomTreeNode n)

Input: n: DomTreeNode

Output:

```
Function computeDF (DomTreeNode n)
var: y, z, w: DomTreeNode
      i, j, k: int
begin
  //computes n.DFlocal: The successors of n that are not strictly dominated by n
  for i:= 1 to n.successor->size do
    begin y := n.successor->at(i)
      if y.idom != n then
        n.domFrontier := n.domFrontier->union(y);
    end;
  for j := 1 to n.children->size do
    begin z := n.children->at(j);
      ComputeDF (z);
  //computes c.DFup: Nodes in the dominance frontier of c that
  //are not strictly dominated by n's immediate dominator.
    for k := 1 to z.domFrontier->size do
      begin w := z.domFrontier->at(k)
        if w.idom != n or w != n then
          n.domFrontier := n.domFrontier->union{n.children->
                                                    at(j).domFrontier->at(k)}
      end;
    end;
  end;
end.
```

B.3 Algorithms to insert ϕ functions

Algorithm insert- ϕ -Functions (DominatorTree aDominatorTree)

Input: aDominatorTree: DominatorTree

Output:

```
Function insert- $\phi$ -Functions (DominatorTree aDominatorTree)
var: t, w, y, p: DomTreeNode
      i, j, k, n, m, l, q, r, s: int
      v, e: ModelElement
      yhasphi, yOrigv: Boolean
      aTerm: Term
begin
  for i:= 1 to aDominatorTree.nodes->size do
    begin t := aDominatorTree.nodes->at(i);
      for j:= 1 to t.term.definedElement->size do
        begin v:= t.term.definedElement->at(j);
          v.defSite.node:= v.defSite.node->union(t);
        end;
      for k:= 1 to aDominatorTree.nodes.term.definedElement->size do
        begin e:= aDominatorTree.nodes.term.definedElement->at(k)
          w:= e.defsite.node
          while w->notEmpty do
            begin n := w->last;
              w := w->excluding(w->last);
              for m := 1 to n.domFrontiers->size do

```



```

begin y := n.domFrontiers->at(m)
  yhasPhi:= false
  for l := 1 to aDominatorTree.
    nodes.term.definedElement.phiSite->size do
      begin p:=aDominatorTree.nodes.term.definedElement.phiSite->
        at(l)
        if y == p then
          yHasPhi := true;
        end;
      if yHasPhi == false then
        Term aTerm := new Term();
        aTerm.definedElement :=aTerm.definedElement->union (e)
        for q := 1 to y.predecessor->size do
          begin aTerm.cuses := aTerm.cuses->union (v)
            aTerm.isPhiFunction := true;
            y.terms := aTerm->union(y.terms);
            aDominatorTree.nodes.term.definedElement.phiSite:=
            aDominatorTree.nodes.term.definedElement.phiSite->union(y);
            yOrigv := false;
            for r := 1 to y.terms->size
              begin //ignore the term just inserted
                for s := 1 to y.terms->at(r).definedElement->size
                  begin if v == y.terms->at(r).defs->at(s) then
                    yOrigv := true;
                  end;
                end;
              if yOrigv = false then
                w := w->union(y);
              endif;
            end;
          end;
        end;
      end;
    end;
  end.

```

B.4 Algorithms to rename variables and get def clear paths

After the ϕ -functions have been inserted, we can walk the dominator tree, renaming the different definitions (including ϕ -Functions) of variable a to $a_1, a_2, a_3 \dots$ and generating def clear paths. In traversing the dominator tree, the algorithm “remembers” for each variable the most recently defined version of each variable, on a separate stack for each variable.

Algorithm rename&GetDefClearPaths (DomTreeNode n)

Input: n : DomTreeNode

Output:

```
Function rename&GetDefClearPaths(DomTreeNode n)
var:      y: DomTreeNode
          i, j, k, m, l, u, p, q, c: int
          x, a, b: ModelElement
          s, t, r: term
          aDupath: Dupath
          origv: String
begin
  for i := 1 to n.term->size do
    begin s := n.term->at(j);
      if s.isPhiFunction == false then
        for j := 1 to s.usedElement->size do
          begin x := s.usedElement->at(j);
            s.usedElement->at(j).name:= new String(x_x.stack.top);
            if ! dupath->exist(d:Dupath|d.definedElement.name =
                                                                    x_x.stack.top) then
              Dupath aDupath := new Dupath(x_x.stack.top);
              dupath:= dupath->union(aDupath);
            endif
            dupath->select(d:Dupath|d.definedElement.name=
                                                                    x_x.stack.top).usedTerm->union(s);
          end;
        endif;
      for k := 1 to s.definedElement -> size do
        begin a := s.definedElement ->at(k);
          a.count := a.count + 1;
          m := a.count;
          a.stack.push(m);
          s.definedElement->at(k).name:= new String(a_m);
          if not dupath->exist(d:Dupath|d.definedElement.name =
                                                                    a_a.m) then
            Dupath aDupath := new Dupath(a_a.i);
            dupath:= dupath->union(aDupath);
          endif
          dupath->select(d:Dupath|d.definedElement.name=
                                                                    a_a.i).definedTerm:= s;
        end;
      end;
    for l := 1 to n.successor->size do
      begin y := n.successor->at(l)
        z := y.predecessor.indexOf(n);
        for c := 1 to y.term->size do
          begin t := y.term->at(c);
            if t.isPhiFunction then
              b := t.usedElement->at(z);
              t.usedElement->at(z).name:= new String(b_b.stack.top);
              if not dupath->exist(d:Dupath|d.definedElement.name =
                                                                    b_b.stack.top) then
                Dupath aDupath := new Dupath(b_b.stack.top);
                dupath:= dupath->union(aDupath);
              endif
              dupath->select(d:Dupath|d.definedElement.name=
                                                                    b_b.stack.top).usedTerm->union(t);
            end;
          end;
        end;
      end;
    end;
  end;
```

```

    for p := 1 to n.children->size do
        rename&GetDefClearPaths (n.children->at(p))
    for q := 1 to n.term->size do
    begin r:=n.term->at(q);
        for u := 1 to r.definedElement->size do
            begin v:=r.definedElement->at(u);
                origv: = v.name.substring(0, w.indexOf("_");
                origv.stack.pop();
            end;
        end;
    end;
end.

```

Algorithm getDuPairs ()

Input:

Output: usedTerm: Sequence(Term)

```

Function getDuPairs()
var:      y: DomTreeNode
        i, j: int
        t1, t2: term
begin
    for i: = 0 to UsedTerm->size - 2 do
    begin t1:= UsedTerm->at(i);
        j:= i+1;
        while j < UsedTerm->size do
        begin t2: = UsedTerm->(j);
            if t1.node.name == t2.node.name then
                usedTerm:= usedTerm->excluding(t2)
            else j:= j+1;
            endif;
        end;
    end;
    return UsedTerm;
end.

```

Note that all three algorithms, that is, computeDF, Inserting ϕ -Function,

Rename&GetDuPaths, and GetDuPairs () run in approximately linear time.

Appendix C Class Diagram for Cruise Control (Design Level)

Figure 62 Class diagram for Cruise Control (at design level)

Appendix D Data Dictionary for Cruise Control

Attributes:

```
state: enum{idle, running, cruising, standby}
throttle: double
brakepedal: int
cruiseSpeed: double
speed: double
```

Methods:

```
engineOn()
pre:
post: self.cruiseSpeed = 0          Term1
      and
      self.speed = 0                Term2
      and
      self.throttle = 0             Term3
      and
      self.brakepedal = 0           Term4
      and
      self.state = #running         Term5

engineOff()
Pre: self.state != #idle            Term1
Post: self.state = #idle            Term2
      and
      if (self.cruiseSpeed@pre > 0) Term3
        then self.cruiseSpeed = 0   Term4
        else true                    Term5
      endif
      and
      if (self.speed@pre > 0)         Term6
        then self.speed = 0          Term7
        else true                    Term8
      endif
      and
      if (self.throttle@pre > 0)      Term9
        then self.throttle = 0       Term10
        else true                    Term11
      endif
      and
      if (self.brakepedal@pre > 0)    Term12
        then self.brakepedal = 0     Term13
        else true                    Term14
      endif

on()
pre: self.state != #idle            Term1
post: if (self.cruiseSpeed@pre != self.speed@pre) Term2
      then self.cruiseSpeed = self.speed@pre Term3
      else true                    Term4
    endif
```

```

and
if (self.cruiseSpeed > 0)
    then self.throttle = self.cruiseSpeed/CruiseController.st
    +(self.cruiseSpeed - self.speed)/CruiseController.diff
else true
endif
and
if self.state@pre != #cruising
    self.state = #cruising
else true
endif

accelerate()
Pre: self.state!= #idle
Post: if (self.brakepedal@pre > 0)
    then self.brakepedal = 0
    else true
endif
and
if(self.throttle@pre < CruiseController.maxThrottle - 5.0)
    then self.throttle = self.throttle@pre + 5.0
    else self.throttle = CruiseController.maxThrottle
endif
and
if(self.speed@pre < self.maxSpeed)
    then
        if ((self.throttle -
            self.speed@pre/CruiseController.airRes)/CruiseController.speedRate)
            < CruiseController.maxSpeed-self.speed@pre)
            then self.speed = self.speed@pre + (self.throttle -
                self.speed@pre/CruiseController.airRes)/
                CruiseController.speedRate
            else self.speed = CruiseController.maxSpeed
            endif
        else true
    endif
and
if (self.state@pre = #cruising)
    then self.state = #standby
    else true
endif

brake()
Pre: self.state!= #idle
Post: if (self.throttle@pre > 0.0)
    then self.throttle = 0.0
    else true
endif
and
if(self.brakepedal@pre < CruiseController.maxBrake - 1)
    then self.brakepedal = self.brakepedal@pre + 1
    else self.brakepedal = CruiseController.maxBrake
endif
and
if (self.speed@pre > 0)
    then
        if((self.speed@pre/CruiseController.airRes - 2* self.brakepedal)
            /#speedRate < self.speed@pre)
            then self.speed = self.speed@pre -
                (self.speed@pre/CruiseController.airRes-
                2*self.brakepedal)/CruiseController.speedRate
            else self.speed = 0

```

endif	
else true	Term12
endif	
and	
if (self.state@pre = #cruising)	Term13
then self.state = #standby	Term14
else true	Term15
endif	
off()	
pre: self.state = #cruising	Term1
post: self.state =#standby	Term2
resume()	
pre: self.state = #standby	Term1
post: if (self.cruiseSpeed@pre > 0)	Term2
then self.throttle = self.cruiseSpeed / CruiseController.st	
+ (self.cruiseSpeed - self.speed) / CruiseController.diff	Term3
else true	Term4
endif	
and self.state =#cruising	Term5

Appendix E The EAFG for Cruise Control Statechart

Figure 63 EAFG for CruiseController statechart

Appendix F Modified Transition Tree EAFGs (Cruise Control)

Control)

Figure 64 Modified EAFG for Transition Tree 2 (Cruise Control)

Figure 65 Modified EAFG for Transition Tree 3 (Cruise Control)

Appendix G Class Diagram for VCR (Analysis Level)

Figure 66 Class Diagram for VCR (at analysis level)

Appendix H Data Dictionary for VCR

Attributes:

```

state: enum{Off, TapeAbsent, RecPaused, Recording, Stopped, Forwarding, Rewinding, Playing, Paused}
eventStartTime: long
currentTime: long
isTapePresent: Boolean
isTapePulledToDrum: Boolean
numTimesTapePulled: int
tape:Tape

```

Methods:

```

EjectButton()
Pre:
Post:
  if self.isTapePresent = true
    then self.isTapePresent = false
  else true
endif
and
  if self.state@pre = #Off
    then self.state = self.state@pre
  else if (self.state@pre = #Playing or self.state@pre = #Recording)
    then isTapePulledToDrum = false
    and
      numTimesTapePulled = numTimesTapePulled@pre + 1
    and
      self.state = #TapeAbsent
    and
      if (self.tape.position@pre + NormalSpeed*(CurrentTime-EventStartTime))
        >self.tape.maxTapePosition
        then self.tape.position = self.maxTapePosition
        else self.tape.position = self.tape.position@pre + NormalSpeed*(CurrentTime@pre -
          EventStartTime@pre))
      endif
endif

```

Term1
Term2
Term3

Term4
Term5
Term6
Term7

Term8

Term9

Term10
Term11
Term12

```

else if (self.state@pre=#Forwarding or self.state@pre = #Rewinding)
then self.state = #TapeAbsent
if self.isTapePulledToDrum@pre = true
then self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
and
if (self.tape.position@pre + FasterSpeed*(CurrentTime@pre - EventStartTime@pre))
> self.tape.maxTapePosition
then self.tape.position = self.maxTapePosition
else self.tape.position = self.tape.position@pre + FasterSpeed *
(CurrentTime@pre - EventStartTime@pre))
endif
endif
else
if (self.tape.position@pre + FastestSpeed*(CurrentTime@pre-EventStartTime@pre)) >
self.tape.maxTapePosition
Term13
Term14
Term15
Term16
Term17
Term18
Term19
Term20
Term21
Term22
Term23
then self.tape.position = self.maxTapePosition
else self.tape.position = self.tape.position@pre + FastestSpeed
*(CurrentTime@pre - EventStartTime@pre))
endif
endif
else if(self.state@pre = RecPaused or self.state@pre =Paused)
Term24
Term25
Term26
Term27
Term28
Term29
Term30
then self.state = #TapeAbsent
and
self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
else if self.state@pre = #Stopped
then self.state = #TapeAbsent
else true
endif
endif
endif
endif
endif

```

```

PowerButton()
Pre:
Post:
if self.state@pre = #Off
then if not self.isTapePresent
then self.state = #TapeAbsent
else self.state=#Stopped
endif
else if (self.state@pre = #Playing or self.state@pre = #Recording)
then
self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
and
self.state = #Off
if (self.tape.position@pre + NormalSpeed*(CurrentTime@pre - EventStartTime@pre)) >
self.tape.maxTapePosition
then self.tape.position = self.maxTapePosition
else self.tape.position = self.tape.position@pre + NormalSpeed*(CurrentTime@pre -
EventStartTime@pre)
endif
else if (self.state@pre=#Forwarding or self.state@pre = #Rewinding)
then self.state = #off
if self.isTapePulledToDrum@pre = true
then
self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
and
if (self.tape.position@pre + FasterSpeed*(CurrentTime@pre -
EventStartTime@pre)) > self.tape.maxTapePosition
then self.tape.position = self.maxTapePosition
else self.tape.position = self.tape.position@pre +
FasterSpeed*(CurrentTime@pre - EventStartTime@pre)
endif
else
if (self.tape.position@pre + FastestSpeed*(CurrentTime@pre-EventStartTime@pre)) >
self.tape.maxTapePosition
then self.tape.position = self.maxTapePosition
else self.tape.position = self.tape.position@pre + FastestSpeed
*(CurrentTime@pre - EventStartTime@pre)
endif
endif
endif

```

```

else if (self.state@pre = #Paused or self.state@pre = #RecPaused)
then self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
and
self.state = #Off
and
else if (self.state@pre = #Stopped or self.state@pre = #TapeAbsent)
then self.state = #Off
else true
endif
endif
endif
endif
endif

InsertingTape(tape: Tape)
Pre: self.isTapePresent = false
Post:
self.isTapePresent = true
and
self.tape.position = tape.position
self.tape.isWriteProtected = tape.isWriteProtected
and
if (self.state@pre = #Off or self.state@pre = #TapeAbsent)
then if self.tape.tapeWriteProtected
then
self.isTapePulledToDrum = true
numTimesTapePulled = numTimesTapePulled@pre + 1
self.state = #Playing
eventStartTime = currentTime
else self.isTapePulledToDrum = false
self.state = #Stopped
endif
else true
endif
endif

Term23
Term24
Term25
Term26
Term27
Term28
Term29

Term1
Term2
Term3
Term4
Term5
Term6
Term7
Term8
Term9
Term10
Term11
Term12
Term13

```

```

RecButton ()
Pre:
Post:
  if self.state@pre = #RecPaused
  then self.state = #Recording
    eventStartTime = currentTime
  else if self.state@pre = #Stopped
    then if not self.tape.isWriteProtected
      then self.isTapePulledToDrum = true
        numTimesTapePulled = numTimesTapePulled@pre + 1
        self.state = #Recording
        eventStartTime = currentTime
      else self.state = #Stopped
        endif
      else true
        endif
    endif
  endif

PauseButton
Pre:
Post:
  if self.state@pre = #Recording
  then self.state = #RecPause
    if (self.tape.position@pre + NormalSpeed*(CurrentTime@pre -
      EventStartTime@pre)) > self.tape.maxTapePosition
      then self.tape.position = self.maxTapePosition
      else self.tape.position = self.tape.position@pre + NormalSpeed*(CurrentTime@pre -
        EventStartTime@pre)
      endif
    else if self.state@pre = #Playing
      if (self.tape.position@pre + NormalSpeed*(CurrentTime@pre - EventStartTime@pre)) >
        self.tape.maxTapePosition
        then self.tape.position = self.maxTapePosition
        else self.tape.position = self.tape.position@pre + NormalSpeed*(CurrentTime@pre -
          EventStartTime@pre)
        endif
      else true
        endif
    endif
  endif

Term1
Term2
Term3
Term4
Term5
Term6
Term7
Term8
Term9
Term10
Term11

```



```

StopButton()
Pre:
Post:
if (self.state@pre = # paused or self.state@pre = #RecPaused)
then self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
and
self.state = #stopped
else if (self.state@pre = #playing or self.state@pre = #Recording)
then self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
and
self.state@pre = #stopped
and
if (self.tape.position@pre + NormalSpeed*(CurrentTime@pre -
EventStartTime@pre)) > self.tape.maxTapePosition
then self.tape.position = self.maxTapePosition
else self.tape.position = self.tape.position@pre + NormalSpeed*(CurrentTime@pre -
EventStartTime@pre))
endif
else if (self.state@pre=#Forwarding or self.state@pre = #Rewinding)
then self.state= #stopped
if self.isTapePulledToDrum@pre = true
then
self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
and
if (self.tape.position@pre + FasterSpeed*(CurrentTime@pre -
EventStartTime@pre)) > self.tape.maxTapePosition
then self.tape.position = self.maxTapePosition
else self.tape.position = self.tape.position@pre +
FasterSpeed*(CurrentTime@pre - EventStartTime@pre)
endif
else
if (self.tape.position@pre + FastestSpeed*(CurrentTime@pre-EventStartTime@pre)) >
self.tape.maxTapePosition
then self.tape.position = self.maxTapePosition
else self.tape.position = self.tape.position@pre +
FastestSpeed*(CurrentTime@pre - EventStartTime@pre))

```

[illegible]

Term18
Term19

```
        then self.state = #Rewinding
        else true
        endif
    endif
```

endif

endif

and

eventStartTime = currentTime

Term20

FFButton()

Pre:

Post:

if self.state@pre = #Playing

then self.state = #Forwarding

if (self.tape.position@pre + NormalSpeed*(CurrentTime@pre - EventStartTime@pre)) >

self.tape.maxTapePosition

then self.tape.position = self.maxTapePosition

else self.tape.position = self.tape.position@pre + NormalSpeed*(CurrentTime@pre -

EventStartTime@pre))

endif

else if self.state@pre = #Rewinding

then self.state = #Forwarding

if self.isTapePulledToDrum@pre = true

then

if (self.tape.position@pre + FasterSpeed*(CurrentTime@pre - EventStartTime@pre)) >

self.tape.maxTapePosition

then self.tape.position = self.maxTapePosition

else self.tape.position = self.tape.position@pre + FasterSpeed*(CurrentTime@pre -

EventStartTime@pre))

endif

else

if (self.tape.position@pre + FastestSpeed*(CurrentTime@pre - EventStartTime@pre))

> self.tape.maxTapePosition

then self.tape.position = self.maxTapePosition

else self.tape.position = self.tape.position@pre + FastestSpeed*(CurrentTime@pre -

EventStartTime@pre))

endif

endif

else if self.state@pre = #Paused

then self.state = #Forwarding

else if (self.state@pre = #Stopped and self.tape.position != self.maxTapePosition)

Term16
Term17
Term18

Term19
Term20

```
        then self.state = #Forwarding
        else true
        endif
    endif
endif
and
eventStartTime = currentTime
```

Term21

EndOfTapeEvent()

Pre:

Post:

```
if (self.state@pre = #Playing or self.state@pre = #Recording)
then self.state = #Rewinding
and
self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
and
if self.tape.position > self.maxTapePosition
then self.tape.position = self.maxTapePosition
else true
endif
and
eventStartTime = currentTime
else if self.state@pre = #Forwarding
then self.state = #Rewinding
if self.isTapePulledToDrum@pre = true
then self.isTapePulledToDrum = false
and
numTimesTapePulled = numTimesTapePulled@pre + 1
else true
endif
and
if self.tape.position > self.maxTapePosition
then self.tape.position = self.maxTapePosition
else true
endif
and
eventStartTime = currentTime
else if self.state@pre = #Rewinding
```

Term1
Term2

Term3

Term4

Term5
Term6
Term7

Term8
Term9
Term10
Term11
Term12

Term13
Term14

Term15
Term16
Term17

Term18
Term19

```

then self.state = #Stopped
  if self.isTapePulledToDrum@pre = true
    then self.isTapePulledToDrum = false
    and
      numTimesTapePulled = numTimesTapePulled@pre + 1
    else true
  endif
  if self.tape.position < self.minTapePosition
    then self.tape.position = self.minTapePosition
    else true
  endif
endif
endif

endif

PlayButton()
Pre:
Post:
if (self.state@pre = #Forwarding or self.state@pre = #Rewinding)
then self.state = #PlayingTerm2
  if self.isTapePulledToDrum@pre = true
    then
      if (self.tape.position@pre + FastestSpeed*(CurrentTime@pre -
        EventStartTime@pre)) > self.tape.maxTapePosition
        then self.tape.position = self.maxTapePosition
        else self.tape.position = self.tape.position@pre +
          FastestSpeed*(CurrentTime@pre - EventStartTime@pre))
      endif
    else
      self.isTapePulledToDrum = true
      and
        numTimesTapePulled = numTimesTapePulled@pre + 1
      if (self.tape.position@pre + FastestSpeed*(CurrentTime@pre-EventStartTime@pre)) >
        self.tape.maxTapePosition
        then self.tape.position = self.maxTapePosition
        else self.tape.position = self.tape.position@pre + FastestSpeed*(CurrentTime@pre -
          EventStartTime@pre))
      endif
    endif
  else if self.state@pre = #Paused
    then self.state = #Playing
    else if self.state@pre = #Stopped

```

```

    then self.state = #Playing
    and
    self.isTapePulledToDrum = true
    and
    numTimesTapePulled = numTimesTapePulled@pre + 1
    else true
    endif
    endif
    endif
    and eventStartTime = currentTime

```

Term15

Term16

Term17

Term18

Term19

Appendix I Transition Trees for VCR

Figure 67 Transition Tree 1 (VCR)

Figure 69 Transition Tree 3 (VCR)

Figure 70 Transition Tree 4 (VCR)

Figure 71 Transition Tree 5 (VCR)

Figure 73 Transition Tree 7 (VCR)

Figure 74 Transition Tree 8 (VCR)

Figure 75 Transition Tree 9 (VCR)

Figure 76 Transition Tree 10 (VCR)

Figure 77 Transition Tree 11 (VCR)

Figure 78 Transition Tree 12 (VCR)