Improving State-Based Coverage Criteria Using Data Flow Information

L.C. Briand, Y. Labiche, Q. Lin
Software Quality Engineering Laboratory
Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada
{briand, labiche, qinglin}@sce.carleton.ca

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 report presents a comprehensive methodology to perform data flow analysis of UML statecharts, applies it to the round-trip path (transition tree) coverage criterion and reports 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.
# Table of Contents

1. Introduction ................................................................................................................. 3
2. Related Works ................................................................................................................ 4
   2.1 Control flow coverage criteria based on UML statecharts ........................................ 5
   2.2 Data flow coverage criteria based on UML statecharts ........................................... 7
   2.3 Deriving data flow information from postconditions .............................................. 8
   2.4 Detailed objectives ................................................................................................... 9
3. Data Flow Analysis of UML Statecharts ........................................................................ 10
   3.1 Transforming UML statecharts into event/action flow graphs .................................. 10
   3.2 Modified EAFGs ..................................................................................................... 20
   3.3 Determining definitions and uses in edges and nodes .............................................. 24
   3.4 Identifying definition use pairs and definition clear paths in EAFG ........................... 43
   3.5 Data Flow Analysis of Transition Trees .................................................................. 46
4. Case Studies .................................................................................................................... 48
   4.1 Cruise Control ........................................................................................................ 49
   4.2 VCR ....................................................................................................................... 54
   4.3 Conclusions from case studies .............................................................................. 59
5. Optimizing Transition Trees .......................................................................................... 60
   5.1 Building an optimal tree ......................................................................................... 60
   5.2 Results from case studies ...................................................................................... 61
   5.3 Conclusions ............................................................................................................ 63
6. Conclusions and Future Work ....................................................................................... 64
7. References ..................................................................................................................... 68
Appendix A Instantiation Examples .................................................................................. 71
Appendix B Data Dictionary for Cruise Control .............................................................. 74
Appendix C Class Diagram for VCR (Analysis Level) ...................................................... 77
Appendix D Data Dictionary for VCR ............................................................................... 78
Appendix E Optimal Transition Tree (VCR) ................................................................... 89
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. 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 [20], all paths in transition trees (round-trip paths) [3], full predicates [20], and all transition pairs [20]. Empirical evaluations have revealed limitations of these criteria [5]. 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 [11], [12]. 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. Operation contracts and guard conditions are assumed to be written in Object Constraint Language (OCL) since it is the natural choice as a formal constraint language in the context of UML.

Combining OCL operation contracts with UML models is advised by current and emerging software paradigms [17]. The principle behind using contracts to specify operations is referred to as the design by contract principle [18]. 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. 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 [23]. 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.

In this research, we propose a comprehensive methodology to conduct data flow analysis of UML statecharts. It 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 [2]. 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 report is organized as follows: Section 2 discusses the related work. Section 3 presents our approach of data flow analysis of UML statecharts. Section 4 reports on two performed case studies. Section 5 describes the strategy to generate optimal trees. Section 6 presents conclusions and points out directions for future work.

2 RELATED WORKS

This section looks at the state of the art in the fields related to our research. Section 2.1 and Section 2.2 review coverage criteria based on UML statecharts that focus on control and data
flow information, respectively. Section 2.3 presents the approach taken by [26] to derive data flow information from postconditions. Section 2.4 states more precisely the objective of this research in light of existing works.

2.1 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 [20]: 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.

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 [20] 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 [9] 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 [3]. 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 [8] generate test cases directly from hierarchical statecharts. Steps for flattening statecharts are provided in [12]. 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 [1], [5], however, have revealed the limitations of the strategy. Antoniol et al. applied the round-trip strategy to a typical state-dependent container class [1]. 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 [5]. 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.

2.2 Data flow coverage criteria based on UML statecharts

All the criteria we discussed in Section 2.1 are defined in terms of states and transitions. Hong et al. [12] 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. [15] generate test cases based on data flow information in a UML statechart. A number of data flow criteria were defined in [21], [11] 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. first transform the statechart into an Extended Finite State Machine (EFSM), then transform the EFSM into a flow graph.

It is worth noting that the data flow criteria proposed by [21], [11] 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. [15] 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.

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. [26], 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.

2.3 Deriving data flow information from postconditions

Zweben et al. [26] focused on applying control flow and data flow testing criteria to ADT modules which are specified in terms of operation pre- and postconditions. Edwards [10] 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. [26] 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 [26], 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 2.2, this approach is not based on a state model.
2.4 Detailed objectives

As discussed in Section 2.1, empirical studies have revealed the limitations of some of the state-based criteria [5]. 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 2.1), 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 2.2 and Section 2.3. 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.

<table>
<thead>
<tr>
<th>Existing works</th>
<th>Statechart notation</th>
<th>Guard</th>
<th>Event</th>
<th>Action</th>
<th>Operation contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Only assignment</td>
<td>No</td>
</tr>
<tr>
<td>[26], [11]</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Only postcondition</td>
</tr>
</tbody>
</table>

Table 1 Summary of contributions and limitations of existing works
3 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 3.1). 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 3.2). 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 3.3. 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 [2] (Section 3.4). Section 0 applies the approach to the transition tree criterion.

3.1 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 [26] and [10] is that EAFGs are transformed directly from statecharts whereas flow graphs in [26] and [10] 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 1, 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.

An EAFG can have cycles when there are cycles in the statechart, as illustrated in Figure 2.
In this research, we analyze actions in an action sequence according to their written order and assume entry and exit actions are moved to transitions. 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 3 shows an example where the statechart has an action sequence $a_1; a_2$, an entry action, $a_3$, an exit action $a_4$, and an internal transition, $e_2/a_5$. 

(a) (b) (c)
3.1.1 EAFG Metamodel

We model the structure of EAFGs as described in the previous section by means of the metamodel in Figure 4.

![EAFG Metamodel Diagram](image)

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.

A UML statechart may contain four kinds of events and eight kinds of actions [4]. 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 5 and Figure 6 (the EAFG corresponding to Figure 5) whose elements are discussed in the following sections.
3.1.2 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 7. This is also the case of events $m1(n1)$ and $m3()$ in Figure 5.

![Figure 7 Representing call events in EAFGs](image)

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 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 3.3 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. A call action takes the form: \texttt{object-set.action-name(parameter list)} [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 \texttt{m2 \(n2\)} and \texttt{m4} in Figure 5. 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).

### 3.1.3 Signal events and send actions

Signal events and send actions denote asynchronous executions. 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 \texttt{<<signal>>}) [22]. For our purpose, this is equivalent to a call event (the operation called is the handler) and the discussion of the previous section applies. Another solution, suggested in [4], is to consider that the handler of the signal event is the action out of the transition. 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.

### 3.1.4 Change events

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 \( \text{when (a Boolean expression)} \) [4]. The Boolean expression in the \( \text{when} \) expression is the condition under which the transition fires and can be considered as part of the precondition for a change event to occur. Change events also have handlers. The only solution to indicate the handler is to consider that the action out of the transition is the handler. 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 8.

![Figure 8 Representing change events in EAFGs](image)

As an example, the transition between \( S_3 \) and \( S_4 \) in Figure 5 is triggered by the change event \( \text{when } a_2=5 \). It is represented in the EAFG (Figure 6) as a node \( S_3 \_ (\text{when } a_2=5) \_ true \) which has an empty OCL expression and its incoming edge is associated with \( a_2=5 \).

### 3.1.5 Time events

A time event triggers a state transition when a specific deadline expires [4]. Like a change event, the handler of a time event is the action out of the transition. So, 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 9

![Figure 9 Representing time events in EAFGs](image)

### 3.1.6 Assignment actions

An assignment action sets the value of an object’s attribute. It is specified as \texttt{aTargetObject.anAttribute:=expression} [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 1) 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 3.3. 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 10). For instance in Figure 5, the action of the transition between \texttt{S1} and \texttt{S2} is an assignment action, \texttt{a1=a2+1}. It is represented in the corresponding EAFG with the node \texttt{S2_(a1=a2+1)_true}, which contains the postcondition of the assignment action, \texttt{a1=a2@pre+1}.

![Figure 10 Representing assignment actions in EAFGs](image)
3.1.7 Create actions and destroy actions

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 [23]. 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).

3.1.8 Terminate actions, return actions, and uninterpreted actions

A terminate action results in the self-destruction of the owning object of the state machines [23]. 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.

3.1.9 Guard conditions

As described in Section 3.1.1, 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 5, \(a_{2}>0\) is the guard condition for the trigger event \(m_{3}()\). It is associated with the incoming edge of the node \(s_{2}\_m_{3}\_a_{2}>0\).

3.1.10 Activities

An activity is an ongoing non-atomic execution while an object is in a state [4]. 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 \((n-1) \times m\) 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, we ignore them in the construction of the EAFG. Future work will address this limitation.

### 3.1.11 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 4 (Section 3.1.1) is therefore complemented with such information, as shown in Figure 11. Note that an ActivityNode is included for future extensions.
3.2 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 3.1, those OCL expressions can be used to identify definitions and uses. Actually, nodes (i.e., postconditions) entail definitions and uses, whereas edges (i.e., guards and preconditions) only entail uses (Section 3.3). For instance, the following operation postcondition can be used to determine whether attributes `attribute1` and `attribute2` are defined, i.e., given a value, during the execution of the operation:

\[
\text{attribute1} = 1 \implies \text{attribute2} = 2
\]

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 3.3.1.
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 4.1) has two conjuncts and four terms, as illustrated in Figure 12.

```plaintext
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
```

Figure 12 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 6, 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 13.

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 13 Templates for connectives](image)

As an example, the compound node (with basic nodes) for `resume()` (Figure 12) is shown in Figure 14.
The EAFG metamodel (Figure 11) is thus modified to account for the structure of compound nodes (Figure 15). 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.
3.3 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 metamodel. Sections 3.3.1 to 3.3.3 discuss several important issues with postconditions written in OCL. Section 3.3.4 presents the metamodel of EAFGs with data flow information. Section 3.3.5 and Section 3.3.6 present descriptions of the rules for determining definitions and uses in nodes and edges. Additional rules are defined for OCL expressions with collection operations (Section 3.3.7). Section 3.3.8 summarizes the rules discussed in previous sections.
3.3.1 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 [7]. 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.

<table>
<thead>
<tr>
<th>Highest level</th>
<th>Intermediate level</th>
<th>Lowest level</th>
</tr>
</thead>
<tbody>
<tr>
<td>if A1 then B1</td>
<td>if A1 then B1</td>
<td>B1 or B2 or B3</td>
</tr>
<tr>
<td>else if A2 then B2</td>
<td>else B2 or B3</td>
<td>// A1 is assumed to be the standard situation</td>
</tr>
<tr>
<td>else B3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Three levels of precision in postconditions, an example from [7]

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.

3.3.2 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 [25]. 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.
3.3.3 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 [23]. 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 [6]. For instance, a transition can trigger the following two actions in sequence: \( v = \text{op1()} \) and \( \text{op2}(v) \). 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 \( \text{op1()} \).

3.3.4 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 16). It is built from the metamodel in Figure 15. 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 3.3.7). A collection operation in OCL takes the form \( \text{collection} \rightarrow \text{op}(\text{parameter}) \). 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 3.3.7). For instance, in the OCL expression \( \text{selfreservation} \rightarrow \text{includes}(\text{reserve}) \), \( \text{selfreservation} \) is an instance of link, and \( \text{includes} \) 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 `PreValue` (or `AfterValue`) of the same model element appears several times in a term, only one instance of `Value` is created. For example, in the OCL expression `att1=f(att2@pre)+g(att2@pre)`, the pre-value of `att2` occurs twice, but only one instance of `Value` for `att2` (with `kind=#PreValue`) is created. `PreValue` and `AfterValue` correspond to different `DataFlow` information. The kinds of
DataFlow information are Def, Cuse, Puse, and Null (though not standard data flow information, Null will be used in Section 3.3.6.1 to define data flow information in frame rules).

Two examples (Appendix A) are given to illustrate the instantiation of the metamodel in Figure 16. The following sections discuss the way definitions, c-uses and p-uses are identified: Section 3.3.5 and Section 3.3.6 do not consider collection operations, i.e., model elements not associated with CollectionTypeOperation. Section 3.3.7 focuses on terms that involve collection operations.

3.3.5 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 3.2). 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 4.1), self.state is p-used in the precondition:

```plaintext
eengineOn()
pre: self.state = #idle
```

Regarding postconditions, 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.

```plaintext
resume()
post:
if (self.cruiseSpeed@pre > 0)
then
  self.throttle = self.cruiseSpeed / CruiseController.st
  + (self.cruiseSpeed - self.speed) / CruiseController.diff
  Term2
else true
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.

```plaintext
Op2()
post:
a = b@pre + 1
Term1
```
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 [22].

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)
3.3.6 Model elements in nodes

Recall that a node contains those OCL expression in a postcondition that are not in the predicate part of an if-then-else expression or on the left part of an implies expression (Section 3.2). 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 is referred to as change specifications and the latter as frame rules [19]. Section 3.3.6.1 and Section 3.3.6.2 discuss data flow information in frames rules and change specifications respectively.

3.3.6.1 Frame rules in nodes

A frame rule specifies what does not change during an operation [19]. A term $T$, is a frame rule if either of the following two conditions holds:

1. $T$ takes the format $a\text{ModelElement} = a\text{ModelElement}@\text{pre}$.

2. $T$ takes the format $\text{coll} \rightarrow a\text{CollectionOperation} = \text{coll}@\text{pre} \rightarrow a\text{CollectionOperation}$, specifying that whole or part (as specified by collection operation $a\text{CollectionOperation}$) of collection $\text{coll}$ is not changed by the operation.

An example for condition 1 is given below. The operation $\text{Op2}()$ 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()
p post: self.b = self.b@pre and self.a = self.c@pre
```

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.

<table>
<thead>
<tr>
<th>Rule 2 Frame rules in nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>context BasicNode</td>
</tr>
<tr>
<td>self.term.isFrameRule = true</td>
</tr>
<tr>
<td>implies</td>
</tr>
<tr>
<td>self.term.valueInTerm.value.dataFlow-&gt;forAll(d:DataFlow</td>
</tr>
</tbody>
</table>

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
```

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
and
self.a = self.b + 1
```

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 3.3.6.2.2.

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.
3.3.6.2 Change specifications

A change specification can contain both pre-values and after-values of model elements. Section 3.3.6.2.1 and Section 3.3.6.2.2 define rules for pre-values and after-values in sequence.

3.3.6.2.1 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 \texttt{op2()} (Section 3.3.6.1), \texttt{self.a = self.c@pre} shows that attribute \texttt{self.a}’s after-value equals \texttt{self.c}’s pre-value. The occurrence of \texttt{self.c@pre} is a c-use of \texttt{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.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Rule 3 Pre-values in nodes} \\
\hline
\textbf{context} BasicNode \\
self.term.isFrameRule = false \\
implies \\
self.term.valueInTerm.value->select(v:Value|v.kind = #PreValue).dataFlow->forall (d:DataFlow|d.kind = #Cuse) \\
\hline
\end{tabular}
\end{center}

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

\begin{verbatim}
Rent(r:Rental)::post 
self.state = #Rented and self.currentRental = r 
\end{verbatim}

Both the pre- and after-values of an \texttt{inout} parameter may occur in a node. When a pre-value of an \texttt{inout} parameter appears in a node, Rule 3 applies, that is, the pre-value is identified as a c-use. For an \texttt{out} or \texttt{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.
3.3.6.2.2 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 3.3.6.1, 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.

```
<table>
<thead>
<tr>
<th>Op5()</th>
<th>Op6()</th>
</tr>
</thead>
<tbody>
<tr>
<td>post:</td>
<td>post:</td>
</tr>
<tr>
<td>self.b = self.b@pre Term1</td>
<td>self.b = self.c@pre Term1</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
</tr>
<tr>
<td>self.a = self.b + 1 Term2</td>
<td>self.a = self.b + 1 Term2</td>
</tr>
</tbody>
</table>
```

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).

```
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->
               select(kind = #AfterValue)->size=1
      implies 4      v.dataFlow.kind = #Def
               and
      5      self.term.valueInTerm.value->select(kind = #AfterValue)->size>1
    implies 7      if fTerm.valueInTerm.value->includes(v)
               then v.dataFlow.kind = #Cuse
       else if dVal->includes(v)
               then v.dataFlow.kind = #Cuse
               else v.dataflow.kind = #Def
        endif
    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.

### 3.3.7 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 [24], as exemplified in Table 3:
Ways to obtain a collection | Examples
--- | ---
Explicitly defined by a literal | Set \{1, 2, 3, 5\}
Obtained by navigation | `context` Title
 | `self.copy`
Obtained by collection operations | `bag->union(bag2)`

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 3.3.7.1 and 3.3.7.2 discuss non-iterative and iterative collection operations respectively.

3.3.7.1 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 3.3.7.1.1 and Section 3.3.7.1.2 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 3.3.5) applies as it is valid for any type of model elements.

When a non-iterative collection operation appears in nodes, both pre-values and after-values of model elements may occur in its parameters. 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) Term1
Post: self.reservation ->includes(reserve) 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.

<table>
<thead>
<tr>
<th>Rule 5 Parameters of non-iterative collection operations in nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context</strong> BasicNode</td>
</tr>
<tr>
<td>self.term.isFrameRule = false</td>
</tr>
<tr>
<td><strong>implies</strong></td>
</tr>
<tr>
<td>self.term.operation-&gt;select</td>
</tr>
<tr>
<td>(o:Operation</td>
</tr>
<tr>
<td>-&gt;forall(d:DataFlow</td>
</tr>
</tbody>
</table>

3.3.7.1.1 Type (1) non-iterative collection operations

Table 4 lists Type (1) collection operations:

<table>
<thead>
<tr>
<th>Set</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set-&gt;union(set2)</td>
<td>sequence-&gt;union(sequence2)</td>
</tr>
<tr>
<td>set-&gt;union(bag)</td>
<td>sequence-&gt;including(object)</td>
</tr>
<tr>
<td>set-&gt;intersection(set2)</td>
<td>sequence-&gt;excluding(object)</td>
</tr>
<tr>
<td>set-&gt;intersection(bag)</td>
<td>sequence-&gt;append(object)</td>
</tr>
<tr>
<td>set-&gt;(set2)</td>
<td>sequence-&gt;prepend(object)</td>
</tr>
<tr>
<td>set-&gt;symmetricDifference(set2)</td>
<td>sequence-&gt;asBag()</td>
</tr>
<tr>
<td>set-&gt;including(object)</td>
<td>sequence-&gt;asSet()</td>
</tr>
<tr>
<td>set-&gt;excluding(object)</td>
<td>sequence-&gt;asSequence()</td>
</tr>
<tr>
<td>set-&gt;asSequence()</td>
<td>sequence-&gt;subsequence(integer, integer)</td>
</tr>
<tr>
<td>set-&gt;asBag()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bag</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bag-&gt;union(bag2)</td>
<td></td>
</tr>
<tr>
<td>bag-&gt;union(set)</td>
<td></td>
</tr>
<tr>
<td>bag-&gt;intersection(bag2)</td>
<td></td>
</tr>
<tr>
<td>bag-&gt;intersection(set)</td>
<td></td>
</tr>
<tr>
<td>bag-&gt;including(object)</td>
<td></td>
</tr>
<tr>
<td>bag-&gt;excluding(object)</td>
<td></td>
</tr>
<tr>
<td>bag-&gt;asSequence()</td>
<td></td>
</tr>
<tr>
<td>bag-&gt;asBag()</td>
<td></td>
</tr>
</tbody>
</table>

Table 4 Type (1) non-iterative collection operations

When a set/bag/sequence appears on 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 3.3.5 and Section 3.3.6). 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 3.3.6), 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.oclIsTypeOf(Type1)}.collection->select
{v:Value|v.kind = #AfterValue}.dataflow->forall
{d:DataFlow|d.kind = #Cuse}

### 3.3.7.1.2 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.

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection-&gt;isEmpty()</td>
</tr>
<tr>
<td>collection-&gt;notEmpty()</td>
</tr>
<tr>
<td>collection-&gt;includes(object)</td>
</tr>
<tr>
<td>collection-&gt;excludes(object)</td>
</tr>
<tr>
<td>collection-&gt;includesAll(collection2)</td>
</tr>
<tr>
<td>collection-&gt;excludesAll(collection2)</td>
</tr>
<tr>
<td>collection-&gt;size()</td>
</tr>
<tr>
<td>collection-&gt;count(object)</td>
</tr>
<tr>
<td>sequence-&gt;at(i)</td>
</tr>
<tr>
<td>sequence-&gt;first</td>
</tr>
<tr>
<td>sequence-&gt;last</td>
</tr>
</tbody>
</table>

Table 5 Type (2) non-iterative collection operations
Note that two of the collection operations above test the numeric properties of a collection: 
\text{collection->size()} returns the number of elements in \text{collection}, and \text{collection->count(object)} counts the number of times the object occurs in \text{collection}.

One question is when these operations appear in contracts, what should be considered as being defined (or used): \text{collection} only or both \text{collection} and \text{collection->size/count(object)}. Our approach is to consider \text{collection} only. Since \text{collection->size} and \text{collection->count(object)} are all properties of a collection, the use or change of properties of \text{collection} implies the use or change of \text{collection}. Therefore, test cases that traverse du pairs/def clear paths (Section 3.4) with respect to \text{collection->size} /\text{collection->count(object)} will be covered by test cases that traverse du pairs/def clear paths with respect to \text{collection}. But the converse is not true. Also, considering \text{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 3.3.5), 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 3.3.6). 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 \text{self.roleName->includes(param)} is part of a postcondition and is not a frame rule. It represents a modification to \text{self.roleName} during the execution of the operation: \text{param} is not an element of \text{self.roleName} before the execution but it is after the execution. Therefore, Rule 4 (Section 3.3.6) 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.

\begin{verbatim}
Reservation::cancel()
Pre: self.state=#Pending or self.state = #Outstanding Term1
Post: self->whenCancelled = TimePoint.currentTime() Term2
       and self.state =#Cancelled Term3
       and if self.copy@pre->notEmpty Term4
          then self.copy@pre.state=#ForRent Term5
             else true Term6
       and self.copy->isEmpty Term7

Member::addReservation(reserve: Reservation)
Pre: not self.reservation->includes(reserve) Term1
Post: self.reservation ->includes(reserve) Term2
\end{verbatim}
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.

### 3.3.7.2 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.

<table>
<thead>
<tr>
<th>Iterative collection operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{collection-&gt;select}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;collect}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;reject}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;forAll}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;exist}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;isUnique}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;sortBy}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;any}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;one}</td>
</tr>
<tr>
<td>\texttt{collection-&gt;iterate}</td>
</tr>
</tbody>
</table>

Table 6 Iterative collection operations

These operations all take an OclExpression as parameter and can follow either of the following syntax [24], where \(v\) is an optional iterator:

\[
\begin{align*}
\text{collection->op(v: Type|OclExpression with v)} \\
\text{collection->op(v|OclExpression with v)} \\
\text{collection->op(OclExpression)}
\end{align*}
\]

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:

- \texttt{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 [11]. 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 clear paths and duplets as discussed in Section 6.

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

**3.3.7.2.1 Iterative collection operations in edges**

When an iterative collection operation appears in an edge, Rule 1 (Section 3.3.5) 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

3.3.7.2.2 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

```plaintext
context BasicNode
let op:Set(Iterative) = self.term.operation->
  select (o:Operation|o.oclIsTypeOf(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)
)
```

3.3.8 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.

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

Table 7 Summary of rules

3.4 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 $(node_0, node_1, ..., node_n)$. 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 \(u\).

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 [2]. The second step is to derive du pairs from def clear paths. This is done by identifying 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 3.1) 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 compound node due to OCL expressions (Section 3.2), not all paths would be evaluated to be true after the execution of a sequence of operations. Consider the Running_accelerate_true node in the EAFG for the Cruise Control statechart. The contracts for event accelerate is in Figure 17, based on which the compound node for Running_accelerate_true in the modified EAFG can be created (Figure 18). Suppose the node Running_accelerate_true is reached from the node Idle_engineOn_true. Because the postcondition for Idle_engineOn_true is \(\text{self.brakepedal}=0 \text{ and self.throttle}=0 \text{ and self.speed}=0 \text{ and self.state}=#\text{running}\), only those terms along the highlighted path in Figure 18 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

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

Figure 17 Pre and postcondition for event **accelerate**

![Diagram of event accelerate](image-url)

Figure 18 Compound node for **Running_accelerate_true**
Last, we do not consider def-clear paths inside compound nodes, i.e., def-clear paths for which the defining and usage nodes are within the same compound node. Such a path would result for instance from a postcondition of the form Term1 and Term2 where Term1 shows the definition of a model element and Term2 shows a usage of that model element: the corresponding EAFG graph would have a node for Term1 followed by a node for Term2, thus resulting in a def-clear path between these two nodes. Such a def-clear path is however a side effect of our decomposition of compound nodes and does not correspond to actual data flow since the postcondition is simply a logical expression that does not specifies any order between its conjuncts (the EAFG artificially introduces one).

### 3.5 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 3.5.1 describes the step of transforming a transition tree to a transition tree EAFG. Section 3.5.2 discusses the step of modifying a transition tree EAFG to represent the feasible path within a node, based on which definitions, uses, and du pairs are derived.

#### 3.5.1 Transforming transition trees into EAFGs

First, each transition tree is transformed into an EAFG using the approach in Section 3.1. 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 19 one of the transition trees (Transition Tree 1) in the Cruise Control case study (Section 4.1) and its corresponding EAFG in Figure 20: To avoid cluttering the diagram, preconditions are not shown.
Recall that once a statechart is transformed into an EAFG, the EAFG is modified to include the branches within a node (Section 3.2). 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 3.4, determining the feasible path within a compound node requires the user input.

The modified transition tree EAFG for Cruise Control Transition Tree 1 (Figure 20) is presented in Figure 21. 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.
After the modified transition tree EAFGs are generated, definitions and uses are determined (Section 3.3), and def clear paths and du pairs contained in the modified transition tree EAFGs are then derived (Section 3.4).

4 CASE STUDIES

We’ve selected two case studies where multiple transition trees can be generated by following Binder’s algorithm [3]. The first case study is a cruise control system [5] that has three transition trees and the second case study is a VCR system that can generate twelve transition trees with Breadth First Search. 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 [13], [14], [16]. 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 Section 3.
4.1 Cruise Control

The Cruise Control system is implemented in Java and contains six classes. The state dependent behavior of the system can be, at analysis level, represented by one class CruiseController (Appendix B). The statechart describing the behavior of CruiseController is shown in Figure 22. Note that in this statechart, transitions have no guard conditions and events have no parameters.

By following Binder’s Algorithm, three transition trees are generated (Figure 23). Note that they are equivalent in the sense that they all cover the round-trip paths in the statechart.

![Statechart for CruiseController](image-url)
### 4.1.1 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 24.

![Distribution of Mutants](image)

Figure 24 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. The next section explains such differences using the data flow information contained in each transition tree EAFG.

### 4.1.2 Comparison of data flow information in each transition tree

Using the approach presented in the previous section, we obtain the information on the number of def clear paths and du pairs in each of the transition trees, as shown in Table 8. To ease discussion, the rest of the paper uses EAFG1 to refer to the EAFG for TT1, EAFG2 for TT2, and EAFG3 for TT3.

<table>
<thead>
<tr>
<th>EAFG</th>
<th>Def clear paths</th>
<th>Du pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAFG1 (TT1)</td>
<td>81</td>
<td>81</td>
</tr>
<tr>
<td>EAFG2 (TT2)</td>
<td>88</td>
<td>88</td>
</tr>
<tr>
<td>EAFG3 (TT3)</td>
<td>81</td>
<td>81</td>
</tr>
</tbody>
</table>

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 4.1.1, 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 25.

![Def Clear Paths vs. Mutation Scores](image)

**Figure 25** Def clear paths/du pairs vs. mutation scores (Cruise Control)

We can see from Figure 25 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.

We 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. The results show that though EAFG1 and EAFG3 cover the same number of def clear paths and du pairs, the sets of def clear paths covered are different and the sets of du pairs covered are different too. The analysis of live mutants missed by the transition trees shows that covering certain du pairs guarantees that certain mutants be detected. For instance, consider the three mutants missed by
both TT2 and TT3 but killed by TT1. 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.

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 26. We can see from Figure 26 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.
A closer look at the specific sets of definitions covered by each EAFG reveals that each of the extra definitions in one EAFG occurs along the same path as a du pair which leads to detecting certain mutants. We, therefore, can draw the conclusion that covering certain definitions also guarantees that certain mutants be detected.

4.2 VCR

The VCR system is implemented in Java using the state design pattern. It contains 29 classes. 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 C, and operation contacts are provided in Appendix D. The statechart of the VCR system at analysis level is presented in Figure 27. Note that the statechart is hierarchical and its flattened equivalent is not shown since it’s too large and complex.

By following Binder’s Algorithm, twelve transition trees are generated with Breadth First Search.
4.2.1 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 28.

Mutation scores for the transition trees are listed in Table 9. 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. Transition trees with the same set of live mutants can be grouped together, so four groups are formed: Group 1 (TT1, TT7), Group 2 (TT2, TT5, TT8, TT11), Group 3 (TT3, TT9), Group 4 (TT4, TT6, TT10, TT12).
In fact, 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.

![Image](image_url)

Figure 28 Mutants distributions across mutation operators (VCR)

<table>
<thead>
<tr>
<th>Mutant Operator</th>
<th>Number of Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOR</td>
<td>38</td>
</tr>
<tr>
<td>ROR</td>
<td>12</td>
</tr>
<tr>
<td>SDL</td>
<td>29</td>
</tr>
<tr>
<td>ICE</td>
<td>20</td>
</tr>
<tr>
<td>CRP</td>
<td>4</td>
</tr>
<tr>
<td>OMR</td>
<td>38</td>
</tr>
</tbody>
</table>

Table 9 Mutation scores for transition trees (VCR)

<table>
<thead>
<tr>
<th>Transition tree</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation score (%)</td>
<td>76</td>
<td>74</td>
<td>74</td>
<td>71</td>
<td>74</td>
<td>71</td>
<td>76</td>
<td>74</td>
<td>74</td>
<td>71</td>
<td>74</td>
<td>71</td>
</tr>
</tbody>
</table>

4.2.2 Comparison of data flow information in each transition tree

Applying the approach in Section 3, 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 10 lists the def clear paths/du pairs information for each EAFG.

<table>
<thead>
<tr>
<th>EAFG</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Du pairs</td>
<td>188</td>
<td>185</td>
<td>185</td>
<td>182</td>
<td>185</td>
<td>182</td>
<td>188</td>
<td>185</td>
<td>185</td>
<td>182</td>
<td>185</td>
<td>182</td>
</tr>
</tbody>
</table>

Table 10 Def clear paths and du pairs in transition tree EAFGs (VCR)

We can see from Table 10 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 11. The relationship between mutation scores and def clear paths for each group is depicted in Figure 29.

<table>
<thead>
<tr>
<th>Group</th>
<th>Def clear paths</th>
<th>Mutation scores (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>188</td>
<td>76</td>
</tr>
<tr>
<td>2</td>
<td>185</td>
<td>74</td>
</tr>
<tr>
<td>3</td>
<td>185</td>
<td>74</td>
</tr>
<tr>
<td>4</td>
<td>182</td>
<td>71</td>
</tr>
</tbody>
</table>

Table 11 Mutation scores and def clear paths/du pairs for each group (VCR)

Figure 29 shows a linear relationship between the number of def clear paths/du pairs and mutation scores. 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.

The analysis of live mutants suggests 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 12.

<table>
<thead>
<tr>
<th>EAFG</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definitions</td>
<td>166</td>
<td>160</td>
<td>160</td>
<td>154</td>
<td>160</td>
<td>154</td>
<td>166</td>
<td>160</td>
<td>160</td>
<td>154</td>
<td>160</td>
<td>154</td>
</tr>
</tbody>
</table>

Table 12 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 13). The relationship between the number of definitions and mutation scores is depicted in Figure 30.

<table>
<thead>
<tr>
<th>Group</th>
<th>Definitions</th>
<th>Mutation Scores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>166</td>
<td>76</td>
</tr>
<tr>
<td>2</td>
<td>160</td>
<td>74</td>
</tr>
<tr>
<td>3</td>
<td>154</td>
<td>74</td>
</tr>
<tr>
<td>4</td>
<td>154</td>
<td>71</td>
</tr>
</tbody>
</table>

Table 13 Mutation score and number of definitions for each group (VCR)

Figure 30 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. It is shown that EAFGs within the same group have an identical set of definitions.
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.

4.3 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. In the VCR case study, transition trees can be divided into groups based on mutation scores and the specific set of missed mutants. EAFGs within one group are found to contain the same number of du pairs/def clear paths and an identical set of definitions.

The result of the case studies suggest both du pairs and definitions are good indicators of the fault detection effectiveness of a transition tree and 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 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 3.3). 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 3.4).

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 section.

5 OPTIMIZING TRANSITION TREES

The motivation is to produce an optimal tree which has maximal fault detection effectiveness and minimal cost. Cost is typically considered as being proportional to test set size [5]. 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 [5].

As suggested by the results of the case studies (Section 4), 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 section is organized as follows: Section 5.1 describes the strategy of building the optimal tree. Section 5.2 reports the results of applying the strategy on the Cruise Control and VCR case studies, and Section 5.3 presents a summary of the strategy.

5.1 Building an optimal tree

Building an optimal tree takes a sequence of steps. These steps are shown in Figure 31 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 $tSet$ (Step 2). Next, we choose a transition tree with the largest number of definitions from $tSet$ as the initial tree (Step 3). Step 4 uses an exhaustive search to
select a set of tree paths from \( \text{tSet} \) 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., \( C_i^p \)), 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 5.3.

### 5.2 Results from case studies

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

#### 5.2.1 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 (\( \text{pathSet3} \) in Figure 31) 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 32).

The optimal tree killed all the mutants missed by TT2 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.
Gi <- Group transition trees with identical sets of definitions

tSet<- select from Gi a transition tree with minimal cumulative length

initialTree <- select from tSet a transition tree with the largest number of definitions

Step1:

Step2:

Step3:

Step4:

pathSet1 <- (tSet \ initialTree).treePaths
pathSet2 <- remove duplicate paths from pathSet1
pathSet3 <- remove paths which doesn't contain any additional defs from pathSet2
augmentingSet<-exhaustive search pathSet3 to get a set of tree paths covering all the defs not already in initialTree, with minimal cumulative length

Step5:

optimalTree<-initialTree U augmentingSet

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.

5.2.2 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 31), which is \{TT1, TT2, TT3, TT4\}. As TT1 has the largest number of definitions, it is taken as the initial tree. pathSet3 (Figure 31) 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 (Appendix E). This leads to kill all the mutants missed by
TT1, 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.

### 5.3 Conclusions

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 5.1, 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.

6 CONCLUSIONS AND FUTURE WORK

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. 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 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.

It is also shown in the case studies that even the most effective tree may miss a significant number of mutants. 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.

There are several limitations of this research. The rest of this section discusses these limitations and possible solutions, and outlines directions for future work.
As discussed in Section 3.2, 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 3.2, 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.

\[
\text{post:} \quad \begin{array}{l}
\text{if}(a@pre > 0 \Rightarrow b=1) \\
\quad \text{then } c = 2 \\
\quad \text{else true} \\
\text{endif}
\end{array}
\]

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 33.
Recall (Section 3.3.7.2) 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 clear paths and du pairs. For instance, in Figure 34, $a$ 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.
There is, however, no easy way to solve this problem. Future work will address this limitation.

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 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.

7 REFERENCES


Appendix A  Instantiation Examples

This Appendix uses two examples to illustrate the instantiation of the modified EAFG metamodel with data flow (Figure 16). The first operation contract (shown below) only contains basic type operations. Figure 35 shows an excerpt of the corresponding metamodel instance.

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

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 3.3.5). 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 3.3.6). 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.

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

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 3.3.7).
Figure 35 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 3.3.5 and Section 3.3.6 do not consider collection operations, i.e., model elements not associated with CollectionTypeOperation. Section 3.3.7 focuses on terms that involve collection operations.
Figure 36 Metamodel instantiation: an example with a collection operation
Appendix B  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
      and
      self.speed = 0
      and
      self.throttle = 0
      and
      self.brakepedal = 0
      and
      self.state = #running

engineOff()
Pre: self.state != #idle
Post: self.state = #idle
and
if (self.cruiseSpeed@pre > 0)
then self.cruiseSpeed = 0
else true
endif
and
if (self.speed@pre > 0)
then self.speed = 0
else true
endif
and
if (self.throttle@pre > 0)
then self.throttle = 0
else true
endif
and
if (self.brakepedel@pre > 0)
then self.brakepedel = 0
else true
endif
on()
pre: self.state != #idle
post: if (self.cruiseSpeed@pre != self.speed@pre)
then self.cruiseSpeed = self.speed@pre
else true
endif
and
if (self.cruiseSpeed > 0)
then self.throttle = self.cruiseSpeed/CruiseController.st+
     +(self.cruiseSpeed - self.speed)/CruiseController.diff
else true
endif
else true
endif
and
if self.state@pre != #cruising
    self.state = #cruising
else true
endif

accelerate()
Pre: self.state! = #idle
Post: 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.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.state@pre = #cruising)
    then self.state = #standby
else true
endif
```
else true
endif

off()
pre: self.state = #cruising
post: self.state = #standby

resume()
pre: self.state = #standby
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
```
Appendix C  Class Diagram for VCR (Analysis Level)

```
+InsertingTape(tape:Tape)
+PowerButton()
+EjectButton()
+RecButton()
+PauseButton()
+FFButton()
+RewButton()
+EndOfTapeEvent()
+StopButton()
+PlayButton()
+PauseButton()

+state : enum {Off, TapeAbsent, RecPaused, Recording, Stopped, FastFF, FastRW, Playing, SlowFF, SlowRew, Paused}
+eventStartTime : long
+currentTime : long
+isTapePresent : bool
+isTapePulledToDrum : bool
+numTimeTapePulled : int
+tape : Tape

VcrController

-tapePosition : int
-tapeWriteProtected : Boolean

Tape

Figure 37 Class Diagram for VCR (at analysis level)
Appendix D  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 Term1
    then self.isTapePresent = false Term2
else true Term3
endif and
if self.state@pre = #Off Term4
    then self.state = self.state@pre Term5
else if (self.state@pre = #Playing or self.state@pre = #Recording) Term6
    then isTapePulledToDrum = false Term7
    and
    numTimesTapePulled = numTimesTapePulled@pre + 1 Term8
    and
    self.state = #TapeAbsent Term9
    and
    if (self.tape.position@pre + NormalSpeed*(CurrentTime-EventStartTime)) Term10
        >self.tape.maxTapePosition
        then self.tape.position = self.maxTapePosition Term11
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 = #TapeAbsent
endif
if self.isTapePulledToDrum@pre = true
    then self.isTapePulledToDrum = false
    and
    numTimesTapePulled = numTimesTapePulled@pre + 1
    and
    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))
else
    endif
endif
else if(self.state@pre = RecPaused or self.state@pre =Paused)
    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
done
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
    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
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
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

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
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 = #Forwading or self.state@pre = #Rewinding)
then self.state@pre = #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
RewButton()

Pre:

Post:
if self.state@pre = #Playing
  then self.state = #Rewinding
    if (self.tape.position@pre - NormalSpeed*(CurrentTime@pre - EventStartTime@pre)) < self.tape.maxTapePosition
      then self.tape.position = self.minTapePosition
    else self.tape.position = self.tape.position@pre - NormalSpeed*(CurrentTime@pre - EventStartTime@pre)
    endif
  endif
else if self.state@pre = #Fowarding
  then self.state = #Rewinding
  and
    if self.isTapePulledToDrum@pre = true
      then
        if (self.tape.position@pre - FassterSpeed*(CurrentTime@pre - EventStartTime@pre)) < self.tape.minTapePosition
          then self.tape.position = self.minTapePosition
        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
        then self.tape.position = self.minTapePosition
      else self.tape.position = self.tape.position@pre - FastestSpeed*(CurrentTime@pre - EventStartTime@pre)
      endif
    endif
  endif

EventStartTime@pre))
endif
def
else if self.state@pre = #Paused
then self.state = # Rewinding
else
if (self.state@pre = #Stopped and self.tape.position>self.minTapePosition)
then self.state = #Rewinding
else true
def
endif
derif
endif
endif
endif
and
eventStartTime = currentTime

FFButton()

Pre:
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))
derif
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))
derif
else
if (self.tape.position@pre + FastestSpeed*(CurrentTime@pre-EventStartTime@pre))
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

if self.tape.position > self.maxTapePosition
    then self.tape.position = self.maxTapePosition
else true
endif

and

eventStartTime = currentTime

else if self.state@pre = #Rewinding
    then self.state = #Stopped
        if self.isTapePulledToDrum@pre = true
            then self.isTapePulledToDrum = false
                numTimesTapePulled = numTimesTapePulled@pre + 1
            else true
                endif
        endif
    endif
endif

else true
endif

endif

PlayButton()

Pre:

Post:
if (self.state@pre = #Forwarding or self.state@pre = #Rewinding)
    then self.state = #Playing
        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
                endif
        endif
    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
Appendix E  Optimal Transition Tree (VCR)

Figure 38 Optimal transition tree (VCR)