SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **20**:177–207
Published online 20 April 2009 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.410

Improving the coverage criteria of UML state machines using data flow analysis

Lionel Briand[1,*,†], Y. Labiche[2] and Q. Lin[2]

[1]*Simula Research Laboratory and University of Oslo, P.O. Box 134, Lysaker, Norway*
[2]*Software Quality Engineering Laboratory, Systems and Computer Engineering, Carleton University, Ottawa, Ont., Canada*

SUMMARY

A number of coverage criteria have been proposed for testing classes and class clusters modeled with state machines. Previous research has revealed their limitations in terms of their capability to detect faults. As these criteria can be considered to execute the control flow structure of the state machine, we are investigating how data flow information can be used to improve them in the context of UML state machines. More specifically, we investigate how such data flow analysis can be used to further refine the selection of a cost-effective test suite among alternative, adequate test suites for a given state machine criterion. This paper presents a comprehensive methodology to perform data flow analysis of UML state machines—with a specific focus on identifying the data flow from OCL guard conditions and operation contracts—and applies it to a widely referenced coverage criterion, the round-trip path (transition tree) criterion. It reports on two case studies whose results show that data flow information can be used to select the best transition tree, in terms of cost effectiveness, when more than one satisfies the transition tree criterion. The results also suggest that different trees are complementary in terms of the data flow that they exercise, thus, leading to the detection of intersecting but distinct subsets of faults. Copyright © 2009 John Wiley & Sons, Ltd.

Received 22 November 2006; Revised 26 February 2009; Accepted 27 February 2009

KEY WORDS: state-based testing; data flow analysis; UML; OCL

1. INTRODUCTION

In the context of object-oriented (OO) development, the Unified Modelling Language (UML) [1] has become the *de facto* standard language for analysing and designing software systems. As the

*Correspondence to: Lionel Briand, Simula Research Laboratory and University of Oslo, P.O. Box 134, Lysaker, Norway.
†E-mail: briand@simula.no

most formalized component of UML, state machines[‡] have long been used as a basis for generating test data for testing the state-dependent behaviour of a class and larger entities such as class clusters or subsystems.

A number of papers have proposed coverage criteria for test data selection from UML state machines, e.g. transition coverage, full predicate coverage, and transition-pair coverage [3], all paths in transition trees (round-trip paths) [4]. Empirical evaluations have revealed that these criteria, i.e. the (possibly numerous) test suites that achieve full coverage, have widely varying cost (e.g. devising and executing test cases) and effectiveness (at finding faults) [5]. The motivation of our research is, therefore, to propose strategies to refine and improve these criteria by providing a mechanism to select the best (most cost effective) test suite among alternative adequate test suites for a given criterion. We especially focus on the transition tree criterion that appears to be a good compromise between transition coverage and transition-pair coverage, which are, respectively, the least and the most expensive criteria experimented so far [5].

Researchers have pointed out, based on theoretical considerations and empirical studies, that data flow testing strategies may be complementary to control flow testing strategies [6,7]. As the abovementioned state-based criteria can be viewed as exercising the control flow structure of the state machine, we are interested in complementing them with data flow analysis. Specifically, we want to identify, among alternative, adequate test suites for a criterion (in our case the transition tree criterion), the test suite that exercises the most state machine data flow, as we make the assumption that this test suite has the highest fault detection cost effectiveness. Note that one important difference with the previously published work on data-flow criteria (e.g. [6,7]) is that we do not intend to use data flow criteria to build test suites. Rather we want to study the data flow coverage of existing test suites to select one of them, which is a much less expensive endeavour.

Some attempts have already been made to derive control and data flow criteria on UML state machines, for example, by analysing uses of variables in guard conditions [7]. These attempts, however, have important limitations such as the partial analysis of guard conditions by only accounting for (in)equalities on variables [7]. We address those limitations by thoroughly examining event/action contracts and guard conditions, that we assume to be written in the Object Constraint Language (OCL) [8]. OCL is a natural choice as it is the standard formal constraint language in the context of UML. Furthermore, its use is advised by current and emerging software paradigms [9], and it is promoted in OO software engineering development methods (e.g. [10]).

In this paper, we propose a *comprehensive* methodology to conduct data flow analysis of UML state machines. It involves three steps: (1) automatically transforming a UML state machine into an event/action flow graph (EAFG) that explicitly specifies the control flow relationships among the events and actions in the state machine, while accounting for control flow inferred from OCL postconditions; (2) identifying definitions and uses from operation contracts and guard conditions with a set of precise, automatable rules; (3) deriving data flow information from the EAFG using a set of well-known algorithms [11]. Our model-based data flow analysis is therefore automatable, thereby making the data flow coverage analysis of existing test suites inexpensive. This is particularly important in the context of UML, model-based testing, an area that is receiving growing interest in the testing community. Among other things, UML, model-based testing will allow certain testing

[‡]Note that the term ‘statechart’ was used up to UML 1.5 [2]. UML 2.0 now uses the term ‘state machine’ [1].

activities (e.g. test planning) to take place earlier during the development process and rely on the existing design models.

Using this data flow coverage analysis methodology, we show how to compare the data flow coverage of different adequate test suites for a given state machine criterion, and we investigate whether the adequate test suite that exercises the most data flow in a state machine 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 an adequate test suite from a set of alternatives. Another objective of our research is to investigate how such data flow information can be used to improve the state machine testing criteria. In this paper we selected the transition tree criterion [4] since (1) empirical results [5] suggest that it is more likely to be useful in practice, and (2) it is a widely referenced and known strategy adapted from the renowned work of Chow [12]. The methodology is applied on two case studies to empirically investigate how data flow information is related to the fault detection effectiveness of transition trees, and how the transition tree criterion can be improved based on the data flow information. Future research will expand this investigation to other criteria.

In summary, we propose a data flow analysis of UML state machines that extends the previous work in three important ways: (i) by analysing both guard conditions and operations contracts to identify data flow, (ii) by thoroughly and comprehensively analysing OCL predicates in those expressions to discover control and data flow information, and (iii) by accounting for all major components of the UML state machine notation thereby allowing the application of our work in a very large number of situations. Our second main contribution is more of a methodological nature. We propose that the model transformations that lead us to identify the data flow information (i.e. the three steps discussed earlier) be carefully and precisely specified with metamodels and OCL rules (between metamodel instances). This way of specifying testing criteria is relevant in the general context of UML model-based testing and is specifically important in the context of this paper since without such precise specifications, it will be difficult for others to compare with our approach and build on it. The third contribution is an empirical evaluation of our approach on two different, representative case studies. Lastly, our findings on the relation between data flow coverage and fault detection effectiveness of transition trees lead us to define a new alternative construction of a transition tree (the round-trip path criterion) that shows to be more effective at finding faults, with only a small increase in cost when compared with previous transition tree construction approaches.

The rest of this paper is organized as follows: Section 2 discusses the related work. Section 3 presents our data flow analysis of UML state machines. Section 4 reports on two case studies. Section 5 presents the conclusions and points out the directions for future work.

Note that this work started with UML 1.5, that is, before the adoption of UML 2.0. But we do not expect the changes in the UML standard to impact the applicability of our approach or the results observed on the case studies. This is further discussed in Section 3.5.3.

2. RELATED WORK

This section looks at the state of the art in the fields related to our research. Sections 2.1 and 2.2 review the coverage criteria based on UML state machines that focus on control and data flow information, respectively. Section 2.3 presents the approach taken by Zweben and Heym [13] to derive the data flow information from postconditions. Section 2.4 states more precisely the objective of this research in the light of existing works.

2.1. Control flow coverage criteria based on UML state machines

Four well-known adequacy criteria for state-based specifications have been defined and we adapt the definitions provided in [3]: The transition coverage criterion requires that all transitions in the state machine be tested; the full predicate coverage criterion requires that the test set includes 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 the first three abovementioned criteria can be automated. As for the complete sequence criterion, authors in [3] 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 [12], 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, and Binder adapted it to UML state machines [4]. This method traverses the state machine to build a transition tree (see [4] for details) and the set of paths (from the tree root to a tree leaf) represent test cases. Binder also proposes an adaptation of the traversing algorithm to account for guard conditions in UML state machines [4]. As the state machine can be traversed either depth or breadth first, different tree structures can be generated, but are all considered equivalent in the sense that they satisfy the criterion.

It is worth mentioning that the criteria discussed so far are based on flat state machines without hierarchical or concurrent structures, though the authors in [14] generate test cases directly from hierarchical state machines[§]. Steps for flattening state machines are provided in [7], though the authors do not address the transformation of OCL expressions[¶] and do not account for the new modelling capabilities of UML 2.0 (e.g. the possibility to reuse state machines [15]).

As one of the most established state-based testing strategies, the W-method has been widely used in protocol testing. Several empirical studies [5,16], however, have revealed some limitations of the strategy in the context of testing software components: Certain kinds of faults cannot be detected by the strategy, suggesting that black-box testing, such as category-partition [17], can play a role complementary to state-based testing, though a manual and expensive one [16]; the round-trip path strategy appears to be a good compromise between transition coverage and transition-pair coverage (or full predicate) in terms of cost-effectiveness, provided that there exists a mechanism to select

[§]The authors adapt a probabilistic method, called statistical functional testing, to the generation of test cases for the all transition criterion. They do not consider the data flow involved in state machines and their work is thus not further described in this paper.

[¶]For example, the OCL constraints that refer to composite states in a hierarchical state machine need to be transformed so that the composite states are replaced by their substates.

effective transition trees when alternative trees are possible [5]. Indeed, alternative test suites from adequate transition trees seem to exhibit significantly different fault detection effectiveness.

2.2. Data flow coverage criteria based on UML state machines

State machine-based criteria are based on the control flow of finite state machines or UML state machines, and have been complemented with data flow criteria [6,18] which examine the associations between the definition of a variable and its uses. To that aim, the authors in [7] first transform the state machine into an Extended Finite State Machine (EFSM) and then transform the EFSM into a flow graph. The authors 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. Though a significant step forward, this approach 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 state machines may contain up to four kinds of events and eight kinds of actions.

To capture a complete set of variable definitions and uses in UML state machines, one can examine the pre and postconditions of events and actions as well as guard conditions. The work done by Zweben and Heym [13], which focuses on testing ADT module specifications, was one of the earliest attempts to examine function postconditions to determine variable definitions and uses. This work is very relevant to our objectives but was performed in a very different context, as described next.

2.3. Deriving data flow information from postconditions

Zweben and Heym [13] focused on applying control flow and data flow testing criteria to ADT modules which are specified in terms of operation pre and postconditions. Edwards [19] extended it to OO software components, where a component has a well-defined interface that is clearly distinguishable from its implementation together with a formal description of its intended behaviour (e.g. a class can be considered a component). In both cases, 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 and Heym [13] 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, e.g. whether the postcondition contains a predicate of the form $x = \cdots$ indicating a definition of variable x. (The same strategy is used in [19].) Definitions and uses are derived from the postconditions of each node according to a set of rules [13]. Then program-based criteria such as all definitions, all uses, and all du paths were applied to the ADT module.

Though very relevant for ADT modules specified by postconditions, this approach has three limitations if one wants to apply it in the context of UML state machines and OCL. First, the precondition of an operation is not taken into account. Second, this approach is not based on state machines or any other state model. Third, the analysis of postconditions to discover uses and definition is incomplete.

Table I. Summary of contributions and limitations of existing works.

Existing works	UML notation	Guard	Event	Action	Operation contract	Analysis of guards or contracts
[7]	Yes	Yes	No	Only assignment	No	(In)Equalities, assignments
[13,19]	No	N/A	N/A	N/A	Only postcondition	Assignments

2.4. Detailed objectives

As previously discussed, the overall objective of our research is to use data flow analysis to refine and improve the test criteria defined on UML state machines with OCL guard conditions and contract expressions for events and actions. Although our approach can apply to any criterion yielding alternative transition sequences, we focus in the current paper on the transition tree criterion since, as discussed above, previous research has shown that it is a reasonable compromise in terms of cost effectiveness. However, when more than one transition tree can be obtained by traversing the state machine, it is then relevant to explore the relationship between the fault detection effectiveness of a tree and the data flow it exercises. In other words, we want to investigate whether the transition tree that 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 which is more likely to detect a larger number of faults.

To summarize, three different pieces of work are relevant to our objectives, as discussed in Sections 2.2 and 2.3. Their contributions and limitations are summarized in Table I. Most importantly, the existing strategies do not fully exploit the information available in guard conditions and operation contracts, either by avoiding some of them in the analysis (e.g. only postconditions are used in [13,19]) or by employing a rudimentary analysis of the predicates they contain (e.g. limited to assignment actions in [7]). Our intent is to extend (with a more extensive support of the UML notation and a more extensive analysis of OCL predicates) and combine them for the purpose of improving the existing criteria based on UML state machines, with a focus on the transition tree criterion in this paper.

3. DATA FLOW ANALYSIS OF UML STATE MACHINES

Our data flow analysis of UML state machines consists of three steps. The first step (Section 3.1) is to transform a UML state machine into an EAFG to facilitate subsequent analysis. The transformation also relies on the analysis of postconditions, assumed to be written in OCL, to infer the control flow of events/actions. The purpose is to explicitly represent the control flow relationship among the events and actions of UML state machines. In the EAFG, nodes represent what executes whereas edges indicate (alternative) flows of execution and under which conditions those flows execute. Operation preconditions and transition guard conditions are associated with edges, and postconditions are associated with nodes.

The elements composing the EAFG (node, edge, condition, etc.) are formalized under the form of a metamodel (Section 3.2). This metamodel (i) summarizes in a concise way the structure of the EAFG, which is important for future extensions or comparisons with our work and (ii) will facilitate the formal definition of data flow identification rules.

By analysing the OCL expressions associated with the nodes and edges of the EAFG, Section 3.3 formally specifies the rules that classify the model elements that appear in those expressions as definitions, c-uses, and p-uses. Those rule specifications are OCL constraints on the EAFG metamodel. This formalization phase is important for different reasons. First, using these OCL expressions, we ensure that our EAFG metamodel contains the information (attributes, associations, etc.) required to derive the data flow information. Second, these OCL rules can be regarded as a precise specification of any tool that implements our approach. Lastly, such a specification is important if we want other researchers and practitioners to precisely understand our approach, replicate our work or build on it. Section 3.4 discusses the notions of definition–use pairs and definition clear paths, which are derived from an EAFG by using a well-established algorithms proposed for compiler optimization [11]. Section 3.5 provides additional discussions on the construction of an EAFG. Section 3.6 applies the approach to the transition tree criterion.

We assume that the state machines have already been flattened and OCL constraints have been transformed accordingly. Such a process, however, can be fully automated, and the resultant flattened state machine can be regarded as an intermediate form, being used by the test case generation algorithms only, rather than being visualized by modelers or testers. Additionally, we assume that the different UML views, including the state machine view and the OCL constraints view, are consistent (e.g. there is no semantic error between OCL and state diagrams). Ensuring that UML views are consistent is an interesting problem but is out of the scope of this paper.

3.1. Transforming UML state machines into EAFGs

Our analysis of data flow information in state machines is based on an *EAFG* that is a directed graph where nodes denote postconditions of events or actions in the state machine, and edges indicate, under the form of predicates, the conditions under which the successor node may be invoked after the predecessor node. Section 3.1.1 presents some initial principles underlying our EAFG. Given those principles, Section 3.1.2 shows how we account for the complete UML state machine notation, and Section 3.1.3 shows how we account for additional control flow as suggested by event/action postconditions. Section 3.1.4 summarizes the differences between our flow graph and the ones used in the literature.

3.1.1. Principles of an EAFG

First, the state machine needs to be modified to facilitate further analysis. Entry and exit actions are moved to transitions, and internal transitions are promoted to external transitions according to standard rules [20]. Figure 1 (parts (a) and (b)) shows an example of such a transformation for state `s4`: its entry (resp. exit) action is moved to incoming (resp. outgoing) transitions. The figure also shows how an internal transition is transformed (i.e. in state `s4`). This transformation is performed for analysis purposes only and the intermediate state machine (Figure 1(b)) does not need to be visualized by the tester.

The intent of our EAFG is to eventually associate p-uses with edges and c-uses as well as definitions with nodes, similarly to standard control and data flow graphs [21]. The predicate for

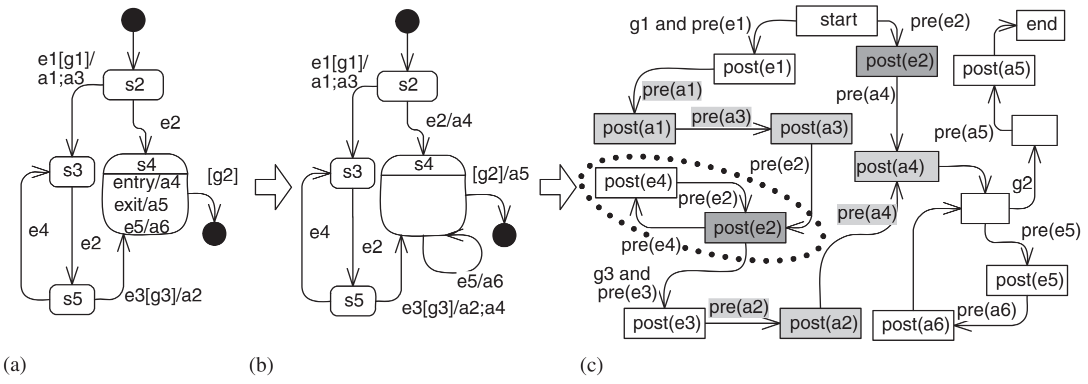


Figure 1. (a) State machine; (b) transformed state machine; and (c) corresponding EAFG.

an edge leading to an event's postcondition is the conjunction of the event precondition and the guard condition of the corresponding transition: the result described in the event's postcondition is obtained when both the guard and the event's precondition are true. Since actions do not have guard conditions, the incoming edges for actions are only associated with preconditions. Since an action on a transition is executed when the transition fires, the edge labelled by the action's precondition links the event's postcondition's node to the action's postcondition's node. When more than one action appears in a transition (i.e. there is an action sequence), we analyse them according to their written order since they are usually assumed to be independent and their order therefore does not matter [20]. Each action in an action sequence in a state machine is denoted by a separate node in the EAFG, and is connected to other actions according to their written order. Figure 1(b) shows two examples of action sequence in the state machine: transition from s2 to s3, transition from s5 to s4. The corresponding nodes and edges in the EAFG of Figure 1(c) are highlighted in light grey. In Figure 1(c), pre(x) and post(x) denote the pre and postconditions of event (or action) x, respectively.

When an event triggers multiple transitions, each occurrence of the event is represented by a separate node in the EAFG. In Figure 1(b), event e2 triggers two transitions: from s3 to s5 and from s2 to s4. In the corresponding EAFG, post(e2) occurs twice (highlighted in dark grey). Note that an EAFG has a start node, and an end node when the state machine has a final state. Furthermore, an EAFG can have cycles when there are cycles in the state machine, as illustrated in Figure 1: cycles in the state machine involving events e2 and e4 (states s3 and s5); nodes labeled post(e2) and post(e4) are in a cycle with edges pre(e2) and pre(e4) in the EAFG (highlighted with a dotted line ellipse).

Two nodes in Figure 1(c) do not show any postconditions (empty nodes on the right of the figure). One is due to the change event from state `s4` to the final state (Figure 1(a)), which is further discussed in the next section. The other illustrates the transformation of self transitions: internal transition `e5/a6` in state s4 in Figure 1(a) has been promoted to a regular (self) transition in Figure 1(b).

Table II. Mapping between UML state machine elements and EAFG.

UML model element		EAFG mapping
Name	Notation	
Call event	$\boxed{\text{state1}} \xrightarrow{\text{aCallEvent}} \boxed{\text{state2}}$	$\downarrow \text{pre(aCallEvent)}$ $\boxed{\text{post(aCallEvent)}}$
Signal event	$\boxed{\text{state1}} \xrightarrow{\text{signalEvent}} \boxed{\text{state2}}$	$\downarrow \text{pre(signalEvent)}$ $\boxed{\text{post(signalEvent)}}$
Change event	$\boxed{\text{state1}} \xrightarrow{\text{when(a boolean expression)}} \boxed{\text{state2}}$	$\downarrow \text{a boolean expression}$ $\boxed{}$
Time event	$\boxed{\text{state1}} \xrightarrow{\text{a time expression}} \boxed{\text{state2}}$	$\downarrow \text{a time expression}$ $\boxed{}$
Call action	$\boxed{\text{state1}} \xrightarrow{\text{event/action}} \boxed{\text{state2}}$	$\downarrow \text{pre(action)}$ $\boxed{\text{post(action)}}$
Send action	$\boxed{\text{state1}} \xrightarrow{\text{event/sendAction}} \boxed{\text{state2}}$	$\downarrow \text{pre(actionHandler)}$ $\boxed{\text{post(actionHandler)}}$
Assignment action	$\boxed{\text{state1}} \xrightarrow{\text{event/assignmentAction}} \boxed{\text{state2}}$	$\downarrow$ $\boxed{\text{post(assignmentAction)}}$
Create action	$\boxed{\text{state1}} \xrightarrow{\text{event/new ClassName}} \boxed{\text{state2}}$	$\downarrow \text{pre(constructor)}$ $\boxed{\text{post(constructor)}}$
Destroy action	$\boxed{\text{state1}} \xrightarrow{\text{event/obj.destroy()}} \boxed{\text{state2}}$	$\downarrow \text{pre(destructor)}$ $\boxed{\text{post(destructor)}}$

3.1.2. Mapping of events, actions, and activities to the EAFG

A UML state machine may contain four kinds of events and eight kinds of actions [20]. Table II summarizes how each event and action kind is transformed into elements of an EAFG, keeping in mind that, as mentioned earlier, nodes of the EAFG represent things that are performed (executed) whereas edges of the EAFG represent the flow and under which conditions those things are performed. The reader interested in a more complete discussion of the mapping is referred to [22].

The discussion in Section 3.1.1 addressed call events and call actions. A call event becomes a node holding the event's postcondition and an edge holding the event's precondition (possibly with a guard condition). In the case of a signal event, ultimately, the event is realized by an operation, usually called the event handler. The transformation is then similar to a call event, using the event handler[‖]. A change event or a time event specifies a condition only and is thus transformed into a node holding an empty postcondition and an edge holding that condition.

As shown in Section 3.1.1, a call action becomes a node holding the action's postcondition and an edge holding the action's precondition[**]. A send action denotes non-determinism but, to account for as many data flow interactions as possible, we assume that the signal will always be

[‖]The handler can be an operation, with the same name as the signal, declared in the class or interface that accepts the signal (such an operation has the stereotype <<`signal`>>) [23]. For our purpose, this is equivalent to a call event (the operation called is the handler). (This case is represented in Table II.) Another solution is to consider that the handler of the signal event is the action triggered by the transition [20]. For our purpose, this is equivalent to a call event where the pre and postconditions of the operation are empty.

[**]Note that the case when the target of the call action is a set of objects does not make a difference since we account for all potential data flows: whether there is one target object or ten does not change the potential data flows.

received and processed by the receiver handler. A send action is therefore transformed like a call action, using the handler. An assignment action is transformed like a call action since it can be considered as an action with an empty precondition and a postcondition (the assignment) that is the assignment itself in which the `@pre` (OCL) postfix is placed after each model element occurring on the right-hand side of the assignment expression. For a create (resp. destroy) action, we use the pre and postconditions of constructor (resp. destructor).

Note that a terminate action results in the self-destruction of the owning object of the state machines [1]. Since the owning object does not exit after a terminate action, we are not interested in such kind of actions. We do not 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.

An activity is an ongoing non-atomic execution while an object is in a state [20]. An activity, either a single activity, a sequence of activities, or a sequence of actions, either completes its execution and then a completion event is sent, or is interrupted by the arrival of an event. Since we cannot foresee the occurrence of an event and thus when the activity 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, this translates into `(n−1)*m` possible interruptions. Considering that do transitions normally represent, according to recommended practice, ongoing activities that do not change the state and do not participate in any data flow within the state machine, we ignore them in the construction of the EAFG.

3.1.3. Accounting for OCL expressions

If we assume, like for any other logical expression expressed in a conjunctive normal form, that OCL postconditions are composed of *conjuncts*, those conjuncts 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 in Figure 3(a) has two conjuncts and four terms. This decomposition is used to distinguish between *compound nodes* and *basic nodes* in the EAFG. An operation postcondition is associated with a compound node, and a compound node is composed of basic nodes, among which are a start node and an end node. The sequence of basic nodes, linked by edges in a compound node, corresponds to the order of appearance of the corresponding conjuncts in the OCL postcondition of the compound node: 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 templates in Figure 2. In these templates, nodes are associated with terms connected by `xor` or `or` operators, terms in the `then` or `else` parts of `if-then-else` operations, or on the right part of `implies` expressions. The predicate parts of `if-then-else` or `implies` expressions are associated with edges. In case those nodes also contain one or more of the four connectives, they are further decomposed using the same templates[††]. The rationale is that those operators are used in OCL expressions to specify alternative operation results, and therefore suggest some control flow in the

[††]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.

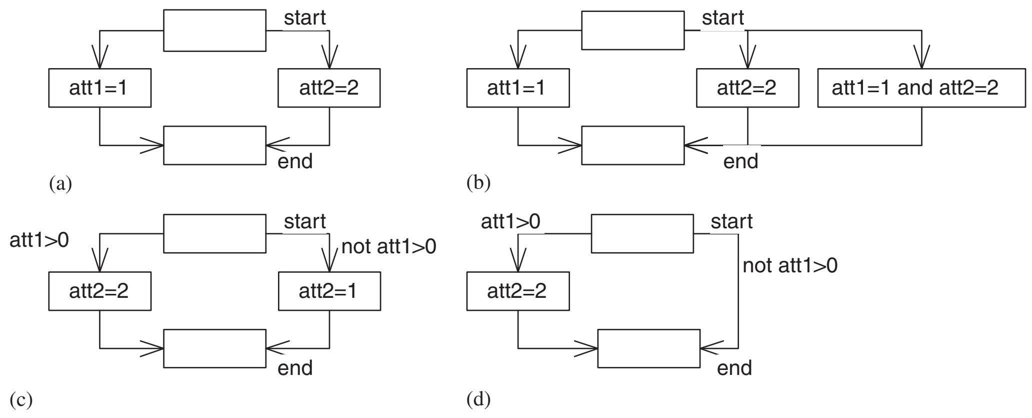


Figure 2. Templates for connectives: (a) template for the xor operator (post: att1 = 1 xor att2 = 2); (b) template for the or operator (post: att1 = 1 or att2 = 2); (c) template for the if-then-else operator (post: if att1>0 then att2 = 2 else att2 = 1); and (d) template for the implies operator (post: att1>0 implies att2 = 2).

```
anOperation()
pre: …
post:
  if ( Conjunct one
    self.a@pre > self.b                     Term1
  )
  then
    self.b = self.a / self.c +
    (self.a@pre - self.d) / self.e          Term2
  else
    true                                    Term3
  endif
  and                                 Conjunct two
  self.state =#StateA                       Term4
```

(a)

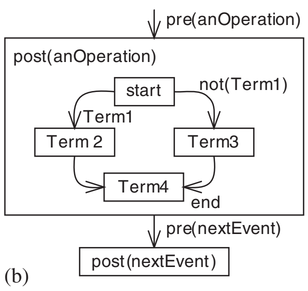


Figure 3. Example of terms in a postcondition (a), with corresponding EAFG (b).

operation execution. 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). For instance, in postcondition `att1=1 implies att2=2`, `att2` is given a value under a specific condition: `att1=1`. 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`. Although this postcondition seems underspecified, we have to account for such situations in practice. The level of details of OCL expressions in postconditions is further discussed in Section 3.5.1.

As an example, the compound node for `anOperation()` (Figure 3(a)) is shown in Figure 3(b). For the sake of brevity, the terms' OCL expressions have been replaced by term numbers.

Note that the Boolean expression on the left of an `implies` expression and the predicate part of an `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 that those OCL expressions only contain uses of model elements (Section 3.3) and do not suggest any control flow during the operation's execution: 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. Although a good specification practice that should be encouraged, this however constitutes 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 will be further investigated in future work.

3.1.4. Comparison with flow graphs in the literature

Our EAFG control flow graph fundamentally differs from flow graphs described in the related work (i.e. [7,13,19]) in several ways. The first main difference is the sources of information used to build the flow graph: operation contracts [13,19], and finite state machines [7], as opposed to a UML state machine. The flow graphs also differ in structure: no notion of compound node to account for control flow possibly suggested in postconditions [7,13,19]; state and transitions become nodes in [7] and edges show the possible flow of control between the nodes as specified by states changes through transitions. A third important difference is that we build our flow graph with full support[‡‡] for the UML notation, which is now the *de facto* standard for the analysis and design of OO software systems.

3.2. EAFG metamodel

We formalize the structure and well formedness of EAFGs as described in the previous section by means of the metamodel in Figure 4. The metamodel defines precisely the form an EAFG can take, i.e. how both control and data flow from a state machine are represented. Metaclasses for the data flow aspects are highlighted in Figure 4 and discussed next.

An `EAFG` consists of `Nodes` and `Edges`. A `Node` is either a `BasicNode` or a `CompoundNode`. A `CompoundNode` can be an `EventNode`, `ActionNode`, or `ActivityNode`. A `CompoundNode` consists of a set of basic nodes (rolename `theNodes`), among which are a start `BasicNode` and an end `BasicNode`[§§]. Nodes have predecessors and successors. 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 more `Terms` depending on whether it is a basic or compound node and the complexity of the postcondition associated with the compound node. An `Edge` may contain zero or one `Precondition` and zero or one `Guard`. `Postcondition`, `Precondition`, and `Guard` are subclasses of abstract class `Constraint` which is associated with `Terms`.

[‡‡]With a few minor exceptions that we discussed earlier, and that we showed do not introduce practical limitations to our approach.

[§§]In the context of `CompoundNode`, we have: `self.theNodes->includesAll(self.start->union(self.end))`.

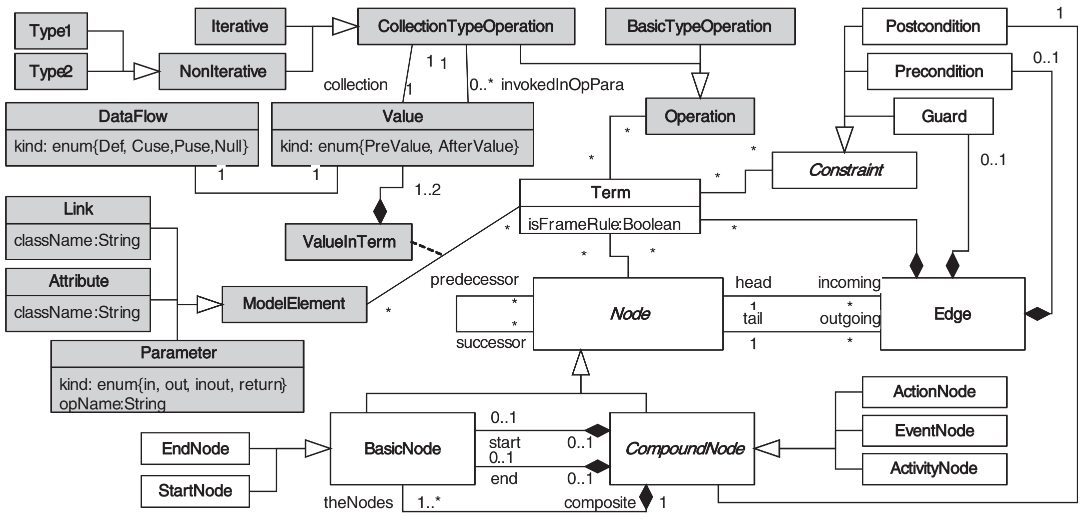


Figure 4. EAFG metamodel.

As mentioned previously, OCL expressions are assumed to be in conjunctive normal form (or transformed into it) and are decomposed into conjuncts terms, that is, OCL expressions without connectives. A `Term` is associated with `ModelElements` and `Operations`. A `ModelElement` refers to the `Attributes` and `Links` of an instance of a class, and `Parameters` of a context operation. (These are the only elements in OCL terms since, as further discussed in Section 3.3.1, all `let` expressions and query operations are replaced with their corresponding definitions.) A term can also contain operations which can be either operations on OCL basic types or OCL collection types (classes `CollectionTypeOperation` and `BasicTypeOperation`, respectively), which is further classified into iterative and non-iterative operations (e.g. operation `size()` vs operation `forAll()`). `NonIterative` collection operations are of two different types that we further discuss in Section 3.3.5. An OCL collection operation takes the form `collection->op(parameter)`. The `collection` of a `CollectionTypeOperation` refers to the `Link` (instance of association) of an instance of a class. The `ModelElements` involved in the parameter of a `CollectionTypeOperation` can be the parameters of the context operation or can refer to the links and attributes of the context object. For instance, in the OCL expression `self.roleName->includes(obj)`, `self.roleName` is an instance of `Link`, `includes` is an instance of a subclass of `CollectionTypeOperation`, and `obj` is an instance of `ModelElement` involved in the parameter of the collection operation.

Association class `ValueInTerm` is associated with 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`. A prevalue is the value of a model element at the start of an operation, whereas an aftervalue is the value of a model element at the end of an operation. Pre-values appear in precondition, and postcondition when postfixed by symbol `@pre`, and postvalues typically appear in postconditions. 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 prevalue of `att2` occurs twice, but only one instance of `Value` for `att2` (with `kind=#PreValue`) is created for this term. `PreValue` and `AfterValue` correspond to different `DataFlow` informations. The kinds of `DataFlow` information are `Def`, `Cuse`, `Puse`, and `Null` (though not standard data flow information, `Null` will be used to specify that a value is not involved in any data flow).

An in parameter can always be considered as a prevalue, whereas an out (or result) parameter can always be considered an aftervalue. This is specified as an OCL constraint on our EAFG metamodel:

```
Context : Parameter
self.kind=#in implies self.valueInTerm.value->forAll (kind = #PreValue)
and
self.kind=#out or self.kind=#result implies self.valueInTerm.value
->forAll (kind = #AfterValue)
```

3.3. Determining definitions and uses in edges and nodes

In this section we specify, using OCL rules on our metamodel, how definitions and uses of model elements can be identified from OCL expressions. Section 3.3.1 discusses how OCL expressions are transformed to remove query operations and local definitions. The following sections use our EAFG metamodel to formally define the rules in OCL for determining definitions and uses in nodes and edges. Section 3.3.2 discusses edges, whereas Sections 3.3.3 and 3.3.4 discuss nodes, i.e. OCL expressions in postconditions that are not in the predicate part of an `if-then-else` expression or on the left part of an `implies` expression (Section 3.1.3). Specifically, a term in a node (i.e. in a postcondition) can describe what is changed during the operation, as well as what is not changed and remains true after the operation, which are referred to as *change specifications* and *frame rules* [24], respectively. Section 3.3.3 discusses frame rules and Section 3.3.4 discusses change specifications. These rules are sufficient when an OCL expression does not contain any collection operation. However, when collection operations are involved, these rules need to be extended, a topic addressed in Section 3.3.5. Note that the rules presented in Sections 3.3.2 to 3.3.5 for identifying definition and uses can be fully automated (through parsing of OCL expressions). These sections focus on the main principles and more details, especially with concrete and/or abstract examples illustrating each situation, are available in [22].

3.3.1. Query operations and local definitions

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 computed 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 operation with the expression it assigns to `result` in its postcondition.

Furthermore, `let` expressions in OCL allow modelers to define a variable or operation that can be used (possibly several times) in a constraint [8]. To simplify the analysis, every occurrence of variables or operations defined by `let` expressions is replaced by the defining expression.

Local variables may also be defined and used in a transition's action sequence [26]. For instance, a transition can trigger the following two actions in sequence: `v=op1()` and `op2(v)`. In such a case, every use of the variable is replaced with the `result` part of the postcondition of the operation that defines it. In our example, `v` is replaced with the `result` part of the postcondition of `op1()`.

3.3.2. *Model elements in edges*

Recall that an edge in an EAFG is possibly labelled 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 3.1.3). Since these constraints are typically implemented as conditional statements, the model elements that appear in an edge are classified as p-uses, whatever the model element type (link, attribute, or parameter), which is stated in Rule 1 below. For example, in an OCL precondition of the form `self.state = #idle`, `self.state` is p-used. This also applies to parts of postconditions that label edges. For example, in the postcondition of Figure 3, the prevalue of `self.a` in `Term1` is a p-use of `self.a`. Similarly, the aftervalue of `self.b` (in `Term1`) is considered a p-use.

Rule 1 Model elements in edges
```
context Edge
self.term.valueInTerm.value.dataFlow->forAll(kind = #Puse)
```

3.3.3. *Frame rules in nodes*

A frame rule specifies what does not change during an operation [24]. A term T, is a frame rule if either of the following two conditions holds:

1 T takes the format `aModelElement = aModelElement@pre`.
2 T takes the format `coll->aCollectionOperation = coll@pre->aCollectionOperation`, specifying that whole or part (as specified by collection operation `aCollectionOperation`) of collection `coll` is not changed by the operation.

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, a change specification will certainly assert such information. Therefore, frame rules do not contain any definition or use, which is stated in Rule 2 below. Frame rules may however help the identification of definitions and uses in postconditions as discussed next.

Rule 2 Frame rules in nodes
```
context BasicNode
self.term.isFrameRule = true
implies self.term.valueInTerm.value.dataFlow->forAll(kind = #Null)
```

3.3.4. *Change specifications in nodes*

A change specification can contain both pre-values (Section 3.3.4.1) and after-values (Section 3.3.4.2) of model elements.

3.3.4.1. Pre-values in nodes. The occurrence of a prevalue in a change specification is considered to be a c-use, as this states that the operation uses the value to set another element's value. For example, in the postcondition of Figure 3, `Term2` shows that attribute `self.b`'s `after-value` is changed using `self.a`'s `pre-value`. The occurrence of `self.a@pre` is a c-use of `self.a`.

Rule 3 below states that the pre-values of model elements appearing in a node are considered to be c-uses provided that the terms in which these pre-values occur are not frame rules.

Rule 3 Pre-values in nodes

```
context BasicNode
self.term.isFrameRule = false
implies self.term.valueInTerm.value->select(kind = #PreValue).dataFlow
                                   ->forAll(kind = #Cuse)
```

Since an `in` parameter in a node can always be considered to be a pre-value, Rule 3 applies, that is, an `in` parameter in a node is a c-use. Similarly, the pre-value of an `inout` parameter in a node is a c-use (Rule 3 applies). `out` and `return` parameters in nodes are after-values, which are discussed next.

3.3.4.2. After-values in nodes. Two cases can be identified when an after-value, say `v`, occurs in a node (i.e. in a term):

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.

Case 2: `v` is not the only after-value in a term t_1. If the intent of the designer is to specify that `v` is not changed by the operation (i.e. the complete postcondition), then there must be a frame rule with respect to `v` in another term t_2 (otherwise, the term of the postcondition is ambiguous[¶¶]). Alternatively, if the intent is to specify that `v` is changed, there is a change specification in a term t_2 that defines `v`. Therefore, if `v` appears in another term t_2 that is either a frame rule or a change specification for `v`, then the occurrence of `v` in term t_1 is a c-use of `v`; otherwise, term t_1 is a change specification that defines `v`. When looking for the potential frame rule or change specification for `v`, we need to focus on the control flow paths in the compound node that involve the term of interest since the operation may modify `v` in some of those paths, and leave `v` unchanged in other paths.

Below are three examples. In the postcondition of `Op4()`, `Term2` has two after-values, `self.a` and `self.b`. `self.b` also appears in `Term1`, which is a frame rule. Therefore, in `Term2`, `self.b` is a c-use whereas `self.a` is a definition. In the postcondition of `Op5()`, `self.b` is defined in `Term1`. Hence, its appearance in `Term2` is again a c-use and the occurrence of `self.a` is again a definition. In the postcondition of `Op6()`, `Term1` (resp. `Term2`) is a frame rule for `a` (resp. `b`). In this particular case, the intent of the designer was to specify that depending on condition `self.c` whether the value of `a` is changed or not. It is changed when `self.c` is false, in which case `a` is set a value using `b`: in `Term3`, `self.a` is a definition and `self.b` is a c-use. In such cases where the control flow of the postcondition is complex, the search for a frame rule

[¶¶]For instance, a postcondition that reads `self.a=self.b+1` is ambiguous since we cannot decide whether `a` and `b` are defined or used. Instead, if the postcondition reads `self.a=self.a@pre` and `self.a=self.b+1` (the first part of the conjunct being a frame rule for `a`), we know `a` is not changed but `b` is changed in the second conjunct.

involving v when v is not the only after-value in a term is limited to a consecutive set of terms, e.g. within the then or else block of an if-then-else statement (a similar search can be made in case of implies, or, and xor statements). Such a simplification is necessary to avoid complex data flow analysis in each feasible control flow path of the postcondition. This limitation is further discussed below and could be addressed by more complex analyses in the future if we find it practically justifiable.

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

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

```
Op6()
post:
  if (self.c) then
    self.a=self.a@pre        Term1
  else
    self.b = self.b@pre      Term2
    and
    self.a = self.b + 1      Term3
  endif
```

Rule 4 summarizes the cases when an after-value of a model element appears in a node. Operation nodesInBlock() of class BasicNode returns the set of BasicNode instances (i.e. terms) that belong to the same block of consecutive BasicNodes of an if-then-else, implies, or, xor statement (recall the templates of Figure 2) as the BasicNode instance on which the operation is called (e.g. when called on the BasicNode for Term3 in Op6()'s postcondition, the operation returns Term2 and Term3 but not Term1). fTerm (line 1) refers to all the terms of the same block (as self) that are frame rules, and nfTerm (line 2) refers to all the terms of the same block (as self) that are not frame rules. dVal (line 3) refers to those values in nfTerm that are definitions. Lines 8 and 9 correspond to the case when v is the only after-value in a term and v is identified as a definition. Lines 11–19 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 (lines 13 and 14), or defined in another term in the compound node (lines 15 and 16), otherwise it is a definition (line 17).

Rule 4 After-values in nodes

```
context BasicNode
1  let fTerm:Set(Term)=self.nodesInBlock()->select(isFrameRule=true)
2  let nfTerm:Set(Term)=self.nodesInBlock()->select(isFrameRule=false)
3  let dVal:Set(Value)=nfTerm.valueInTerm.value->select(dataflow.kind =#Def)
4  in
5   self.term.isFrameRule = false
6   implies (
7     self.term.valueInTerm.value->select(kind=#AfterValue)->forAll(v:Value|
8       self.term.valueInTerm.value->select(kind = #AfterValue)->size=1
9       implies  v.dataFlow.kind = #Def
10      and
11      self.term.valueInTerm.value->select(kind = #AfterValue)->size>1
12      implies
13        if fTerm.valueInTerm.value->includes(v)
14        then v.dataFlow.kind = #Cuse
15        else if dVal->includes(v)
16             then v.dataFlow.kind = #Cuse
17             else v.dataflow.kind = #Def
18             endif
19        endif
20     )
21   )
```

The search of frame rules in a block, as performed by operation `nodesInBlock()`, is limited as we do not consider nested conditional structures (e.g. nested `if-then-else` statements). This is admittedly a restriction since by doing so we may not discover all the c-uses and definitions. However, we consider this to be a reasonable approximation since (1) nested conditional structures rarely occur in postconditions (e.g. in the two representative case studies we discuss in Section 4, no such situation was encountered) and (2) we believe that when nested conditional statements are used, the terms that would be used to identify definitions following our approach—i.e. a term with more than one after-value and term(s) with frame rule(s)—would most likely appear in the same block instead of different blocks (e.g. the frame rule in one block and the term with multiple after values in another block).

3.3.5. Contracts with collection operations

A collection can be defined explicitly by a literal (e.g. `Set {1, 2, 3, 5}`), obtained by navigation (e.g. `self.roleName`), or operations on collections (e.g. `c1->union(c2)` where `c1` and `c2` are two collections) [8]. Collection operations can be broadly divided into two categories: Iterative collection operations (Section 3.3.5.2) and non-iterative collection operations (Section 3.3.5.1). Iterative collection operations in this work refer to those operations that iterate over collection elements and take an `OclExpression` as parameter. We make this distinction because iterative and non-iterative collection operations involve different data flow information and hence require separate rules.

3.3.5.1. Non-iterative collection operations. Some non-iterative collections require a parameter (either a pre- or after-value) that does not suggest any change to the model element(s) involved in the parameter. These parameters are simply used when evaluating the collection operation. Therefore, when the collection operation appears in an edge, parameters are p-used and Rule 1 applies, and when it appears in a node, parameters are c-used. In this latter case, a new rule, Rule 5 below, is required since Rule 3 only applies to pre-values.

Rule 5 Parameters of non-iterative collection operations in nodes
```
context BasicNode
self.term.isFrameRule = false
implies self.term.operation->select(o:Operation|o.oclIsTypeOf(NonIterative))
            .involvedInOpParam.dataflow->forAll(d:DataFlow|d.kind = #Cuse)
```

The collection, on which the non-iterative collection operation is used, can be involved in some data flow. When a non-iterative collection operation appears in an edge, the collection is considered a p-use (Rule 1). When the pre-value of the collection appears in a node, it is considered a c-use, provided that the term in which it occurs is not a frame rule: Rule 2, and Rule 3 apply.

When the after-value of the collection appears in a node, we need to distinguish between the collection operations that return a new collection (e.g. `union(...)` or `asSet(...)`), referred to as Type (1) operations, and the operations that constrain some characteristic of the collection and return a Boolean, Integer, or an element of the collection (e.g. `size()`, `includes(...)`), referred to as Type (2) operations. (See [22] for the complete list of Type (1) and Type (2) operations.)

When the after-value of a collection in a Type (1) operation appears in nodes, it is identified as a c-use (provided that the collection does not appear in a frame rule) since 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 only used. A new rule, Rule 6 below, specifies this situation.

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

```
context BasicNode
self.term.isFrameRule = false
implies self.term.operation->select(oclIsTypeOf(Type1)).collection
             ->select(v:Value|v.kind = #AfterValue).dataFlow
             ->forAll(d:DataFlow|d.kind = #Cuse)
```

Unlike Type (1) non-iterative collection operations, Type (2) non-iterative collection operations specify possible modifications to the collection on which they are applied. For instance, `self.roleName->includes(param)` in a postcondition specifies a modification to `self.roleName` during the execution of the operation: `param` is not an element of `self.roleName` before the execution but it is after the execution. Therefore, Rule 4 (Section 3.3.4) is used to determine definitions and c-uses of the collection.

3.3.5.2. Iterative collection operations. Iterative collection operations in this work refer to those operations that iterate over the elements of a collection and take an `OclExpression` as parameter (e.g. `select()`, `exist()`). `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.

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 [6].

With iterative collection operations, 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. However, those collection operations are typically used in postconditions to specify changes to elements of the collection (definitions) or uses of those elements (e.g. to define a new collection). Our approach is therefore to consider that all the objects in the collection are defined (or used). This is a conservative approach as it will not miss any definition (or use) but may lead to the identification of definitions (or uses) that do not actually exist. This approach has an impact on the identification of def clear paths and du pairs which will be investigated in future work.

As for the previous cases, when an iterative collection operation appears in an edge, Rule 1 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`.

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 `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->select` is used: `self.roleName->select(att1=param)->size=self.roleName@pre->select(att1 =param)->size+1`. Hence, we cannot simply apply Rule 3, which is for pre-values only, and we use Rule 7 below instead.

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)
in
self.term.isFrameRule = false
implies (
    op.collection.dataFlow.kind = #Cuse)
    and op.involvedInOpParam.dataFlow->forAll(kind = #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 changes to a collection and the creation of objects (through defining the attributes and links of the objects). This is the purpose of Rule 8.

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

```
context BasicNode
let op:Set(Iterative) = self.term.operation->select(o:Operation|
      o.oclIsTypeOf(Iterative)  and  o.collection.kind = #AfterValue)
in
self.term.isFrameRule = false
implies (
  op.collection.dataFlow.kind = #Def)
  and  op.involvedInOpParam->select(kind = #AfterValue).dataFlow
                     ->forAll(kind = #Def)
  and  op.involvedInOpParam->select(kind = #PreValue).dataFlow
                     ->forAll(kind = #CUse)
)
```

3.4. Identifying definition–use pairs and definition clear paths in EAFG

A definition clear path (def clear path) with respect to a model element `e` is a path in a flow graph (in our case the EAFG) that starts at a node where `e` is defined and ends at a node (or edge) where `e` is used and `e` is not redefined on the path. 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 (or edge) 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 [11]. 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 removing duplicates.

Note that 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. Those paths are a side effect of our decomposition of compound nodes and do not correspond to the actual data flow since the postcondition is simply a logical expression that does not specify any order between its conjuncts (the EAFG artificially introduces one).

3.5. Discussion

We discuss in this section a number of issues potentially impacting the practical use of our approach.

3.5.1. 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 [27]. 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.

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. Assessing the impact of the precision of postcondition on the data flow analysis results will be the subject of future work.

3.5.2. Infeasible paths

As for traditional control flow graphs, EAFGs may contain paths that are infeasible. For instance, some sequences of transitions in the state machine may not be feasible because of conflicting guard conditions. In addition, some paths in EAFGs may not be feasible because of incompatible sub-path in compound nodes, leading to infeasible du-paths. (Concrete examples are provided in [22].) Theoretically speaking, determining these infeasible paths in EAFGs cannot be fully automated.

If an EAFG is used to derive test cases based on data flow information, then the user currently has to remove infeasible paths. Future work will investigate ways to facilitate this task.

Since, as we will see in Section 3.6, we intend to use this technique to analyse the data flow coverage of an existing test suite (in our case a transition tree test suite), we can assume that the test suite already contains feasible state machine transition sequences. In other words, we rely on a (partially automated) mechanism that builds feasible transition sequences. Then, once the data flow coverage of an existing test suite is determined, uncovered du pairs may be an indication of infeasible paths, and this has to be investigated. In other words, instead of evaluating the feasibility of each du pair, the tester can focus on the uncovered ones.

3.5.3. UML 1.5 vs UML 2.0

The changes from UML 1.5 to UML 2.0 are mostly related to terminology (e.g. state machine vs statechart, constraint vs guard condition, triggers vs events) and the underlying metamodel (e.g. class `ProtocolStateMachine` specializes class `StateMachine`) plus new concepts to help reuse (e.g. submachines, state machine specialization) [15]. Similarly, the OCL language has been enhanced in UML 2.0 (from UML 1.5), mainly at the metamodel level. Other changes to the OCL language do not have any impact on our approach (e.g. the new collection type `OrderedSet` has no impact since we do not rely on collection types) or would only lead to minor modifications of our rules (specifically, to account for the `OclMessage` type and its operations).

As already mentioned, our research assumes that the state machines have already been flattened and OCL constraints have been transformed accordingly. In other words, our research assumes that the state machine does not contain any concurrency between states or composition of states (in the UML 1.5 notation), or any state machine specialization or sub-machine (in UML 2.0).

This is the reason why the changes in the UML standard (from version 1.5 that we use in this research to version 2.0) do not impact the applicability of our approach or the results observed on the case studies presented below.

The flattening process, which can be automated, is out of the scope of this paper but does not present specific technical difficulties and is assumed to be a required initial step. The resultant flattened state machine can be regarded as an intermediate form, being used in the context of our approach only, rather than being visualized by modelers or testers.

3.6. Data flow analysis of transition trees

Recall that our objective is not to apply data flow criteria alone to UML state machines. Rather, we aim to use data flow analysis to refine and improve the existing state-based criteria. Although the previous sections are not specific to any coverage criterion, for reasons explained in Section 2, we are focusing in this paper on the transition tree criterion.

It is rather straightforward to apply the data flow analysis approach to the transition tree criterion since each tree edge corresponds to a state transition and each tree node to a state. Thus, each transition tree is transformed into an EAFG by simply using the approach used for flattened state machines. But in that case the resulting EAFG is also a tree. As an example, we present in Figure 5(a) a simple, abstract transition tree (obtained from Figure 1(b) using a breadth first traversal) and its corresponding EAFG in Figure 5(b). Owing to space constraints, Figure 5(b) does not show the details of compound nodes.

After the transition tree EAFGs are generated, definitions and uses are determined (Section 3.3), and def clear paths and du pairs contained in the transition tree EAFGs are then derived (Section 3.4). Similar to an EAFG produced from a state machine, an EAFG produced from a transition tree may contain unfeasible paths. The analysis of def clear paths in a transition tree EAFG is therefore an approximation, unless the designer manually (or semi-automatically) removes the unfeasible paths (Section 3.5.2).

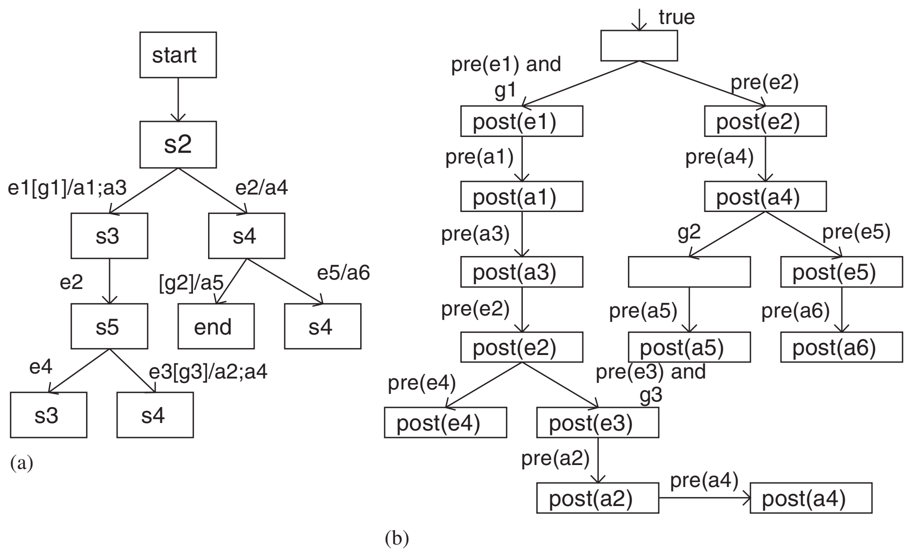


Figure 5. Transition Tree (a) and corresponding EAFG (b).

4. EMPIRICAL RESULTS

For our experiments we have selected two state machine subjects where multiple transition trees can be generated by following Binder's algorithm [4]. The first subject is a Cruise Control (CC) system [5] that has three transition trees and the second subject is a VCR system that is more complex than the CC system because of a larger number of states and transitions, guard conditions, and event parameters. As a result, the VCR state machine yields 12 transition trees with a Breadth First Search. To determine the fault detection effectiveness of a transition tree, we seeded faults into the code of the two subjects, using mutation operators proposed in [28–30]. When a large number of faults are needed to enable quantitative analysis, using mutation to seed faults is a common practice that has shown to yield realistic results in the past and is commonly used throughout testing research [31]. Our seeding strategy was 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 subject. To derive data flow information contained in a transition tree, we apply the approach proposed in Section 3.

Although our two subjects may appear to be of modest size, we consider that the CC, and the VCR subjects are representative examples of a large portion of state-based components that designers and testers would encounter in practice. Indeed, it is common to use state machines to design control classes in charge of the execution flow of use cases, such as the control of devices in reactive, real-time systems. Following standard UML-based development methodologies, state machines are often designed for classes and class clusters, sometimes for sub-systems, but rarely for complete systems. Indeed, in most cases, building a state machine for a complete system, if ever feasible, would result in a too large and unmanageable model for modelers and testers. This is also supported by industrial case studies reported in the literature. For instance, part of a flight guidance system developed at Rockwell–Collins is used as a case study in [14]. No state machine modeling the complete system is built and instead one of the most complex (flattened) state machines, built for a sub-system, consists in eight state and 44 transitions. Similarly, in [32], a safety controller subsystem for a system developed at ABB is modelled with a (flattened) state machine of 12 states and 51 transitions. These are to be compared with the nine states and 41 transitions of the flattened state machine of the VCR case study.

4.1. Subject descriptions

The VCR system is implemented in Java using the state design pattern. At the analysis level, one single class `VcrController` can be used to model the state-dependent behaviour of VCR, and the application of the state design pattern leads to a design of 29 classes. The class diagram and state machine, at the analysis level, as well as operation contracts are provided in [22]. The original hierarchical state machine has nine simple states, three composite states, 31 transitions, 10 distinct events, and seven guard conditions. (The flattened state machine has nine states and 41 transitions.) The code of the implementation contains a total of 1000 LOC. By following Binder's Algorithm, 12 transition trees are generated with Breadth First Search.

Following a similar implementation strategy, the CC is made of six classes for a total of 460 LOC. Its state-dependent behaviour is represented by a state machine of four states and 15 transitions. The class diagram and state machine, at the analysis level, as well as operation contracts are also provided in [22]. Three transition trees are generated.

4.1.1. Mutation scores

After analysing the VCR code, eight applicable mutation operators[‖‖] were used, yielding a total of 131 mutants to be seeded. Mutation scores for the 12 transition trees are listed in Table III. 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, hence, four groups are formed: Group 1 (TT1, TT7), Group 2 (TT2, TT5, TT8, TT11), Group 3 (TT3, TT9), and 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 the data flow information is related to the mutation scores.

For CC, six mutation operators were used to seed faults based on an analysis of the code (AOR, CRP, MNR, ROR, RSR, SDL), leading to the seeding of 91 faults. Important variations were observed in terms of fault detection ratio among the three possible transition trees: 91, 96, and 85%, for TT1, TT2, and TT3, respectively.

[‖‖]Arithmetic Operator Replacement (AOR), Constant Replacement (CRP), Method Name Replacement (MNR), Relational Operator Replacement (ROR), Return Statement Replacement (RSR), Statement Deletion (SDL), Instance Creation Expression Changes (ICE), and Overriding Method Removal (OMR) [28–30].

Table III. Mutation scores for transition trees (VCR).

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

Table IV. Def clear paths/du pairs in transition tree EAFGs (VCR).

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

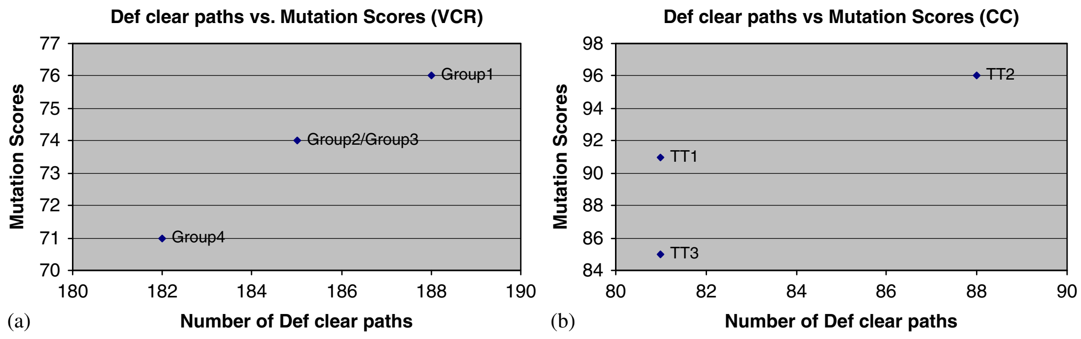


Figure 6. Def clear paths/du pairs vs mutation scores (VCR).

4.1.2. *Comparison of data flow information in each transition tree*

Applying the approach in Section 3 to VCR, we compute the def clear paths and du pairs in each of the EAFG transition trees. For each EAFG transition tree, the number of def clear paths happens to equal 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. Hence, each du pair is traversed by one def clear path. Table IV reports the def clear paths/du pairs data for each EAFG (numbered after their Transition Tree). We can see from Table IV that some EAFGs have the same number of def clear paths/du pairs. In fact, transition trees with the same mutation scores (i.e. within the same group) have the same number of def clear paths/du pairs, as illustrated in Table IV. The relationship between mutation scores and def clear paths for each group is depicted in Figure 6(a), showing a linear relationship between the number of def clear paths/du pairs and mutation scores.

Similar results are presented for CC in Table V and Figure 6(b). Similar to VCR, the number of def clear paths is the same as the number of du pairs and we can see from Figure 6(b) 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.

Table V. Def clear paths/du pairs in transition tree EAFGs (CC).

EAFG	EAFG for TT1	EAFG for TT2	EAFG for TT3
Def clear paths/DU pairs	81	88	81

Table VI. Number of definitions for EAFGs (VCR).

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

For TT1 and TT3, however, though they contain the same number of def clear paths/du pairs, the mutation score for TT1 is 7% higher than that for TT3.

Analysing the correlation between def clear paths/du pairs and mutation scores is a necessary first step. But to ensure that this is not a spurious result, we need to understand the mechanisms explaining the observed relationships. For VCR, 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.

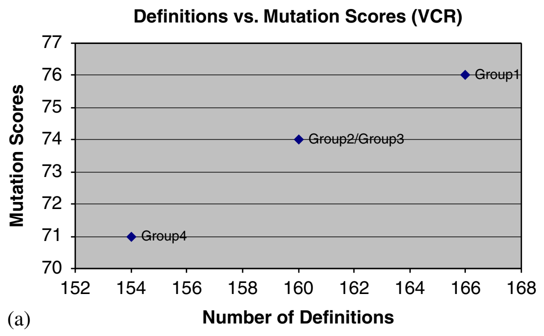
Furthermore, the analysis of live mutants suggests that traversing certain du pairs ensures that certain mutants be detected. These observations are also confirmed when analysing the mutants in CC. Mutation scores are explained not only by the number of du pairs covered but specific sets of du pairs tend to kill more mutants.

We also analyse the number of definitions in each EAFG, to see if it appears to be a good indicator of fault detection effectiveness as well (Table VI). For VCR, as in the case of def clear paths/du pairs, EAFGs within the same group have the same number of definitions, and there is again a clear linear relationship between the number of definitions and mutation scores (Figure 7(a)). Similar results can be observed for CC in Figure 7(b) for the three transition trees.

For VCR, we further investigated the specific sets of definitions in each EAFG and we determined that EAFGs within the same group have an identical set of definitions. 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.

4.2. Summary of results

The two case studies, based on state machines of very different nature and complexity, suggest that by following Binder's adaptation of Chow's algorithm, multiple transition trees with different mutation scores may be generated from one single state machine. In the VCR case study, which



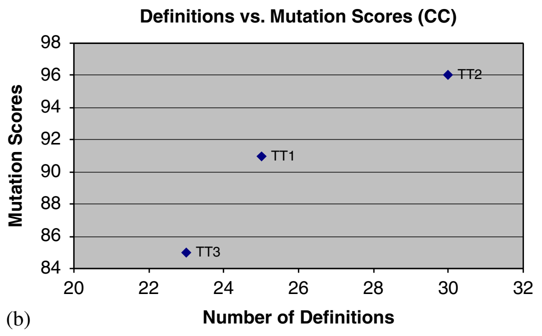


Figure 7. Definitions vs mutation scores.

is the most complex one, transition trees can be divided into groups based on mutation scores and their 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. A careful analysis of live mutants in each group shows that certain du pairs need to be covered to kill them, thus, explaining the relationship between du pairs covered by a transition tree and mutation scores.

Results therefore suggest that 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 do not contain any definitions (Section 3.3). Consequently, less complex and fewer rules, requiring simple OCL expression analysis, will be needed. 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 case studies, we have looked at the live mutants with each transition tree. In many cases, the live mutants with one transition tree are a complement set of the live mutants with another transition tree. For instance, in CC, 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. In other words, the effectiveness of a transition tree could be improved by adding paths from other trees that cover du pairs that are not already covered. 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. Following this heuristic, we next present a way to build augmented transition trees to improve fault detection effectiveness.

4.3. Augmenting transition trees

Building an augmented tree takes a sequence of steps. First, transition trees with an identical set of definitions are grouped together (Step 1). Then, we select from each group a transition tree with

Table VII. Comparing different criteria for the Cruise Control.

	Transition	Transition tree	Augmented transition tree	Transition-pair
Cost	25 (on average)	38	43	100
Mutation score (%)	96 (on average*)	85, 91, and 96 (for the three trees)	100	100

*The distribution of mutation scores of transition adequate test suites is wide, making it unlikely to detect all faults, although the average may seem high.

minimal cumulative length to form a tree set, say `tSet` (Step 2). Cumulative length, i.e. the total number of arcs in the tree, is an indication of cost (test set up, test execution, etc.) of a test suite. 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 `tSet` that cover all the definitions not already covered by the initial tree and has minimal cumulative length.

For the CC, the optimal transition tree built according to this strategy kills all the mutants, at the expense of a 13% increase in cumulative length of the test suite, our surrogate measure for cost. Note that following the same strategy but selecting du pairs instead of definitions leads to the same effectiveness but leads to an 89% increase in cumulative length. Comparing these results with the ones reported in [5] (see Table VII), for which the same mutants and cost measure have been used for the CC to study the cost effectiveness of the transition, transition-pair, and transition tree criteria, we can see that the augmented transition tree becomes a more interesting alternative (than the initial transition tree criterion) to the ineffective transition coverage criterion and the very expensive transition-pair coverage criterion. Similarly, for the VCR case study, the optimal transition tree built according to this strategy kills all the mutants, at the expense of a 29% increase in cumulative length. Using du pairs instead of definitions leads to the same effectiveness but a 106% increase in cumulative length. (VCR has not been used in [5].)

From the above results, we can conclude that with moderate increases in cost, all mutants are killed by test suites based on augmented transition trees. Such a strategy therefore seems promising from a practical standpoint.

5. CONCLUSIONS

The objective of this research is to investigate how data flow information can be used to improve state machine-based testing criteria. This is very important as recent research has shown that the existing coverage criteria are either too expensive, too weak, or too unpredictable in terms of fault detection effectiveness when applied to software components. To this end, we provide a methodology to conduct data flow analysis of UML state machines, apply it to a specific, well-known testing

***The exhaustive search is $O(2^p)$, where p is the number of paths in trees. This is usually all right for most state machines in practice. But for exceptionally large sets of trees, other search strategies, such as genetic algorithms, could be investigated in future work.

criterion (round-trip paths, an adaption for UML state machines of Chow's W-method based on transition trees), and use it on two case studies that are representative models of state-based software components being produced in OO systems. Our data flow analysis is comprehensive as it accounts for all types of events and actions in UML state machines and relies on an extensive analysis of OCL guard conditions and OCL contracts (operation pre and postconditions). The precise rules for detecting definition–use (du) pairs are defined by deriving a specific representation of control and data flow in state machines. We define the structure and well formedness of a state machine control and data flow using a metamodel (class diagram notation), and define the conditions under which definitions and uses are expected using the Object Constraint Language (OCL) on the metamodel. It is important to note that we do not use data flow criteria to create test cases. Rather we analyse the data flow coverage of alternative, adequate test suites for the existing state machine criteria, in order to select the suite with the highest fault-revealing power. Additionally, our model-based data flow analysis is automatable, thereby making the whole approach inexpensive.

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 state machine, the transition tree that is the most effective at detecting faults tends to cover the largest number of du pairs and definitions. We examine undetected faults in both of the case studies. The results show that certain du pairs and definitions guarantee that some faults 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 that contains the largest number of du pairs or definitions would be most effective at detecting faults. Since both du pairs and definitions can be used to select trees, using the latter, however, may be a better choice in practice since identifying definitions is easier and less costly than identifying du pairs. The research also suggests that combining different parts of transition trees so as to optimize data flow coverage could yield highly cost-effective results but this is the topic of current research. Following this heuristic we propose a new way to build augmented transition trees, based on data flow analysis, that improve fault detection effectiveness while only incurring a modest additional cost.

There are several limitations of this research. Currently, we rely on the user input to determine the incompatible sequences of operations and infeasible paths within a compound node in EAFGs. Although this cannot be fully automated from a theoretical point of view, future work will explore practical heuristics that can provide the approximate solutions. Furthermore, since we apply our data flow analysis to an existing test suite, we can reasonably assume that the test suite contains feasible transition sequences. And, since we analyse the data flow coverage of the test suite, user input is only required when some du pairs (or definitions) are not covered, instead of requiring the analysis of every single du pair (definition).

As for any empirical work, 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 that 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. In addition, since our results are based on transition trees generated using Breadth First Search, transition trees that are obtained by Depth First Search should be investigated to confirm our results. Lastly, our approach uses a flattened state machine as input and work remains to be done to automatically flatten UML 2.0 state machine and transform the OCL guard conditions and OCL operation contracts accordingly.

Other future work will study the impact of the level of precision in OCL contracts, as well as the writing practices of OCL expressions (e.g. we assume that the predicate part of an if–then–else expression does not contain definitions of model elements). A study of whether a more advanced analysis technique for OCL expressions is warranted should also be conducted.

REFERENCES

1. OMG. UML 2.0 Superstructure Specification. *Final Adopted Specification ptc/03-08-02*, Object Management Group, 2003.
2. OMG, Unified Modeling Language (UML) Specification Version 1.5. *Object Management Group formal/03-03-01*, March, 2003.
3. Offutt AJ, Liu S, Abdurazik A, Ammann P. Generating test data from state-based specifications. *Software Testing, Verification and Reliability* 2003; **13**(1):25–53.
4. Binder RV. Testing object-oriented systems—Models, patterns, and tools. *Object Technology*. Addison-Wesley: Reading, MA, 1999.
5. Briand LC, Labiche Y, Wang Y. Using simulation to empirically investigate test coverage criteria. *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2004; 86–95.
6. Frankl PG, Weyuker E. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering* 1988; **14**(10):1483–1498.
7. Hong HS, Kim YG, Cha SD, Bae DH, Ural H. A test sequence selection method for statecharts. *Software Testing, Verification and Reliability* 2000; **10**(4):203–227.
8. OMG. OCL 2.0 Specification. *Final Adopted Specification ptc/03-10-14*, Object Management Group, 2003.
9. Kleppe A, Warmer J, Bast W. *MDA Explained—The Model Driven Architecture: Practice and Promise*. Addison-Wesley: Reading, MA, 2003.
10. Bruegge B, Dutoit AH. *Object-oriented Software Engineering Using UML, Patterns, and Java* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 2004.
11. Appel AW. *Modern Compiler Implementation in Java* (2nd edn). Cambridge University Press: Cambridge, 2002.
12. Chow TS. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 1978; **SE-4**(3):178–187.
13. Zweben SH, Heym WD. Systematic testing of data abstractions based on software specifications. *Journal of Software Testing, Verification and Reliability* 1992; **1**(4):39–55.
14. Chevalley P, Thévenod-Fosse P. Automated generation of statistical test cases from UML state diagrams. *Proceedings of the International Computer Software and Applications Conference*, Chicago, IL, U.S.A., 2001; 205–214.
15. Pender T. *UML Bible*. Wiley: New York, 2003.
16. Briand LC, Di Penta M, Labiche Y. Assessing and improving state-based class testing: a series of experiments. *IEEE Transactions of Software Engineering* 2004; **30**(11):770–793.
17. Ostrand TJ, Balcer MJ. The category-partition method for specifying and generating functional test. *Communications of the ACM* 1988; **31**(6):676–686.
18. Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* 1985; **SE-11**(4):367–375.
19. Edwards SH. Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability* 2000; **10**(4):249–262.
20. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*. Addison-Wesley: Reading, 1999.
21. Beizer B. *Software Testing Techniques* (2nd edn). Van Nostrand Reinhold: New York, 1990.
22. Briand LC, Labiche Y, Lin Q. Improving state-based coverage criteria using data flow information. *Technical Report SCE-04-17*, Carleton University, 2004. Avialable at: www.sce.carleton.ca/Squall [January 2009].
23. Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language Reference Manual*. Addison-Wesley: Reading, MA, 1999.
24. Mitchell R, McKim J. *Design by Contract, by Example*. Addison-Wesley: Reading, MA, 2001.
25. Warmer J, Kleppe A. *The Object Constraint Language* (2nd edn). Addison-Wesley: Reading, MA, 2003.
26. Briand LC, Labiche Y, Cui J. Automated support for deriving test requirements from UML statecharts. *Journal of Software and Systems Modeling* 2005; **4**(4):399–423.
27. Briand LC, Labiche Y, Sun H. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Software—Practice and Experience* 2003; **33**(7):637–672.
28. Kim S, Clark JA, McDermid JA. The rigorous generation of Java mutation using HAZOP. *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications* (*ICSSEA'99*), vol. 3, Paris, France, 1999; 9–10(11).

29. Kim S, Clark JA, McDermid JA. Class mutation: Mutation testing for object-oriented programs. *Proceedings of the Net.ObjectDays*, Erfurt, Germany, 2000.
30. King KN, Offutt AJ. A Fortran language system for mutation-based software testing. *Software—Practice and Experience* 1991; **21**(7):686–718.
31. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *Proceedings of the IEEE International Conference on Software Engineering*, St. Louis, Missouri, U.S.A., 2005; 402–411.
32. Holt NE, Anda BCD, Asskildt K, Briand LC, Endresen J, Frøystein S. Experiences with precise state modeling in an industrial safety critical system. *Proceedings of the Critical Systems Development Using Modeling Languages*, *Workshop in Conjunction with UML'06*, Genova, Italy, 2006; 68–77.