

# Slicing of UML Models

K. Lano, S. Kolahdouz-Rahimi

Dept. of Computer Science, King's College London

**Abstract.** This paper defines techniques for the *slicing* of UML models, that is, for the restriction of models to those parts which specify the properties of a subset of the elements within them. The purpose of this restriction is to produce a smaller model which permits more effective analysis and comprehension than the complete model, and also to form a step in factoring of a model. We consider class diagrams, individual state machines, and communicating sets of state machines.

## 1 Introduction

Slicing of programs was introduced by [12] as a technique for the analysis of software, and has also been used for reverse-engineering and re-factoring of software. With the advent of UML and model-based development approaches such as Model-Driven Development (MDD) and Model-Driven Architecture (MDA), models such as UML class diagrams and state machines have become important artifacts within the development process, so that slicing-based analysis of these models has potential significance as a means of detecting flaws and in restructuring these models.

Slicing techniques for specification languages such as Z [13] and Object-Z [1] have been defined, based on variants of the concepts of control and data dependence used to compute slices for programs. However, UML contains both declarative elements, such as pre- and post-conditions, and imperative elements, such as behaviour state machines, so that slicing techniques for UML must treat both aspects in an integrated manner.

We will use a generalised concept of slicing: a slice  $S$  of an artifact  $M$  is a transformed version of  $M$  which has a lower value of some complexity measure, but an equivalent semantics with respect to the sliced data:

$$S <_{syn} M \wedge S =_{sem} M$$

The relations  $<_{syn}$  and  $=_{sem}$  used depend on the type of analysis we wish to perform on  $M$ :  $S$  should have identical semantics to  $M$  for the properties of interest, but may differ for other properties. The slicing may be structure-preserving, so that  $S$  has the same structure as  $M$  although containing only a subset of its elements, or may be *amorphous*, with a possibly completely different structure [2].

We will consider techniques for the structure-preserving and amorphous slicing of class diagrams and state machines. Model transformations will be used

to perform slicing, each transformation will satisfy the  $<_{syn}$  and  $=_{sem}$  relations, resulting in a slice which also satisfies these relations compared to the original model. In general, applying slicing at a high level of abstraction simplifies the calculation of the slice, and means it is possible to detect specification flaws at an early development stage, thus reducing development costs.

## 2 Slicing of Class Diagrams

We consider first the most abstract form of UML specification using class diagrams to define the structure of classes and associations, and constraints in OCL to define behaviour by means of invariant constraints of classes and pre- and post-condition constraints of operations. The range of UML constructs considered here corresponds to the UML-RSDS subset of UML [5]. We assume that the client-supplier relation between different classes forms a tree structure. The class at the root of this tree is termed the *controller* class of the system: it will usually serve as the access point to the services of the system for external users. Operations are assumed to be deterministic.

Figure 1 shows an example of this style of specification, for a lift control system.

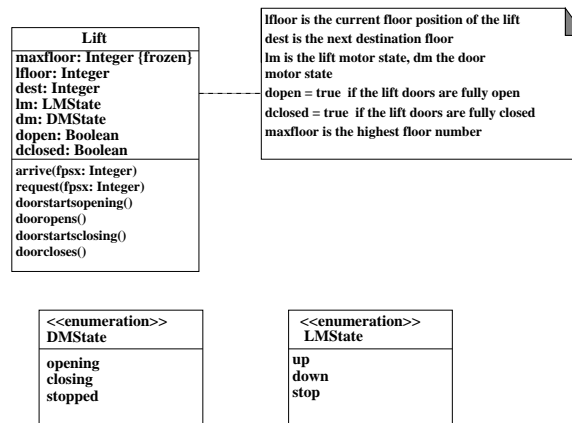


Fig. 1. Lift control system

The *data features* of such a system are all the attributes of classes in the system, including association ends owned by the classes. For each class  $C$  there is also the system data feature  $C.allInstances()$  of all currently existing instances of  $C$ .

Slicing will be carried out upon class invariants and operation pre and post-conditions by considering the *predicates*  $P$  of which they are composed. A predicate is a truth-valued formula, whose main operator is not *and*. Class invariant

predicates can be either *assertions*: properties which are expected to be invariant for objects of the class, but which do not contribute to the effect of operation postconditions, or *effective*: implicitly conjoined to the postcondition of each update operation.

In order to reduce the dependencies in a model, we assume that certain effective class invariant predicates, and operation postcondition predicates are *operational*<sup>1</sup>, if they have the form

$$L \text{ implies } R$$

where  $R$  is a formula such as  $f = e$ ,  $f \rightarrow \text{includes}(e)$ ,  $f \rightarrow \text{excludes}(e)$ ,  $f \rightarrow \text{includesAll}(e)$ ,  $f \rightarrow \text{excludesAll}(e)$ ,  $\text{ref}.f = e$ ,  $\text{ref}.f \rightarrow \text{includes}(e)$ ,  $\text{ref}.f \rightarrow \text{excludes}(e)$ ,  $\text{ref}.f \rightarrow \text{includesAll}(e)$ ,  $\text{ref}.f \rightarrow \text{excludesAll}(e)$ .  $f$  is a feature name, possibly with a selector  $\rightarrow \text{at}(g)$  in the case of equality, for some expression  $g$ , in the case of an ordered role  $f$ .  $f$  is not a pre-form of a feature (ie, not  $x@pre$  for some feature  $x$ ).  $f$  is called the *writable feature* of the constraint.  $L$  may be omitted.  $L$  is the *test* part of the predicate,  $e$  the *value* part,  $\text{ref}$  the *reference* part, and  $g$  the *index* part.

Other predicates are termed *non-operational*, they may (in the case of postcondition predicates) consist of a mixture of features in pre form and post form, all the features in post form are assumed to be writable in this case, and to be mutually data-dependent.

In the lift example, there are class invariant constraints

```
dclosed = false implies lm = stop
dopen = true implies dclosed = false
0 ≤ dest and dest ≤ maxfloor
0 ≤ lfloor and lfloor ≤ maxfloor
```

of *Lift*.

Apart from the first predicate, these are all assertions. The operations of the lift can be specified as follows:

```
init() /* Invoked at object creation */
post:
  lfloor = 0 and dest = 0 and dclosed = false and
  dopen = true and lm = stop and dm = stopped

arrive(fpsx : Integer) /* Lift arrives at floor fpsx */
pre: 0 ≤ fpsx and fpsx ≤ maxfloor
post:
  lfloor = fpsx and
  fpsx = dest implies lm = stop and
  fpsx = dest implies dm = opening and
  dclosed = false implies lm = stop

request(destx : Integer) /* Request to go to floor destx */
```

<sup>1</sup> In the UML2Web toolset for UML-RSDS, developers can designate effective constraints as operational when they are created.

```

pre: 0 ≤ destx and destx ≤ maxfloor
post:
  dest = destx and
  destx > lfloor and dclosed = true implies lm = up and
  destx < lfloor and dclosed = true implies lm = down and
  destx = lfloor implies lm = stop and
  destx = lfloor and dopen = true implies dm = stopped and
  destx = lfloor and dopen = false implies dm = opening and
  destx ≠ lfloor and dclosed = false implies dm = closing and
  destx ≠ lfloor and dclosed = true implies dm = stopped and
  dclosed = false implies lm = stop

doorstartsopening()
pre: dclosed = true
post: dclosed = false and lm = stop

doorstartsclosing()
pre: dopen = true
post: dopen = false

doorcloses()
pre: dclosed = false and dopen = false
post:
  dclosed = true and dm = stopped and
  dest > lfloor implies lm = up and
  dest < lfloor implies lm = down

dooropens()
pre: dopen = false and dclosed = false
post: dopen = true and dm = stopped

```

For any system there are also implicit operational constraints which relate the two ends of binary associations (Figure 2). In this case a change to one end of the association implies a change to the other, because of the implicit constraints relating these ends (Table 1). The features which consist of these opposite association ends are data-dependent upon each other because they are derived features in terms of each other.

Assertion constraints arise from the multiplicity restrictions of association ends, ie, a 0..n multiplicity end *br* has a constraint  $br \rightarrow size() \leq n$ .

If class *C* is a subclass of class *D* then there is an effective non-operational constraint  $D.allInstances() \rightarrow includesAll(C.allInstances())$  and likewise for association ends *r* which specialise other ends *s*:  $s \rightarrow includesAll(r)$  and union relationships between association ends:  $ru = r_1 \cup \dots \cup r_n$ .

Further implicit constraints for a class are the invariant constraints inherited from any superclass. Preconditions and postconditions from superclass versions of an operation are not inherited, if an explicit definition of the operation is defined in the subclass: this explicit definition replaces superclass definitions for objects of the subclass.



**Fig. 2.** Binary association

M1	M2	Constraints
*	*	$ar = A.allInstances() \rightarrow select(ax)$   $ax.br \rightarrow includes(self)$ $br = B.allInstances() \rightarrow select(bx)$   $bx.ar \rightarrow includes(self)$
1	*	$ar = A.allInstances() \rightarrow select(ax)$   $ax.br \rightarrow includes(self) \rightarrow any()$ $br = B.allInstances() \rightarrow select(bx)$   $self = bx.ar$
*	1	$ar = A.allInstances() \rightarrow select(ax)$   $self = ax.br$ $br = B.allInstances() \rightarrow select(bx)$   $bx.ar \rightarrow includes(self) \rightarrow any()$
1	1	$ar = A.allInstances() \rightarrow select(ax)$   $self = ax.br \rightarrow any()$ $br = B.allInstances() \rightarrow select(bx)$   $self = bx.ar \rightarrow any()$

**Table 1.** Implicit derivation constraints linking association ends

We will slice a class diagram specification  $M$  by slicing the controller class  $C$  of  $M$ : this class is directly or indirectly a client of all other classes in  $M$  and is not a supplier of any other class in  $M$ .

A *history* of a specification  $M$  is a finite sequence of invocations of update operations on instances of  $C$ . A history is valid if operations are only invoked on instances which have been created, and for which the operation precondition is true at the point of call, in addition the history should conform to the protocol state machine  $SM_C$  of  $C$ . For example,

$create_{Lift}(cx), cx.request(10), cx.doorstartsclosing(), cx.doorcloses(), cx.arrive(1)$

is a valid history for the lift control system example, if  $maxfloor = 15$ .

The definition of slicing we will use for class diagrams  $M$  is the following:  $S <_{syn} M$  if the slice  $S$  has a subset of the elements of  $M$ .  $S =_{sem} M$  holds, with respect to a given state  $s$  in the protocol state machine of  $C$ , and set  $V$  of data features of  $M$ , if any valid history  $\sigma$  with final state  $s$  of  $M$  is also a valid history with final state  $s$  in  $S$ , and if the values of the features in the slice set  $V$  in  $S$  in the final state of  $\sigma$  are equal to the values of these features in  $M$  at the final state when  $\sigma$  is applied to both models, starting from the same initial values for the features they have in common. We assume that the protocol state machines of  $C$  in  $M$  and  $S$  have the same states (more generally, the states in  $M$  could be a refinement of those in  $S$ ).

The set of features and constraints in a slice  $S$  can be determined by examining the data dependencies of the constraints of the original model.

The first step in producing a slice of a class  $C$  is to normalise the class invariant, and each pre and post condition, so that these are all in the form of conjunctions of predicates. Formulae  $P \text{ implies } R \text{ and } Q$  are rewritten as  $P \text{ implies } R$  and  $P \text{ implies } Q$ , etc. Then the effective invariant predicates are copied to the postconditions of each update operation. Assertion predicates are not copied.

For each postcondition predicate  $p$  we define the sets of features read and written in  $p$ , and its internal data dependencies:

- $wr(p)$  is the set of features written to in  $p$ . If  $p$  is operational, this set is the single writable feature of  $p$ , otherwise it is the set of features not in  $pre$  form in  $p$ .
- $rd(p)$  is the set of features read in  $p$ . If  $p$  is operational this is the set of all features occurring in the test, value, reference or index expressions in  $p$ , and the pre form  $f@pre$  of the writable feature  $f$  (in the case of a simple equality  $f = e$ , the pre form  $f@pre$  of the writable feature  $f$  itself is not included, unless it occurs in  $e$ ). For other predicates  $rd(p)$  is the set of features in  $pre$  form, or input parameters.
- In the case of a formula  $C.allInstances() \rightarrow exists(P)$  specifying creation of an instance of  $C$  satisfying  $P$ ,  $C.allInstances()$  is a written feature and  $C.allInstances()@pre$  a read feature.
- In the case of formulae  $x.occlIsNew()$  or  $x.isDeleted()$ , where  $x$  is of class type  $C$ ,  $C.allInstances()$  is a written feature and  $C.allInstances()@pre$  and  $x$  are read features.
- The internal data-dependencies of an operational predicate  $p$  are then:

$$dep(p) = rd(p) \times wr(p)$$

and for a non-operational predicate

$$dep(p) = (rd(p) \cup wr(p)) \times wr(p)$$

There is data-dependency between two different predicates  $p$  and  $q$  in the same postcondition if  $p$  writes a feature which  $q$  reads:

$$p \mapsto q \equiv wr(p) \cap rd(q) \neq \{\}$$

The *write frame* of an operation  $op$  is the union of  $wr(p)$  for the predicates  $p$  in its postcondition, this is denoted  $wr(op)$ .

The predicates in the postcondition of an operation are assumed to be control dependent on the predicates in the precondition [1].

At the level of particular features,  $f$ ,  $g$ , there is a direct dependency of  $f$  on  $g$  in an operation  $op$ , if:

- $g \mapsto f$  is in some  $dep(p)$  for a postcondition predicate  $p$  of  $op$ .

Let  $r_{op}$  be the (non-reflexive) transitive closure of this relation. Then the feature dependency relation  $\rho_{op}$  of  $op$  includes the pairs:

- $g \mapsto f$  if  $g$  occurs in the precondition and  $f$  is in  $wr(p)$  for some postcondition predicate  $p$
- $g \mapsto f$  if  $g$  is an input parameter of the operation, or is a feature not in  $wr(op)$ , and  $g \mapsto f$  is in  $r_{op}$
- $g \mapsto f$  if  $g@pre \mapsto f$  is in  $r_{op}$
- $x \mapsto x$  if  $x \notin wr(op)$ .

The meaning of this relation is that the value of  $g$  at the start of the operation may affect the value of  $f$  at the end. Initial values of features not in  $\rho_{op}^{-1}(V)$  cannot affect the value of any feature in  $V$  at termination of the operation.

The feature dependencies of *arrive* in the control system are therefore:

$$\{maxfloor \mapsto lfloor, fpsx \mapsto lfloor, fpsx \mapsto lm, \\ dest \mapsto lm, fpsx \mapsto dm, dest \mapsto dm, dclosed \mapsto lm, \\ lm \mapsto lm, dm \mapsto dm, maxfloor \mapsto lm, maxfloor \mapsto dm, \\ dest \mapsto dest, dclosed \mapsto dclosed, dopen \mapsto dopen, maxfloor \mapsto maxfloor\}$$

For *doorcloses* they are:

$$\{dclosed \mapsto dclosed, dopen \mapsto dclosed, dclosed \mapsto dm, dopen \mapsto dm, \\ dclosed \mapsto lm, dopen \mapsto lm, lm \mapsto lm, dest \mapsto lm, lfloor \mapsto lm, \\ dopen \mapsto dopen, dest \mapsto dest, lfloor \mapsto lfloor, maxfloor \mapsto maxfloor\}$$

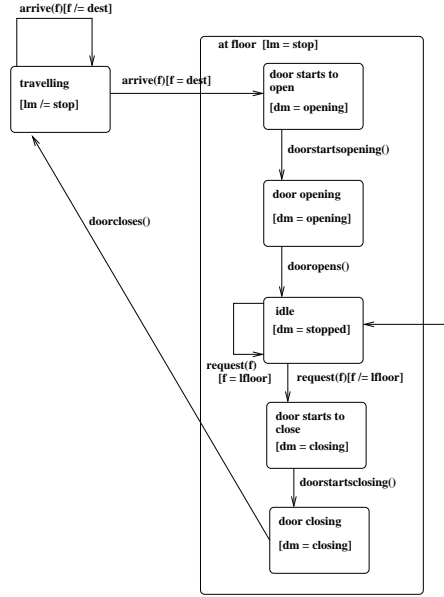
In order to analyse dependencies between data in different operations, we need to consider the possible life histories of objects of the class. A UML class  $C$  may have a protocol state machine  $SM_C$  as its *classifierBehavior* [6], this state machine defines what sequences of operations can be applied to the object, and under what conditions. This allows us to restrict dependencies by considering the possible orders in which data can be defined in one operation and used by another.

Figure 3 shows the protocol state machine for the *Lift* class. State invariants such as  $dm = opening$  are considered as assertions and do not contribute to data or control dependencies. If an explicit protocol state machine is not provided for a class, then the default behaviour of the class is a state machine with a single state, and self transitions for each operation on this state, guarded by the operation preconditions.

The state machine  $SM_C$  can be used as the basis of the data and control flow graph  $G_C$  of the class  $C$ . We assume that only basic and OR-composite states are present in  $SM_C$ , and there are no history states. We carry out normalisation of the pre and post-conditions of each transition of  $t$ , so that these are conjunctions of predicates.

The primary nodes of  $G_C$  are:

- The basic states of  $SM_C$
- A precondition/guard node  $pre_t$  for each transition  $t$  of  $SM_C$
- A postcondition node  $post_t$  for each transition  $t$  of  $SM_C$ .



**Fig. 3.** Lift protocol state machine

Within each node  $pre_t$  there are subordinate nodes for each predicate of the guard of  $t$ , and each predicate of the precondition of the operation  $op$  that triggers  $t$ . Within each node  $post_t$  there are subordinate nodes for each predicate of the postcondition of  $t$ , and each predicate of the postcondition of the operation  $op$  that triggers  $t$ .

There is direct data dependency from a write occurrence  $d$  of a feature  $f$  within predicate  $p$  of a postcondition node  $n$ , to a read occurrence  $d'$  of  $f$  within predicate  $q$  of a node  $n'$  if either:

1.  $n = n'$ ,  $p \neq q$  and both  $d$  and  $d'$  are  $f$
2.  $n \neq n'$ ,  $d$  is  $f$ ,  $d'$  is  $f@pre$  in a postcondition node, or  $f$  in a precondition node, and  $n'$  is reachable from  $n$  along a path  $\sigma$  following the control flow of the state machine, and there is no intermediate node  $m$  strictly between  $n$  and  $n'$  in  $\sigma$  which contains a write occurrence of  $f$  in its predicates.

Figure 4 shows part of the data and control flow graph of the lift system, with direct data dependencies for  $d_{closed}$  between different top-level nodes marked by dashed lines. The dependency relation  $\rho_C$  is then the transitive closure of the union of the data-dependency and control-flow relations on predicates.

The following algorithm is used to carry out the slicing transformation, for a state  $s$  and set  $V$  of features, using the graph  $G_C$ . We associate a set  $V_x$  of features to each basic state node of the data and control flow graph.

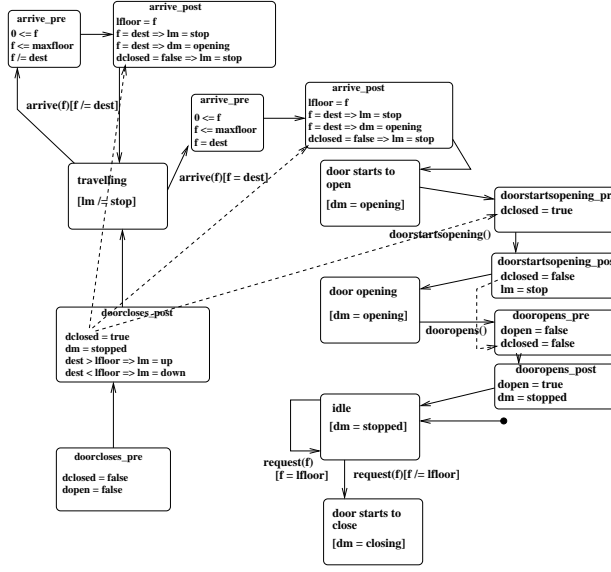


Fig. 4. Lift data and control flow graph

1. Initialise each  $V_x$  with the empty set of features, except for the target state  $s$ , which has the set  $V$  of features.
2. For each transition  $t : s_1 \rightarrow s_2$ , add to  $V_{s_1}$  the set  $\rho_{op}^{-1}(\bigcap V_{s_2})$  of features upon which  $V_{s_2}$  depends, via the version of the operation  $op$  executed by this transition (with precondition and postcondition formed from both the class and state machine predicates for  $op$  and  $t$ ).

Mark as included in the slice those predicates of  $pre_t$ ,  $post_t$  which are in  $\rho_C^{-1}(\bigcap ps)$  where  $ps$  is the set of predicates contained in  $post_t$  which have write occurrences of features in  $V_{s_2}$ . The set of features (with  $pre$  removed) used in the marked predicates are also added to  $V_{s_1}$ .

The second step is iterated until a fixed point is reached. Each  $V_x$  then represents the set of features whose value in state  $x$  can affect the value of  $V$  in state  $s$ , on one or more paths from  $x$  to  $s$ . (Parameter values of operations along the paths may also affect  $V$  in  $s$ ). Let  $V'$  be the union of the  $V_x$  sets, for all states  $x$  on paths from the initial state of  $SM_C$  to  $s$ . The set of features retained in the slice  $S$  will be set equal to  $V'$ .

If the lift system is sliced with  $s = \text{idle}$ ,  $V = \{dm\}$ , then  $V'$  is the set of all features of the lift, with  $lm$  removed. The new class invariant constraints are:

$$\begin{aligned}
 & dopen = true \text{ implies } dclosed = false \\
 & 0 \leq dest \text{ and } dest \leq \text{maxfloor} \\
 & 0 \leq lfloor \text{ and } lfloor \leq \text{maxfloor}
 \end{aligned}$$

The sliced *init* and *request* operations of the lift are:

```

init()
post:
  lfloor = 0 and dest = 0 and dclosed = false and
  dopen = true and dm = stopped

request(destx : Integer)
pre: 0 ≤ destx and destx ≤ maxfloor
post:
  dest = destx and
  destx = lfloor and dopen = true implies dm = stopped and
  destx = lfloor and dopen = false implies dm = opening and
  destx ≠ lfloor and dclosed = false implies dm = closing and
  destx ≠ lfloor and dclosed = true implies dm = stopped

```

If an effective invariant predicate does not occur in any operation postcondition in the slice, then it can be removed from the class. If a data feature does not occur in any operation or effective invariant within the specification, then it can be removed, together with any assertions that refer to it. In the above example,  $lm$  can be removed from the *Lift* class, together with the invariant involving  $lm$ .

In general, the sliced specification will be smaller, more easy to analyse and verify, and will define the same properties as the original system for the features  $V$ . Therefore, slicing may substantially reduce the size of the model to be checked, and make verification, eg, by translation to a proof tool such as B [11], feasible when verification of the original model was not feasible. This form of slicing is structure-preserving.

The transformation we have described above does produce a semantically correct slice  $S$  of a model  $M$ , using the definition  $=_{sem}$  of semantic equivalence, because, if  $\sigma$  is a valid history of  $M$  (ie, of the controller class  $C$  of  $M$ ), ending in the slice target state  $s$ , and  $V$  a set of features of  $M$ , then:

- $\sigma$  is also a valid history of  $S$ , since the set of states in the controller class state machine are the same in both models, as are the preconditions and guards of each transition in the models
- the features retained in  $S$  are the union  $V'$  of the sets  $V_x$  of the features upon which  $V$  in  $s$  depends, for each state  $x$  of any path to  $s$ , and hence  $V'$  contains  $V_x$  for each state on the history  $\sigma$ , and in particular for the initial state
- since the values of the features of  $V'$  in the initial state are the same for  $S$  and  $M$ , and the values of operation parameters are also the same in the application of  $\sigma$  to  $S$  and  $M$ , the values of  $V$  in  $s$  are also the same in both models.

### 3 Behaviour State Machine Slicing

Operations may have their effect defined by a behaviour state machine, instead of a postcondition. Such state machines do not have operation call triggers on their transitions, instead their transitions are triggered by completion of the source

state. The state machines will also have a final state, representing termination of the operation whose effect they define. Slicing can be carried out for such state machines, using data and control flow analysis to remove elements of the machine which do not contribute to the values of a set of features in the final state of the machine.

The criteria for a slice  $S$  of a state machine  $M$  are:  $S <_{syn} M$  if  $S$  has fewer elements (states, transitions, transition actions, etc) than  $M$ .  $S =_{sem} M$  if for all possible sequences  $e$  of input events of  $M$ , starting from  $S$  and  $M$  in their initial states, with the same values for their common features in the initial states, the state  $s$  of interest is reached by  $S$  as a result of the sequence  $e$  whenever it is reached by  $M$  as a result of the same sequence, and then the value of the features  $V$  of interest in the state  $s$  of interest are the same in the two models. We assume that  $s$  is in both state machines<sup>2</sup>.

This means that any analysis which concerns the value of the slice features  $V$  in the selected state  $s$ , over all paths to this state, can be performed on the slice  $S$ , and the result will also apply to  $M$ . In particular, if predicate  $P$  can be proved to be a state invariant of  $s$  in  $S$ , then it will also be a state invariant of  $s$  in  $M$ .

States which cannot occur in paths from the initial state to the state of interest can be deleted from the model, together with their incoming and outgoing transitions.

Care must be taken concerning the state machine notation considered, and the semantics adopted, since the computation of the slice will differ from version to version. Three alternative semantics can be used, in the case that there is not a complete set of guards covering all possibilities of an event occurrence in a given state [8]: skip/ignore semantics; precondition/exception semantics; blocking semantics. We will assume skip semantics for behaviour state machines. Figure 5 shows the metamodel of our state machine notation. Dashed lines indicate associations which are not part of the metamodel but are used to hold computed data during the slicing process. Statements will either be assignments or, for communicating state machines, invocations of operations on supplier objects.

Figure 6 shows an example state machine, of the *doorcloses* operation. The predicate *dclosed = true* remains in the operation postcondition, the other effects of the operation are defined by the state machine.

Given a particular state  $s$  in a state machine and a set  $V$  of features of interest in that state, we determine the slice of the state machine with respect to  $s$  and  $V$  by computing for each state  $x$  of the state machine, a set  $V_x$  of features such that: the value of the features of  $V_x$  in state  $x$  may affect the value of a feature in  $V$  in state  $s$ , but that no other feature in state  $x$  can affect  $V$  in state  $s$ . That is, for all possible paths from  $x$  to  $s$ , the value of  $V$  in  $s$  at the end of the path depends only upon the values of  $V_x$  in  $x$  at the start of the path.

The sets  $V_x$  are computed by an iteration over all the transitions of the state machine. They are initialised to  $\{\}$  for  $x \neq s$ , and to  $V$  for  $V_s$ . For each

---

<sup>2</sup> In the case of behaviour state machines for operations, the empty sequence is the only case that needs to be considered for  $e$ .

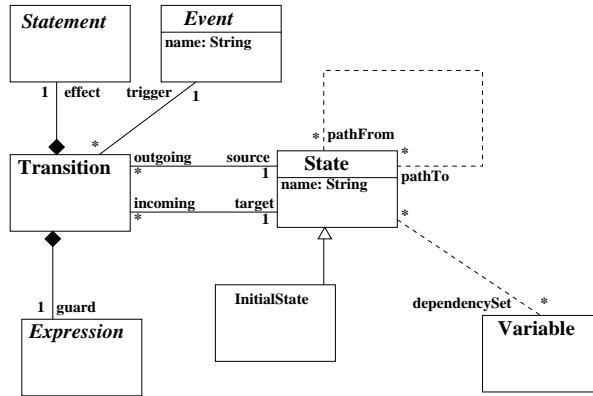


Fig. 5. Behaviour state machine metamodel

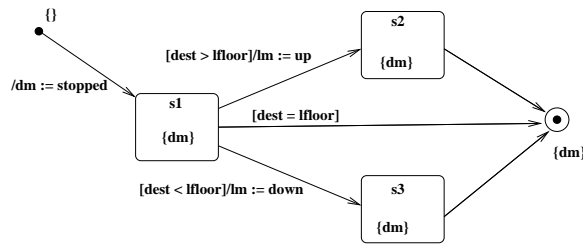


Fig. 6. doorcloser state machine

transition

$$tr : s1 \rightarrow_{op(p)[G]/acts} s2$$

the set  $V_{s1}$  of features of interest in  $s1$  are augmented by all features which appear in  $Pre_{op}$  and  $G$ , and by all features which may affect the value of  $V_{s2}$  in  $s2$  as a result of the class definition of  $op(p)$ , followed by  $acts$ <sup>3</sup>. This iteration is repeated until there is no change in any  $V_x$  set.

These dependencies can be simplified, and the  $V_x$  sets made smaller, by omitting features of the guards in cases where the values of features in  $V_s$  in  $s$  always depend on the  $V_x$  in the same way, regardless of the paths taken from  $x$  to  $s$  because of different guards. In Figure 6 the transition guards do not affect the value of  $dm$  in the final state, so their features do not need to be added to the dependency set  $\{dm\}$  of  $s1$ .

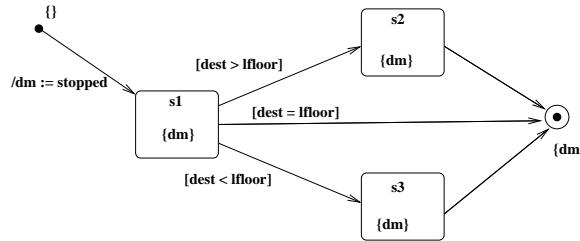
Using the sets  $V_x$ , further transformations can be applied to simplify the state machine: transition slicing, transition deletion, transition merging and state merging, and replacing variables by constants.

Individual transitions are sliced by removing actions which cannot contribute to the values of the features  $V$  in state  $s$ . For a transition

$$tr : s1 \rightarrow_{op(x)[G]/acts} s2$$

all updates in  $acts$  which do not affect  $V_{s2}$  in  $s2$  can be deleted from  $acts$  to produce a simpler transition.

Figure 7 shows the operation example after transition slicing, for  $V = \{dm\}$  and  $s$  being the final state.



**Fig. 7.** *doorcloses* state machine after transition slicing

Transitions can be deleted if their guard is equivalent to *false*. Two transitions can be merged if their sources, targets and actions are the same. The

<sup>3</sup> The usual definition of data-dependency due to assignment is used: in  $x := e$   $x$  depends on all features in  $e$ , and features apart from  $x$  depend on themselves. Dependencies for operation calls are calculated from the definition of the called operation.

guard of the resulting transition is the disjunction of the original guards.  $tr1 : s1 \rightarrow_{op(x)[G1]/acts} s2$  and  $tr2 : s1 \rightarrow_{op(x)[G2]/acts} s2$  can be replaced by:

$$tr : s1 \rightarrow_{op(x)[G1 \text{ or } G2]/acts} s2$$

A further transformation that can be applied is to replace a feature  $v$  by a constant value  $e$  throughout a state machine, if  $v$  is initialised to  $e$  on the initial transition of the state machine, and is never subsequently modified.

A group  $K$  of states can be merged into a single state  $k$  if:

1. All transitions between the states of  $K$  have no actions.
2. All transitions which exit the group  $K$  are triggered by events distinct from any of the events that trigger internal transitions of  $K$ . If two transitions that exit  $K$  have the same trigger but different target states or actions, they must have disjoint guard conditions.
3. Each event  $\alpha$  causing exit from  $K$  cannot occur on states within  $K$  which are not the explicit source of a transition triggered by  $\alpha$ .

Two states can also be merged if their sets of outgoing (or incoming) transitions are equivalent [3].

In the *doorcloses* example, the states  $s2$  and  $s3$  can be merged, and the two incoming transitions to the merged state can then be merged, to give a condition  $dest \neq lfloor$ .

The transformations have been specified as rules in the UML-RSDS model transformation language [7]. For example, transition merging can be defined by:

```
mergeTransitionPair(t1 : Transition, t2 : Transition) : Transition
pre: t1.source = t2.source and t1.target = t2.target and
     t1.trigger = t2.trigger and t1.effect = t2.effect
post:
  Transition.allInstances() → includes(result) and
  result.ocIsNew() and
  result.source = t1.source and result.target = t1.target and
  result.trigger = t1.trigger and result.effect = t1.effect and
  result.guard = or(t1.guard, t2.guard) and
  t1.isDeleted() and t2.isDeleted()
```

An algorithm combining these state machine slicing transformations has been implemented in the UML2Web tool, and applied to several complex examples, including manufacturing control systems based upon [10]. Slicing with respect to a subset  $V$  of the actuators of a control system produces a subcontroller which explicitly identifies which sequences of events and which subset of sensors actually determine the actuator settings, supporting semantic analysis and modularisation of controllers.

## 4 Slicing of Communicating State Machines

The above slicing approach can be extended to systems which consist of multiple communicating state machines, attached to linked objects, provided that the

communication dependencies  $M1 \rightarrow M2$  ( $M1$  sends messages to  $M2$ ) form a tree structure. The data of a state machine then also includes implicitly the data of all machines directly or indirectly subordinate to it (ie, the data of objects whose operations are invoked from the state machine).

The same concepts of  $<_{syn}$  and  $=_{sem}$  can be used as for single state machines, but with respect to the full data of each state machine, including the data of subordinate (supplier) machines.

## 5 Related Work

Several approaches have been defined for the slicing of state machine models, such as Korel [4] and Clark [2]. These are based upon generalisations of program slicing using concepts of node post-dominance [9] within graphs. However these approaches involve the construction of potentially a possibly very large data-and-control flow graphs. Our technique is based instead upon the semantic notion of path-predicates, as used in static analysis tools such as SPADE. This technique assigns to each program path a predicate which defines how the values of variables at the end state of the path relate to the values at the start state, over all executions of the path. We approximate these predicates by computing the sets  $V_x$  of variables in the path predicate for each state in the state machine.

*Acknowledgement* This research has been supported by the SLIM EPSRC project.

## References

1. I. Bruckner, H. Wehrheim, *Slicing Object-Z Specifications for Verification*, ZB 2005, LNCS 3455, Springer-Verlag, pp. 414–433, 2005.
2. D. Clark, *Amorphous Slicing for EFSMs*, PLID' 07, 2007.
3. L. Ilie, R. Solis-Oba, S. Yu., *Reducing the size of NFAs by using Equivalences and Preorders*, CPM 2005, LNCS 3537, pp. 310–321, 2005.
4. B. Korel, I. Singh, L. Tahat, B. Vaysburg, *Slicing of State-based Models*, ICSM '03, 19th IEEE International Conference on Software Maintenance, IEEE Press, 2003.
5. K. Lano, *Constraint-Driven Development*, Information and Software Technology, 50, 2008, pp. 406–423.
6. K. Lano (ed.), *UML 2 Semantics and Applications*, Wiley, 2009.
7. K. Lano, S. Kolahdouz-Rahimi, *Model Migration Transformation Specification in UML-RSDS*, TTC 2010, Malaga, Spain, 2010.
8. OMG, *UML superstructure, version 2.1.1. OMG document formal/2007-02-03*, 2007.
9. V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, *A New Foundation for Control Dependence and Slicing for Modern Program Structures*, ACM Trans. Prog. Lang. and Sys. Vol 29, No 5, August 2007.
10. A. Sanchez, E. Aranda-Bricaire, F. Jaimes, E. Hernandez, A. Nava, *Synthesis of product-driven coordination controllers for a class of discrete-event manufacturing systems*, Elsevier Science preprint, 2009.
11. C. Snook, P. Wheeler, M. Butler, *Preliminary Tool Extensions for Integration of UML and B*, IST-2000-30103 deliverable D4.1.2, 2003.

12. M. Weiser, *Program slicing*, IEEE Transactions on Soft. Eng., 10, July 1984, pp. 352–357.
13. F. Wu, T. Yi, *Slicing Z Specifications*, ACM Sigplan, vol. 39 (8), August 2004.