Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

# Slicing Techniques for UML Models

Kevin Lano[a] Shekoufeh Kolahdouz-Rahimi[a]

a. Dept. of Informatics, 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 and behaviour 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, single state machines, and communicating sets of state machines.

Keywords Slicing, UML, Model transformation

## 1 Introduction

Slicing of programs has been a widely-used analysis technique for many years [31, 9] and has also been used for reverse-engineering and re-factoring of code. In general this technique considers a specific point within the program code, such as the end point of the program, and a set of variables of interest at this point, and calculates a subset of the program statements which can affect the variables at the selected point, discarding any statements which do not contribute to the values of the variables of interest at the selected point.

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 [32] and Object-Z [3] have been defined, based on variants of the concepts of control and data dependence used to calculate slicing for programs. However, UML contains both declarative elements, such as pre- and post-conditions, and imperative elements, such as state machines and activities, so that slicing techniques for UML must treat both aspects in an integrated manner.

In the formulation of Harman and Danicic [8], slicing is generalised to include any form of software artifact: a slice is considered to be a transformed version $S$ of an artifact $M$ which has a lower value of some syntactic complexity measure, but an equivalent semantics with respect to the sliced data:

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

Kevin Lano, Shekoufeh Kolahdouz-Rahimi. *Slicing Techniques for UML Models*. In Journal of Object Technology, vol. 10, 2011, pages 11:1–49. doi:10.5381/jot.2011.10.1.a11

The form of slicing used depends 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 [8].

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 produce a target model satisfying the $<_{syn}$ and $=_{sem}$ relations with respect to the source model, so that the successive applications of the transformations will result in a slice which also satisfies these relations compared to the original model.

Existing work on UML model slicing has focussed upon unstructured state machines, considered in isolation from other UML models [11, 1]. However in practical systems, state machines usually depend upon some data model such as a class diagram, and use mechanisms such as state nesting and intercommunication by message passing.

We will extend the techniques of [1] to encompass the combined slicing of class diagrams and state machines, and communication between state machines. In this paper we will focus specifically on *reactive systems*, consisting of finite configurations of objects representing system components, such as controllers, linked together in an acyclic manner.

In general, applying slicing at a high level of abstraction simplifies the calculation of the slice, and means that it is possible to detect specification flaws at an early development stage, thus reducing development costs.

Slicing may also be used to compare a new version of a system with a previous version, to establish that the behaviour of the new version is consistent with the previous version. Slicing can be used to modularise and decompose a system, particularly to factor large control algorithms into smaller and more cohesive parts.

Section 2 considers the slicing of abstract class diagram models consisting of classes, associations, attributes and constraints. Section 3 considers the slicing of class diagrams which additionally contain operations and are linked to state machines to define object life histories. Section 4 defines slicing techniques for state machines for classes or operations, Section 5 defines slicing techniques for systems of communicating state machines. Section 6 evaluates the approach on application case studies. Finally, Section 7 considers related work.

## 2 Slicing of Constraint-based Class Diagrams

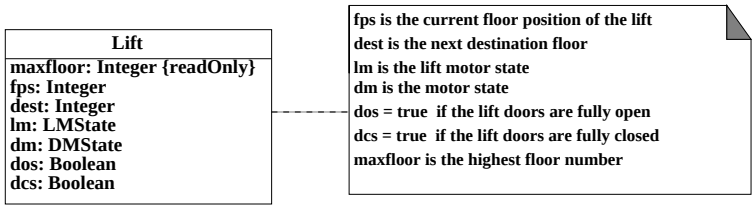
We consider first the most abstract form of UML specification using a class diagram to define the structure of classes and associations, and constraints in OCL to define behaviour by means of invariant constraints of classes. At this initial stage in the development process no explicit operations are considered, and the model can be regarded as a *computation-independent model* (CIM) in terms of the model-driven architecture (MDA) [24]. The semantics of such a model $M$ consists of the set of possible value assignments which can be made to the features of the model, that is, the set of possible objects of each class and the values of the attributes and association ends of these objects.

The range of UML constructs considered here corresponds to the UML-RSDS subset of UML, which is specifically oriented to the specification and design of reactive

systems [17]. For such class diagrams there is an axiomatic semantics, which defines the meaning of a class diagram $M$ as a set $\Gamma_M$ of sentences in a first-order logic which extends OCL [20]. We write $\Gamma_M \models \psi$ for a sentence $\psi$ of the language of $M$ to mean that $\psi$ is true in each possible instantiation of $M$.

In this paper we will further restrict consideration to *reactive system* specifications, in which attributes of classes are categorised as representing *sensors* (input data), *actuators* (output data) or internal data. In such systems classes represent (the types of) particular system components such as controllers or subcontrollers (which determine what actions to take when particular input events are received), or data stores holding data used in the processing. Associations are unidirectional and their end properties are *readOnly* (that is, they cannot be modified after initialisation). Instances of the classes represent specific controller or data store instances, the client-supplier relation between these objects must be acyclic.

Figure 1 shows an example of this style of specification, for a lift control system. The lift actuators are *dm* and *lm* and the lift sensors are *fps*, *dest*, *dos* and *dcs*. *maxfloor* is an internal constant parameter.



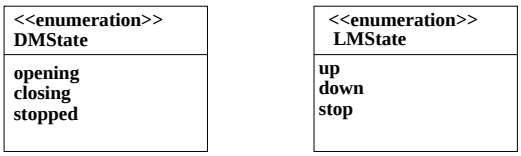




Figure 1 – Abstract lift control system

The *data features* of such a system are all the attributes of classes in the system, and all member end properties of associations.

In the lift example, there are the following class invariant constraints of *Lift*:

$$
\begin{array}{l}
(dest > fps \ and \ dcs = true \ implies \ lm = up) \ and \\
(dest < fps \ and \ dcs = true \ implies \ lm = down) \ and \\
(dest = fps \ implies \ lm = stop) \ and \\
(dest = fps \ and \ dos = true \ implies \ dm = stopped) \ and \\
(dest = fps \ and \ dos = false \ implies \ dm = opening) \ and \\
(dest \neq fps \ and \ dcs = false \ implies \ dm = closing) \ and \\
(dest \neq fps \ and \ dcs = true \ implies \ dm = stopped) \ and \\
(dcs = false \ implies \ lm = stop) \ and \\
\\
(dos = true \ implies \ dcs = false) \ and \\
0 \leq dest \ and \ dest \leq maxfloor \ and
\end{array}
$$

$$0 \leq fps \quad and \quad fps \leq maxfloor$$

These constraints implicitly define the behaviour of the lift. For example, the first constraint expresses that if the destination is above the current floor and the doors are fully closed, then the lift will be moving upwards.

A specific configuration of this system will consist of a number of unconnected lift components $l1 : Lift$, $l2 : Lift$, etc. Each instance has a copy of the data features of $Lift$: $l1.dest$, $l1.fps$, etc., and these satisfy the class invariants instantiated for this instance, ie.:

$$l1.dest > l1.fps \ and \ l1.dcs = true \quad implies \quad l1.lm = up$$

and so forth.

A more complex system is a reactive system which controls a robot with two light sensors *lsleft*, *lsright*, a distance sensor *distance*, and two motors, *mleft*, *mright*, for the left and right wheels of the robot. Figure 2 shows the physical robot configuration, Figure 3 the class diagram of a control system which combines line-following behaviour (the robot is steered so that it remains on dark areas of a surface and avoids light areas) with collision avoidance (the robot is halted if it comes too close to an obstacle) via a subsumption combinator [2].

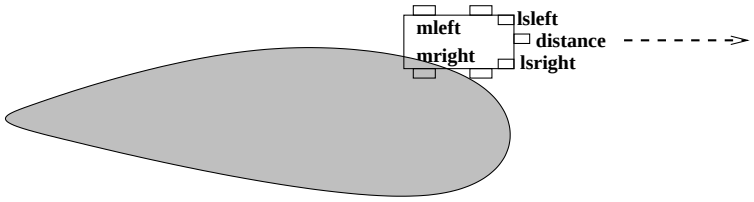



Figure 2 – Robot configuration

The *LineFollower* constraints are:

$$lsleft = on \ and \ lsright = off \quad implies \quad mleft = on \ and \ mright = half$$
$$lsleft = on \ and \ lsright = on \quad implies \quad mleft = on \ and \ mright = on$$
$$lsleft = off \ and \ lsright = on \quad implies \quad mleft = half \ and \ mright = on$$
$$lsleft = off \ and \ lsright = off \quad implies \quad mleft = on \ and \ mright = on$$

where *half* indicates half speed, so that in the first case the robot steers to the right if it detects a dark area ($lsright = off$) under its right sensor *lsright* and light ($lsleft = on$) under its left *lsleft*. In this specification there are only sensors and actuators (*mleft* and *mright*), so that the component has no memory: its output is determined by the current values of its sensors. We could introduce an internal variable *direction* to describe more complex behaviour (separate modes of reversing and forward movement).

Collision avoidance is specified by:

$$dist < 10 \quad implies \quad mleft = off \ and \ mright = off$$
$$dist \geq 10 \quad implies \ mleft = none \ and \ mright = none$$

These are then combined by subsumption, with collision avoidance having priority over line following.

The combinator is specified as:

$$in1 = none \quad implies \quad out = in2$$
$$in1 \neq none \quad implies \quad out = in1$$

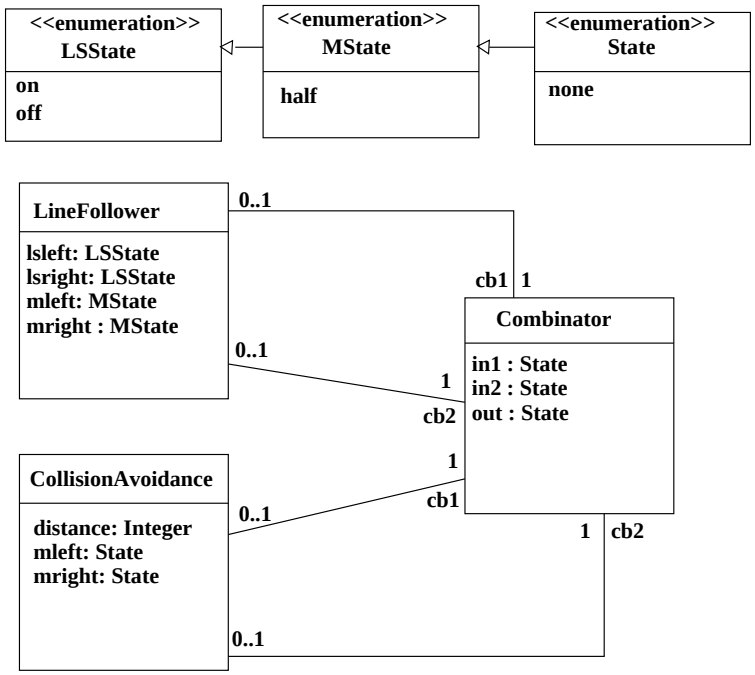



Figure 3 – Robot control system abstract class diagram

The combinator $cb1$ arbitrates between the line follower and collision avoidance commands for *mleft*, with the collision avoidance component having priority ($in1$ input to $cb1$). $cb2$ likewise arbitrates commands for *mright*.

The motivation for slicing such specifications is to facilitate validation and consistency analysis: if we wish to prove some validation property $\varphi$ of a specification, it should be sufficient to consider only those data features upon which the features of $\varphi$ depend, rather than all features of the specification. In UML-RSDS we translate specifications into the B language to perform validation via proof and animation [16]. The complexity of such proofs can be reduced by only translating the parts of a specification directly relevant to a given property, enabling automated proof of correctness.

Slicing will be carried out upon class invariants by considering the *predicates* $P$ of which they are composed. A predicate is a truth-valued formula, not containing the *and* operator except in the antecedent of a top-level implication. We assume that invariants have been expressed in the form of a conjunction of predicates, as in the examples above.

Class invariant predicates are classified as either *assertions*: properties which are expected to be invariant for objects of the class, but which should not or cannot be maintained by modifying data features (for example, they are environmental assumptions rather than behaviour obligations of the system), or as *effective*: defining what changes of actuator or internal data are necessary when a sensor or internal data feature changes value. There are also system-level constraints defining the configuration of the system: what components it consists of, and how these are connected.

Effective predicates have the form

$$L \ \ implies \ \ R$$

where $R$ is a formula $f = e$, with $f$ an internal or actuator attribute or variable.

The occurrence of $f$ in this formula is termed a *writable occurrence* of $f$, and $f$ is called the *writable feature* or *writable variable* of the constraint. $L$ may be omitted. $L$ is the *test* part of the predicate, $e$ the *value* part, these should normally refer only to sensor and internal data.

Assertions do not contain writable occurrences of actuator or internal data. Normally they contain only internal and sensor data. In the lift system

$$dos = true \ implies \ dcs = false$$

is an assertion, because it relates the values of two sensors.

Configuration constraints declare specific objects (components) and define the links between them, for example:

$lf : LineFollower$
$ca : CollisionAvoidance$
$cb1 : Combinator$
$cb2 : Combinator$
$lf.cb1 = cb1$
$lf.cb2 = cb2$
$ca.cb1 = cb1$
$ca.cb2 = cb2$
$lf.mleft = cb1.in2$
$lf.mright = cb2.in2$
$ca.mleft = cb1.in1$
$ca.mright = cb2.in1$

for the robot controller.

The *variables* $Variables_M$ of a system $M$ are the distinct instances $c.att$ of attributes $att$ of its classes, where $c$ is an instance in the system of the class $C$ that owns $att$. Attribute instances equated by the configuration constraints are considered to be the same variable, so the robot control system has 9 variables:

$$\{lf.lsleft, lf.lsright, lf.mleft, lf.mright, ca.distance, ca.mleft, ca.mright, cb1.out, cb2.out\}$$

$lf.mleft$, $lf.mright$, $ca.mleft$, $ca.mright$, are internal variables of the complete system, $lf.lsleft$, $lf.lsright$, $ca.distance$ are sensors, and $cb1.out$, $cb2.out$ actuators.

The last five predicates of the lift invariants are assertions, the others are effective. Together, the effective constraints define the permissible states of the lift actuators $dm$ and $lm$ in all possible combinations of sensor settings (the lift sensors are $fps$, $dest$, $dos$ and $dcs$). All predicates of the robot controller are effective except for the configuration constraints.

Before slicing a specification $M$, it is normalised, so that the class invariants are in the form of conjunctions of predicates. Formulae $P \ implies \ R \ and \ Q$ are rewritten as $P \ implies \ R$ and $P \ implies \ Q$, formulae $(P \ and \ Q) \ or \ R$ are rewritten as $(P \ or \ R) \ and \ (Q \ or \ R)$, and so forth.

The definition of syntactic reduction we will use for abstract class diagrams $M$ is the following:

$$S <_{syn} M$$

if $S$ has a subset of the elements of $M$: the classes of $S$ are a subset of the classes of $M$, the components of $S$ are a subset of those of $M$, likewise for types and variables. The variables of $S$ have the same types in $S$ and $M$. This is therefore a structure-preserving form of slicing.

For such an $S$, the semantic equality

$$S =^{V}_{sem} M$$

holds for a given set $V$ of variables of $M$, if $V \subseteq \mathit{Variables}_S$, and if any value assignment of the variables of $M$ valid for $M$ is also valid (when restricted to the variables of $S$) for $S$. The relation $=^{V}_{sem}$ is reflexive and transitive, but not symmetric.

If $S <_{syn} M \wedge S =^{V}_{sem} M$, then $\Gamma_S \models \varphi$ implies $\Gamma_M \models \varphi$, for a sentence $\varphi$ containing only variables of $V$, since any instantiation of $M$ can be restricted to an instantiation of $S$.

A stronger semantic relation $\equiv^{V}_{sem}$ requires in addition that any value assignment valid for $S$ can be extended to a valid value assignment for $M$. This relation is also transitive and reflexive. The $\equiv^{V}_{sem}$ relation allows us to deduce the reverse implication between $\Gamma_M$ and $\Gamma_S$: if $\Gamma_M \models \varphi$, for a sentence $\varphi$ containing only variables of $V$, so does $\Gamma_S$, since any instantiation of $S$ can be extended to an instantiation of $M$ satisfying the same $S$-sentences. In particular if $S$ is *weakly satisfiable* (that is, it has at least one class which can be instantiated with an object) then so is $M$.

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

For $=^{V}_{sem}$ we only use effective predicates to calculate the set of variables of $S$: these are $V$ together with any variables which a variable of $V$ depends on via the effective predicates. For $\equiv^{V}_{sem}$ it is necessary also to consider all the constraints of $M$, including assertions.

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

- *The write frame* $wr(p)$ is the set of variables written to in $p$. If $p$ is effective, this set is the single writable variable of $p$, otherwise it is all variables in $p$.
- *The read frame* $rd(p)$ is the set of variables read in $p$. If $p$ is effective this is the set of all variables occurring in the test or value expressions in $p$. For other predicates $rd(p)$ is empty.
- The set of variables used in $p$ is $var(p)$:

  $$var(p) = rd(p) \cup wr(p)$$

- The internal data-dependencies of an effective predicate $p$ are then:

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

  and for an assertion predicate

  $$dep(p) \;=\; var(p) \times wr(p)$$

The dependency relation $\rho_M$ of the class diagram is the non-reflexive transitive closure of the union of the $dep(p)$ for all effective constraint predicates.

The slice $S$ has all the effective constraints whose writable variable is in $V' = V \cup \rho_M^{-1}(\!|\ V\ |\!)$, and all assertions whose variables are all in $V'$. Variables of $M$ which do not occur in $V'$ can be omitted from $S$. Components which have no variable in $V'$ can also be omitted from $S$ and the configuration constraints.

This slicing definition satisfies the $<_{syn}$ and $=^V_{sem}$ relations, because the elements and constraints of $S$ are a subset of those of $M$. In particular, any value assignment for the variables of $M$ that satisfies the constraints of $M$ will also satisfy those of $S$ when restricted to $V'$.

In the lift example, $\rho_M$ is the relation

$$\{dest \mapsto lm, fps \mapsto lm, dcs \mapsto lm,$$
$$dest \mapsto dm, fps \mapsto dm, dos \mapsto dm, dcs \mapsto dm\}$$

A $=^{\{lm\}}_{sem}$ slice $S$ therefore does not need to include $dm$, $maxfloor$ or $dos$.

The set of constraints in the slice is:

$$(dest > fps \ \ and \ \ dcs = true \ \ implies \ \ lm = up) \ \ and$$
$$(dest < fps \ \ and \ \ dcs = true \ \ implies \ \ lm = down) \ \ and$$
$$(dest = fps \ \ implies \ \ lm = stop) \ \ and$$
$$(dcs = false \ \ implies \ \ lm = stop) \ \ and$$

$$0 \leq dest \ \ and \ \ 0 \leq fps$$

A property such as

$$lm = up \ \ implies \ \ dest > fps$$

can be proved in $\Gamma_S$ and therefore deduced also for $\Gamma_M$.

In the robot control example, a proof of

$$ca.distance < 10 \ \ implies \ \ cb1.out = off$$

only needs to consider the six variables in $V1 = \{ca.distance, ca.mleft, lf.mleft, lf.lsleft, lf.lsright, cb1.out\}$ instead of 9 in the complete system, because $V1$ is the set of variables which $\{ca.distance, cb1.out\}$ depend upon via $\rho_M$. The $cb2$ component can be removed.

For the stronger semantic relation $\equiv^V_{sem}$, the slice needs to be defined using the dependency relation $\rho'_M$, which includes the data dependencies of assertions in the relation: if variable $f$ occurs in an assertion then it is considered dependent on all variables in the assertion. $\rho'_M$ is then the non-reflexive transitive closure of the dependencies for effective constraints together with the assertion dependencies, and $S$ is defined as above from $\rho'_M$ instead of from $\rho_M$.

In the lift example, $\rho'_M$ is the relation

$$\{dest \mapsto lm, fps \mapsto lm, dcs \mapsto lm, dos \mapsto lm, maxfloor \mapsto lm,$$
$$dest \mapsto dm, maxfloor \mapsto dm, fps \mapsto dm, dos \mapsto dm, dcs \mapsto dm\}$$

The resulting set of constraints in the slice for $V = \{lm\}$ is:

$$(dest > fps \ \ and \ \ dcs = true \ \ implies \ \ lm = up) \ \ and$$
$$(dest < fps \ \ and \ \ dcs = true \ \ implies \ \ lm = down) \ \ and$$
$$(dest = fps \ \ implies \ \ lm = stop) \ \ and$$

$(dcs = false \ implies \ lm = stop) \ and$

$(dos = true \ implies \ dcs = false) \ and$
$0 \leq dest \ and \ dest \leq maxfloor \ and$
$0 \leq fps \ and \ fps \leq maxfloor$

# 3 Slicing of Operation-based Class Diagrams

A more explicit form of class diagram specification includes operations of classes, together with pre- and post-condition constraints to define these operations, and state machines of classes to define the behaviour of their instances. These specifications can be considered to be at the platform-independent model (PIM) level in terms of the MDA. We assume that the client-supplier relation between different classes is acyclic for such specifications. Operations are assumed to be deterministic.

Explicit specifications $M$ consist of a set of instances (the components of $M$) of classes $C$ of $M$, each class has an associated state machine $SM_C$ as its *classifierBehavior* (Chapter 15 of [23]), this state machine defines what sequences of operations can be applied to objects of $C$, and under what conditions.

Figure 4 shows the metamodel for state machines which we use, this is based upon a subset of UML state machine notation (Chapter 15 of [23]). Dashed lines indicate associations which are not part of the metamodel but are used to hold computed data during the slicing process. $st.writes$ for a statement $st$ is the write frame $wr(st)$ of $st$. $st.reads$ for a statement $st$ is the read frame $rd(st)$ of $st$. We will only consider transition effects that are sequence statements composed of basic statements: assignments or, for communicating state machines, invocations of operations on supplier objects. We denote the set of states of a state machine $Sm$ by $States_{Sm}$ and the set of transitions by $Transitions_{Sm}$. $Variables_{Sm}$ is its set of variables.

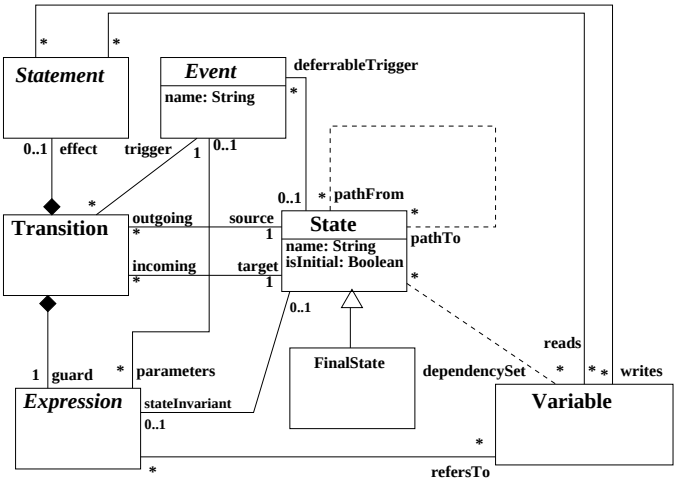



Figure 4 – State machine metamodel

Individual transitions $tr : Transition$ have the general form

$$src \rightarrow_{op(x)[G]/acts} trg$$

where $src = tr.source$, $trg = tr.target$, the invocation of $op(x)$ is $tr.trigger$, with

parameters $x$, $G = tr.guard$ and $acts = tr.effect$ (an assignment, skip, operation call or a sequence of such statements). Implicitly the precondition of $op(x)$ is conjoined to $G$, and the complete effect of the transition when it fires is given by the sequence of the effect of $op(x)$ followed by $acts$ (page 591 of [23]). Therefore in slicing such specifications, the effects of both the operation execution and the explicit transition need to be considered.

Typically, the operation postcondition defines local data modifications of the component, whilst the transition effect invokes operations of other components.

The state machine $c.SM_C$ of an instance $c : C$ has states $\{c.st \mid st \in States_{SM_C}\}$, transitions $\{c.tr \mid tr \in Transitions_{SM_C}\}$, with attributes $x$ in $tr.effect$ replaced by variables $c.x$ in $(c.tr).effect$, and similiarly for $tr.guard$. $c.tr : c.src \rightarrow c.trg$ if $tr : src \rightarrow trg$, and $(c.tr).trigger = c.(tr.trigger)$.

An explicit specification $M$ has an interface $M_I$ consisting of the set of operations which may be externally invoked upon $M$. For reactive systems this is usually the set of input sensor events that it reacts to: that is, the union of $\alpha N$ for the components $N = c.SM_C$ of $M$ which have no clients within $M$, where the set of input events of state machine $Sm$ is denoted by $\alpha Sm$.

Because operation-based class diagrams and state machines are inter-related, the slicing definition for such class diagrams depends on the existence of slicing definitions for state machines. In this section we assume the slicing definition $=^{s,V}_{sem}$ from Section 4 is used for state machines, and we describe how such slicing affects the class diagram linked to the state machines.

If an explicit 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.

Operations may alternatively be defined by a structured activity instead of preconditions and postconditions, using the statement language of Figure 5.

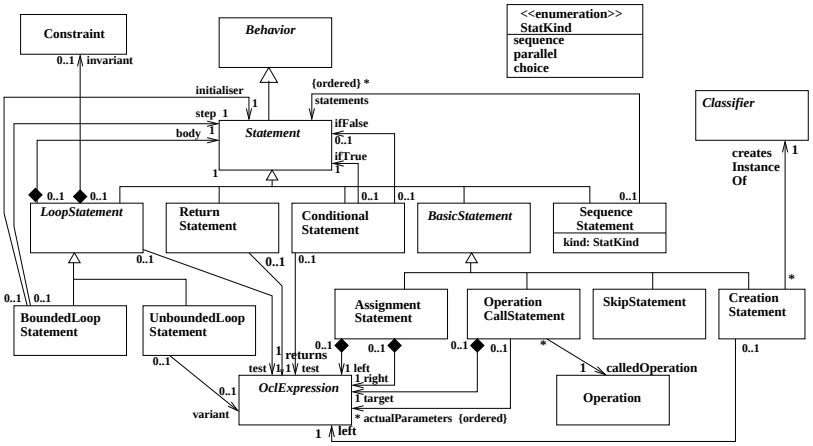



Figure 5 – Statement metamodel

In this section we will define slicing for such combined class diagram and state machine specifications, by using a function (defined in Appendix A)

$$slice : Statement \times \mathbb{F}(Variable) \rightarrow Statement$$

which computes the statement $slice(stat, V)$ which has the same effect on $V$ as $stat$. We will extend $slice$ to include slicing of operation definitions as a generalised form

of statement.

Figure 6 shows a refinement of the lift control system using this style of specification.

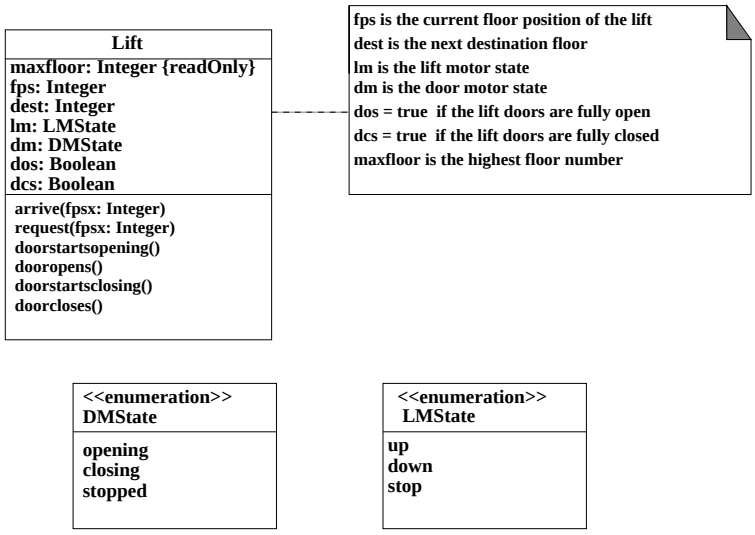



Figure 6 – Lift control system refinement

The explicit operations of the lift can be specified as follows:

$init()$
`post:`
$fps = 0 \ and \ dest = 0 \ and \ dcs = false \ and$
$dos = true \ and \ lm = stop \ and \ dm = stopped$

$arrive(fpsx : Integer)$ `/* Lift arrives at floor` $fpsx$ `*/`
`pre:` $0 \leq fpsx \ and \ fpsx \leq maxfloor$
`post:`
$fps = fpsx \ and$
$(fpsx = dest \ implies \ lm = stop) \ and$
$(fpsx = dest \ implies \ dm = opening) \ and$
$(dcs = false \ implies \ lm = stop)$

$request(destx : Integer)$ `/* Request to go to floor` $destx$ `*/`
`pre:` $0 \leq destx \ and \ destx \leq maxfloor$
`post:`
$dest = destx \ and$
$(destx > fps \ and \ dcs = true \ implies \ lm = up) \ and$
$(destx < fps \ and \ dcs = true \ implies \ lm = down) \ and$
$(destx = fps \ implies \ lm = stop) \ and$
$(destx = fps \ and \ dos = true \ implies \ dm = stopped) \ and$
$(destx = fps \ and \ dos = false \ implies \ dm = opening) \ and$
$(destx \neq fps \ and \ dcs = false \ implies \ dm = closing) \ and$
$(destx \neq fps \ and \ dcs = true \ implies \ dm = stopped)$

$doorstartsopening()$
`pre:` $dcs = true$
`post:`

$dcs = false \ \ and \ \ lm = stop$

*doorstartsclosing*()
**pre:** $dos = true$
**post:**
$dos = false$

*doorcloses*()
**pre:** $dcs = false \ \ and \ \ dos = false$
**post:**
$dcs = true \ \ and \ \ dm = stopped \ \ and$
$(dest > fps \ \ implies \ \ lm = up) \ \ and$
$(dest < fps \ \ implies \ \ lm = down)$

*dooropens*()
**pre:** $dos = false \ \ and \ \ dcs = false$
**post:**
$dos = true \ \ and \ \ dm = stopped$

The *init* operation is implicitly invoked when an object of the class is created ([20], Chapter 6). An operation is included for each input sensor event of the component. Such specifications may be derived from the implicit specifications: if the basic effect of an operation writes to a variable $v$, setting it to value $vx$, then:

- All assertions $p$ with $v \in var(p)$ are included in the precondition of $op$, in the form $p[vx/v]$.
- All effective constraints $p$ with $v \in rd(p)$ are included in the postcondition of $op$, in the form $p[vx/v]$.

For example, the definition of *request*, which has the basic effect $dest = destx$, has been derived in this manner.

Figure 7 shows the state machine for the *Lift* class. State invariants such as $dm = opening$ do not contribute to data or control dependencies.

A *history* $e$ of an explicit specification $M$ is a finite sequence of invocations of operations from $M_I$ on its components. A history is valid if operations are only invoked on existing instances, and for which the operation precondition is true at the point of call, in addition the history actions specific to each instance $c$ of class $C$ should conform to the state machine $c.SM_C$ of $c$.

For example, if the lift control system consists of a single instance $cx : Lift$, then

$$cx.request(10), cx.doorstartsclosing(), cx.doorcloses(), cx.arrive(1)$$

is a valid history for the system, if $cx.maxfloor = 15$.

$$lf.lslefton(), lf.lsrighton(), ca.setdistance(5)$$

is a valid history for the robot control system example with objects $lf : LineFollower$ and $ca : CollisionAvoidance$.

The *state machine* $SM_M$ of a specification $M$ has states all tuples of states from the state machines of its individual components:

$$States_M = States_{c1.SM_{C1}} \times ... \times States_{cn.SM_{Cn}}$$

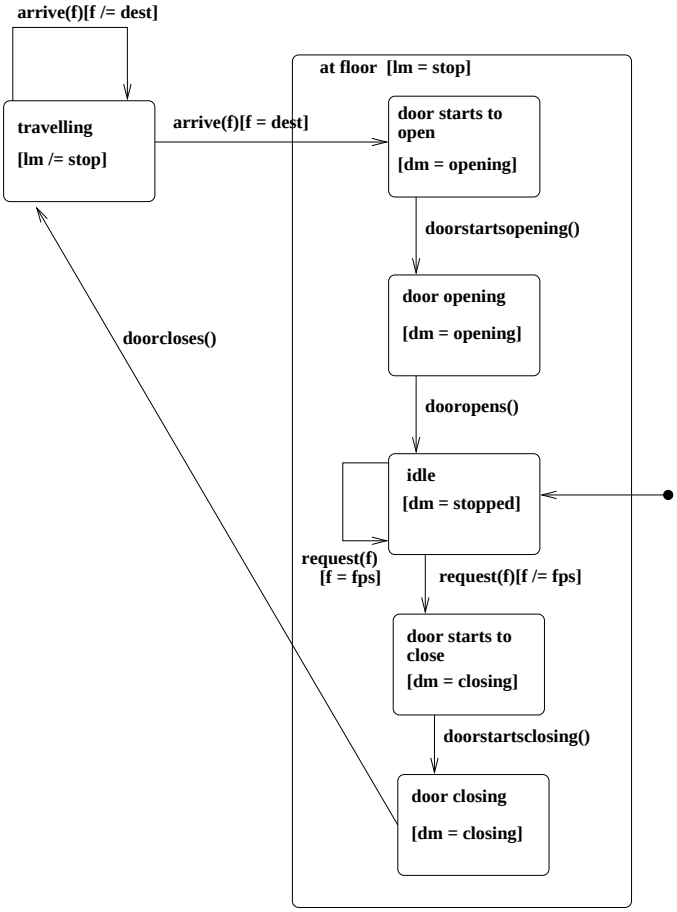



Figure 7 – Lift state machine $SM_{Lift}$

where the $ci$ are all those components which have no clients within $M$. Each $ci$ is an instance of some class $Ci$ of the specification. The initial state of $SM_M$ is the tuple of initial states from these components. The transitions of $SM_M$ are derived from those of each component: if $tr : src \rightarrow trg$ in $ci.SM_{Ci}$ and $(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$ is a tuple of states from other the components, then

$$tr' : (s_1, \ldots, s_{i-1}, src, s_{i+1}, \ldots, s_n) \rightarrow (s_1, \ldots, s_{i-1}, trg, s_{i+1}, \ldots, s_n)$$

is a transition of $SM_M$, with the same trigger, guard and effect as $tr$.

In the robot control system there are component state machines for the $lf$, $ca$, $cb1$ and $cb2$ components (Figure 8). Here $lf$ and $ca$ are the components without clients. We have omitted some transitions, in the $low$ states there are self-transitions $set2on()/out := on$, $set2half()/out := half$, etc. The input event $set1on$ can be removed since it is never invoked in this system.

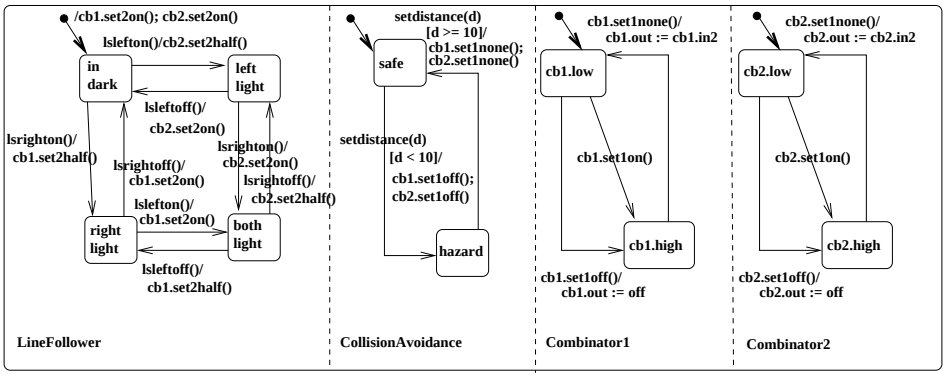



Figure 8 – Robot component state machines

When a specification $M$ is sliced to a specification $S$, each of its component state machines may be abstracted, because some of their states may be merged (Section 4). Such state merging corresponds to an abstraction map $\sigma_N : States_N \rightarrow States_{N'}$ where $N$ is the original component, and $N'$ the sliced component. This mapping can then be lifted to an abstraction map $\sigma$ from $States_M$ to $States_S$. Likewise for transitions.

The same notion of syntactic reduction

$$S <_{syn} M$$

as in Section 2 is used for operation-based class diagrams, with the additional requirement that the operations of each class $C$ of $S$ are a subset of the operations of $C$ in $M$, and the predicates of each operation postcondition $Post_{op}$ of $C$ in $S$ are a subset of the corresponding set of postcondition predicates for $op$ in $M$. The input events of $M$ and $S$ should be the same: $M_I = S_I$.

Semantic equivalence

$$S =^{s,V}_{sem} M$$

holds with respect to a given state tuple $s$ in $States_M$, and set $V \subseteq Variables_S$ if:

1. any valid history $e : \text{seq}(M_I)$ of $M$ with end state $s$ is also a valid history of $S$ with end state $\sigma(s)$

2. if $e$ is applied to both models, starting from the same initial values for variables in the respective initial states, then the values of the variables in the slice set $V$ in $S$ in $\sigma(s)$ are equal to the values of these variables in $s$ in $M$.

Formally:

$$\forall\, e : \mathrm{seq}(M_I);\ \forall\, v_0, v \cdot \\ \quad initial_M[Variables_M = v_0] \longrightarrow_e^M s[V = v] \ \Rightarrow \\ \quad\quad \sigma(initial_M)[Variables_S = v_0'] \longrightarrow_e^S \sigma(s)[V = v]$$

where $src[P] \longrightarrow_e^N trg[Q]$ denotes that there is a sequence of transitions in $N$ triggered in order by the event sequence $e$, starting from state $src$ with $P$ true, and ending with state $trg$ with $Q$ true in $trg$. $v_0'$ is the subset of values of $v_0$ used to initialise $Variables_S$.

The relation $=_{sem}^{s,V}$ is reflexive and transitive, but not symmetric.

The slice $S$ can be determined by examining the data dependencies of the constraints of the original model, and the state machines of its components.

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

- The write frame $wr(p)$ is the set of variables written to in $p$. If $p$ is effective, this set is the single writable variable of $p$, which cannot be an input parameter of $op$, otherwise it is the set of variables of $p$ not in $pre$ form $v@pre$ in $p$, and that are not input parameters.

  In particular, for the basic effect $sv = svx$ of an operation $setsv(svx : T)$, the write frame is $\{sv\}$.

- The read frame $rd(p)$ is the set of variables read in $p$. If $p$ is effective this is the set of all variables occurring in the test or value expressions in $p$. For a predicate $p$

  $$E \ implies \ v = e$$

  $v$@pre is included in $rd(p)$ since in the case that $E$ is false, the value of $v$ remains $v$@pre by default.

  For other predicates $rd(p)$ is the set of variables which are either in $pre$ form, or are input parameters.

- The internal data-dependencies of a postcondition predicate $p$ are then:

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

The *write frame* $wr(op)$ of an operation $op$ is the union of $wr(p)$ for the predicates $p$ in its postcondition. For example, $wr(arrive) = \{fps, lm, dm\}$.

*Control dependency* in a program or specification occurs when the value of a variable $v$ at a certain point influences which updates will be applied to a variable $w$ [27]. For example, in:

$$if \ v > 0 \ then \ w := 2 \ else \ w := 3$$

$w$ is control dependent upon $v$.

The variables in the predicates in the postcondition of an operation can be considered to be control dependent on the variables in the predicates in the precondition

[3]. The rationale for this is that *pre* : $P$ *post* : $Q$ can be interpreted as "if $P$ holds, carry out the updates specified by $Q$, else perform an arbitrary update". Similarly, the effects of a transition are considered to be control dependent upon the transition guard.

At the level of particular variables, $f$, $g$, 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 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 $p$ (control dependency)
- $g \mapsto f$ if $g$ is an input parameter of the operation, or is a variable 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 $x \notin wr(op)$. ($x$ is not modified by $op$, so its value at the end of the operation is determined by its value at the start.)

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 variables not in $\rho_{op}^{-1}(\!|\ V\ |\!)$ cannot affect the value of any variable in $V$ at termination of the operation.

The $\rho_{op}$ dependencies of *arrive* in the lift control system are therefore:

$$\{maxfloor \mapsto fps, fpsx \mapsto fps, fpsx \mapsto lm,$$
$$dest \mapsto lm, fpsx \mapsto dm, dest \mapsto dm, dcs \mapsto lm,$$
$$lm \mapsto lm, dm \mapsto dm, maxfloor \mapsto lm, maxfloor \mapsto dm,$$
$$dest \mapsto dest, dcs \mapsto dcs, dos \mapsto dos, maxfloor \mapsto maxfloor\}$$

Slicing of statements from the activity language of Figure 5 can be carried out using standard data-slicing processes for procedural programs. For example, we have:

$$slice(x := e, V) \ =$$
$$\quad x := e \ \textit{if}\ x \in V$$
$$\quad \textit{skip otherwise}$$

and

$$slice(S_1;\ S_2, V) \ = \ slice(S_1, V_1);\ slice(S_2, V)$$

where $V_1$ is the set of variables whose values at commencement of an execution of $S_2$ can affect the values of $V$ at the end of the execution: $dependents(S_2, V) = V_1$. The definition of *slice* and *dependents* is given in Appendix A.

We can extend this definition to calculate the slice of operation executions, for operations $op(x : X)$ defined by precondition $Pre_{op}$ and postconditions $Post_{op}$, as follows. $slice(op(x : X), V)$ is defined by precondition $Pre_{op}$ and postcondition $Post'_{op}$, where $Post'_{op}$ includes all predicates $p$ of $Post_{op}$ such that $wr(p) \cap (r_{op}^{-1}(\!|\ V\ |\!) \cup V) \neq \{\}$. In other words, such that $p$ may affect the values of $V$ at termination of $op$.

For example, the slice of *arrive* with $V = \{dm\}$ is:

*arrive*(*fpsx* : *Integer*) `/* Lift arrives at floor` *fpsx* `*/`
`pre:` 0 ≤ *fpsx* *and* *fpsx* ≤ *maxfloor*
`post:`
(*fpsx* = *dest* *implies* *dm* = *opening*)

Compared to the dependency analysis of constraint-based class diagram specifications, additional dependencies between variables are included because the updated variables of operation postconditions are assumed to be dependent upon the variables of the precondition. But dependencies may be reduced because we take account of the restrictions of state machines of components in limiting the sequences of operation executions that can occur.

The following algorithm is used to compute a slice of a complete specification $M$, with respect to a state $s$ and set $V$ of variables of $SM_M$.

We associate a set $V_x$ of variables to each state $x$ of $SM_M$.

1. Initialise each $V_x$ with the empty set of variables, except for the target state $s$, which has the set $V$ of variables.

2. For each transition $tr : s1 \rightarrow s2$ of $SM_M$, add to $V_{s1}$ the set

$$var(tr.guard) \cup \rho_{op}^{-1}(\!|\ dependents(tr.effect,\ V_{s2})\ |\!)$$

of variables upon which $V_{s2}$ depends, via the version of the operation $op$ executed by this transition (with precondition the conjunction of $tr.guard$ and $Pre_{op}$). The overall effect of the transition is the sequential composition of $Post_{op}$ and $tr.effect$ (page 591 of [23]), so we need to compose $\rho_{op}$ and $dependents(tr.effect)$. Both the guard and $Pre_{op}$ contribute to the control dependencies.

The second step is iterated until a fixed point is reached. Each $V_x$ then represents the set of variables whose value in state $x$ can affect the value of $V$ in state $s$, on one or more paths from $x$ to $s$, by either control or data dependence. (Parameter values of operations along the paths may also affect $V$ in $s$). More generally, for any two states $x$ and $y$, the set $V_x$ includes the set of variables whose value in state $x$ can affect the value of $V_y$ in state $y$, on any path from $x$ to $y$.

Each transition effect and operation occurrence can then be sliced, as described above. This may produce different versions of the operation invoked at different points of the state machine, ie, $slice(op(x : X), W)$ for different sets $W$ of variables. To resolve this, we could combine the different postconditions of these versions into a single postcondition, conditioned by state membership of the source state:

`post:`
(*in* *s*1 *implies* *Post*1) *and* ... *and*
(*in* *sm* *implies* *Postm*)

where the *Posti* are the postconditions of the different versions. However, this adds to the syntactic complexity of the operation. Instead, we take the union of the sets of predicates in the different *Posti* as the overall sliced definition of *op*.

Let $V'$ be the union of the $V_x$ sets, for all states $x$ on paths from the initial state of $SM_M$ to $s$. The set of variables retained in the slice $S$ will be set equal to $V'$.

If the lift system is sliced with $V = \{dm\}$, then $V'$ is the set of all variables of the lift, with *lm* removed.

The new class invariant constraints are:

$(dest = fps \ and \ dos = true \ implies \ dm = stopped) \ and$
$(dest = fps \ and \ dos = false \ implies \ dm = opening) \ and$
$(dest \neq fps \ and \ dcs = false \ implies \ dm = closing) \ and$
$(dest \neq fps \ and \ dcs = true \ implies \ dm = stopped) \ and$

$(dos = true \ implies \ dcs = false) \ and$
$0 \leq dest \ and \ dest \leq maxfloor \ and$
$0 \leq fps \ and \ fps \leq maxfloor$

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

$init()$
`post:`
$fps = 0 \ and \ dest = 0 \ and \ dcs = false \ and$
$dos = true \ and \ dm = stopped$

$request(destx : Integer)$
`pre:` $0 \leq destx \ and \ destx \leq maxfloor$
`post:`
$dest = destx \ and$
$(destx = fps \ and \ dos = true \ implies \ dm = stopped) \ and$
$(destx = fps \ and \ dos = false \ implies \ dm = opening) \ and$
$(destx \neq fps \ and \ dcs = false \ implies \ dm = closing) \ and$
$(destx \neq fps \ and \ dcs = true \ implies \ dm = stopped)$

The transformation we have described above does produce a semantically correct slice $S$ of a model $M$, using the definition $=^{s,V}_{sem}$ of semantic equivalence, because, if $e$ is a valid history of $M$, ending in the slice target state $s$, and $V$ a set of variables of $M$, then:

- $e$ is also a valid history of $S$, ending in $\sigma(s)$, since the triggers of each transition in the models are the same: only their effects have been simplified, and their guards potentially weakened.
- The variables retained in $S$ are the union $V'$ of the sets $V_x$ of the variables upon which $V$ in $s$ depends, for each state $x$ of any path to $s$, and hence $V'$ contains $V_x$ for each state $x$ on the history $e$, and in particular for the initial state
- since the values of the variables 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 $e$ to $S$ and $M$, the values of $V$ in $s$ in $M$ at termination of the response to $e$ are the same as their values in $\sigma(s)$ in $S$.

# 4 State Machine Slicing

Both classes and operations can have their behaviour defined by state machines (termed *behavioural state machines* in Chapter 15 of [23]). For class behaviour, the state machine will have a set of input events, corresponding to the call events of the operations of the class.

Reactive systems will also have a state machine, consisting of the concurrent composition (product) of the state machines of their components, as described in the

previous section. If $M$ is the state machine $SM_P$ of a system $P$, then $\alpha M \subseteq P_I$ ($M$ can only respond to invocations of operations of $P$).

State machines may also generate output events, the set of output events for a state machine $M$ is denoted by $out(M)$. We assume that $\alpha M$ and $out(M)$ are disjoint, to avoid circularities in message processing. Objects of classes (including components in a reactive system) have a behaviour defined by the state machine of their class.

Operations may have their effect defined by a state machine, instead of a postcondition or activity. Such state machines do not have operation call triggers on their transitions, instead their transitions are triggered by completion of the transition source state. $\alpha M$ is therefore the empty set for such state machines. Operation state machines will also have a final state, representing termination of the operation whose effect they define.

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 (page 553 of [23]): *skip/ignore semantics*, *precondition/exception semantics* and *blocking semantics*:

1. Skip/ignore semantics: if a logical case is missing for the transitions triggered by an operation, leaving a particular state, then the operation is permitted to execute in this case, but has no effect on any data or the current state.

2. Precondition semantics: alternatively, an attempt to execute the operation in such a case may result in arbitrary behaviour.

3. Blocking semantics: execution of the operation in such a case is not permitted, the caller of the operation will be blocked until the guard of an explicit transition for the operation from the state becomes true, or until a state is reached where the operation can be accepted.

Skip semantics is used for general state machines and precondition semantics for protocol state machines in UML (page 581 of [23]). Blocking semantics can be used to protect a shared resource from incorrect use in a concurrent execution environment. The *defer* $e$ declaration in a state asserts that $e$ is blocked in the state if $e$ is invoked when no guard of an explicit transition from the state triggered by $e$ is true (page 569 of [23]).

A *skip-free* sequence of input events for a state machine with skip semantics is a sequence which causes no implicit skips to take place. $sf(M)$ is the set of skip-free input sequences in $\text{seq}(\alpha M)$.

In this section and Section 5.1 we will assume skip semantics for all state machines, with no occurrences of *defer*. We will also assume that the state machines are deterministic. Our transformations preserve determinism and completeness of state machines.

In Figure 4 the association member end properties *pathFrom*, *pathTo* and *dependencySet* are derived from the other features. For example, the algorithm to compute forward reachability, *pathTo* can be defined as:

```
for t : Transitions_M do
  t.source.pathTo := { t.target } ;
while any x.pathTo changes do
  for t : Transitions_M do
```

$$t.source.pathTo \;:=\; t.source.pathTo \;\cup\; t.target.pathTo$$

Slicing can be carried out for both class and operation 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 particular states of the machine, particularly the final state of the machine, if there is such a state. This is termed *data-based* slicing of state machines.

Figure 9 shows a simplified state machine for the Alaris GP infusion pump [4], which is a widely-used intravenous infusion device. We have added a state invariant $vtbi > 0$ to the *infusing* state, as this is a safety property which should be proved about this state. Likewise, we would need to check that an undesirable condition such as $rate = 0$ is not possible in this state (the infusion rate cannot be 0).

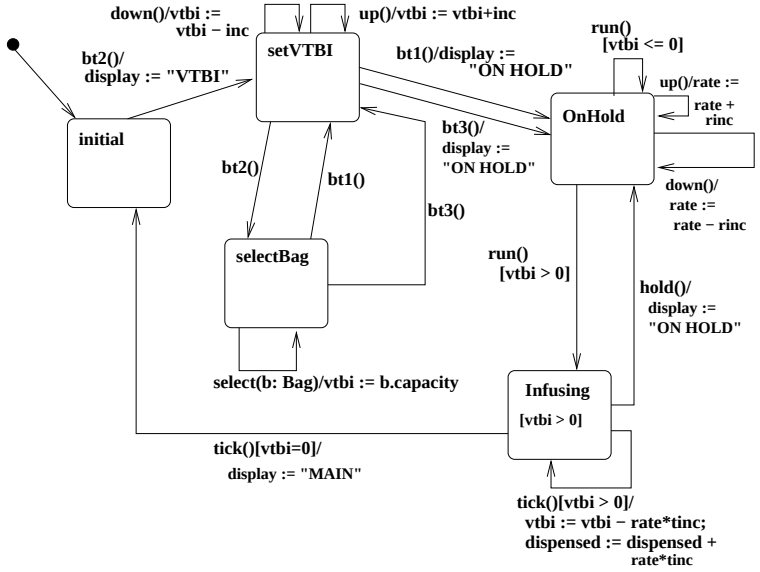



Figure 9 – Alaris infusion pump state machine

Slicing techniques for state machines based on control and data-flow analysis have been defined by Korel [11] and Clark [6]. These use graph-theoretic structural properties of the state machine. Our approach to data-based state machine slicing uses instead the semantic concept of *path-predicates*, as used in static analysis tools such as SPADE [26]. 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. These predicates have the form

$$Cond_1 \Rightarrow v' = f_1(v) \wedge ... \wedge Cond_n \Rightarrow v' = f_n(v)$$

for some exclusive conditions $Cond_i$ on the starting state variables $v$, and $v'$ represent the values of the variables at the end of the path. These identify a precise semantic relation between variables on each path, however their calculation is impractical for general state machines with loops.

An algorithm for computing path predicates for loop-free state machines is as follows (disregarding the preconditions and postconditions of operations). Assuming that $x.stateInvariant$ is initially *true* for all states, we set it to $v' = v$ for final states, where $v$ is the tuple of variables of $Variables_M$ of interest.

```
while any x.stateInvariant changes do
  for tr : Transitions_M do
    tr.source.stateInvariant :=
        conjunction(tr.source.stateInvariant,
          implication(tr.guard, wpc(tr.effect, tr.target.stateInvariant)))
```

where $wpc : Statement \times Expression \rightarrow Expression$ is the usual weakest-precondition operator (Appendix A). $conjunction(e1, e2)$ constructs the OCL expression that represents the conjunction of its arguments, etc. Algebraic simplification can be used at each step to reduce the complexity of expressions. Figure 10 shows an example of this derivation.

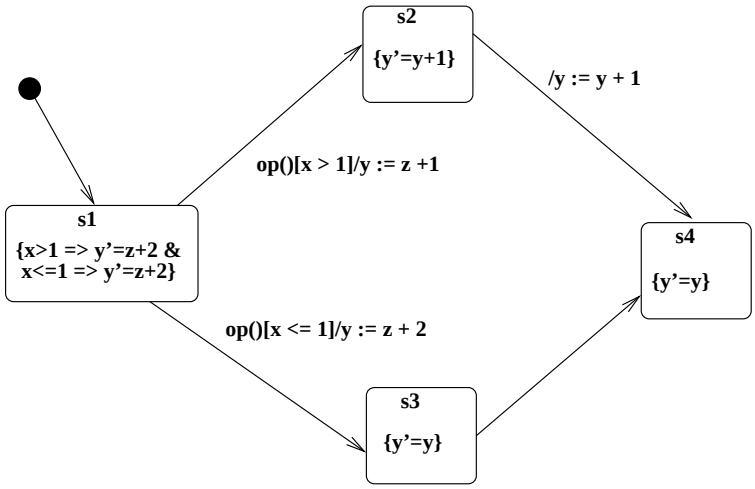



Figure 10 – Example of weakest precondition derivation

Alternatively, *path conditions* [29], which give information on the path-dependent data dependencies can be used.

As described in the previous section, we approximate path predicates and path conditions by assigning a set $V_x$ of variables to every state $x$, such that $V_x$ includes all variables which may affect a certain set $V$ of variables in a specific state $s$ or in all states. $V_x$ corresponds to $x.dependencySet$ in Figure 4. For loop-free state machines, $V_x$ contains $var(x.stateInvariant)$ as computed by the above algorithm.

The initial step of data-based slicing computes the sets $V_x$ by repeated iteration over the transitions of the state machine.

An alternative form of slicing a state machine $M$ is to compute its behaviour when its sets of input events or output events are restricted. The initial step of such *event-based* slicing [21] is to remove transitions triggered by the omitted input events, for input event slicing, and to remove invocations of the omitted output events from transition actions, for output event slicing.

Event-based and data-based slicing are inter-related, because data slicing of a client component in a reactive system may remove certain events from its output event set, so that its suppliers may be input event sliced on this reduced set of requested events (if they have no other clients that could send the events). For example, the input events $set1on$, $set1half$, $set2none$ can be removed for the $cb1$ and $cb2$ combinator components in Figure 8 since they are never invoked.

Conversely, data slicing of a supplier component may remove all transitions trig-

gered by certain events, so that its clients may be output event sliced on these events (if they have no other suppliers receiving the events).

For either data-based or event-based slicing, the following transformations can be subsequently applied to simplify the sliced state machines: (i) removing unreachable states and their incident transitions, (ii) slicing transitions to remove actions that cannot affect the values of $V_{trg}$ in the target state *trg* of the transition (for data-based slicing), (iii) replacing unmodified variables by their initial values, (iv) deleting transitions with false guards, (v) merging transitions with the same source, target, actions and trigger, (vi) R-merging states: combining pairs of states that have equivalent sets of outgoing transitions, (vii) G-merging states: combining groups of states with equivalent behaviour.

For data-based slicing, this sequence is followed by a recalculation of the $V_x$ sets and repetition of the simplification steps, until no further reduction in the state machine can be made[1].

The criteria for data-slicing a state machine $M$ are: $S <_{syn} M$ if $S$ has fewer elements (states, transitions, transition actions, etc) than $M$. Formally, if

$$Q(M) = s_M + t_M + v_M + a_M$$

is defined as a measure of the size of $M$, where $s_M$ is the number of states $\#States_M$ of $M$, $t_M$ is the number of transitions $\#Transitions_M$ in $M$, $a_M$ is the number of basic transition action statements and $v_M$ the number of variables $\#Variables_M$, then we require $Q(S) < Q(M)$. There should be an abstraction mapping

$$\sigma_M : States_M \to States_S$$

of states, such that $\sigma$ respects initial states:

$$\sigma(initial_M) = initial_S$$

and an abstraction mapping

$$\sigma_M : Transitions_M \to Transitions_S$$

of transitions, such that $\sigma$ respects triggers, sources and targets:

$$\sigma_M(tr).trigger = tr.trigger$$
$$\sigma_M(tr).source = \sigma_M(tr.source)$$
$$\sigma_M(tr).target = \sigma_M(tr.target)$$

for any $tr : Transitions_M$.

The $\sigma_M$ mappings are total functions on the reachable states and executable transitions of $M$: unreachable states and transitions of $M$ may be deleted, as may transitions with *false* guards, but these do not contribute to the semantics of $M$.

In addition, $\alpha S = \alpha M$ and $Variables_S \subseteq Variables_M$.

Syntactically, the slice is structure-preserving except for cases where elements of $M$ are deleted or merged.

We define three versions of semantic equivalence, in each case $V \subseteq Variables_S$ is assumed:

[1]The data dependencies need to be recalculated because the set of states and the set of state machine paths may have changed.

- $S =^{s,V}_{ssem} M$ if for all skip-free sequences $e$ of input events of $M$, starting from $S$ and $M$ in their initial states, with the same values for common variables in the initial states, the state $\sigma(s)$ is reached by $S$ as a result of the sequence $e$ whenever $s$ is reached by $M$ as a result of $e$, and then the values of the variables $V$ of interest are the same in the two models, as are the sequences $\underline{sent}$ of sent messages:

$$\begin{array}{l}\forall\, e : sf(M);\ \forall\, v_0, v;\ \forall\, sq : \mathrm{seq}(out(M)) \cdot \\ \quad initial_M[Variables_M = v_0] \longrightarrow^M_e s[V = v, \underline{sent} = sq] \ \Rightarrow \\ \quad\quad \sigma(initial_M)[Variables_S = v_1] \longrightarrow^S_e \sigma(s)[V = v, \underline{sent} = sq]\end{array}$$

  where $src[P] \longrightarrow^N_e trg[Q]$ denotes that there is a sequence of transitions in $N$ triggered in order by the event sequence $e$, starting from state $src$ with $P$ true, and ending with state $trg$ with $Q$ true in $trg$. $v_1$ are the initial values for the elements of $Variables_S$ defined by $Variables_M = v_0$.

  In addition, all skip-free input sequences of $M$ should be skip-free input sequences of $S$[2]:

$$sf(S) \subseteq sf(M)$$

  $=_{ssem}$ is reflexive and transitive (Appendix B).

- $S =^{s,V}_{sem} M$ is the same except that all input sequences $e : \mathrm{seq}(\alpha M)$ of events of $M$ are considered, and the second condition is not required. These definitions are therefore equivalent for complete state machines.

- $S =^{s,V=v}_{wsem} M$ if $s[V = v]$ is reachable in $M$ iff $\sigma(s)[V = v]$ is reachable in $S$:

$$\begin{array}{l}\exists\, e : \mathrm{seq}(\alpha M);\ v_0 \cdot initial_M[Variables_M = v_0] \longrightarrow^M_e s[V = v] \ \equiv \\ \exists\, e' : \mathrm{seq}(\alpha S);\ v'_0 \cdot \sigma(initial_M)[Variables_S = v'_0] \longrightarrow^S_{e'} \sigma(s)[V = v]\end{array}$$

$=_{sem}$ and $=_{ssem}$ essentially express that $S$ can simulate $M$'s behaviour on the variables $V$, for paths ending at $s$. A client of $S$ will be unable to distinguish $S$'s behaviour from that of $M$ if it can only observe the end-to-end transformation of data values ($v_0$ at the start to $v$ at the end).

The sequence of sent output events (invoked operations) is also preserved by this form of slicing, so again an observer will be unable to distinguish sequences of executions of $S$ and $M$ by their output behaviour.

The first semantic definition $=_{ssem}$ enables any analysis which concerns the value of the slice variables $V$ in the selected state $s$, over all skip-free paths to this state, to be performed on the slice $S$, and the result will then 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 $M$. (Each event sequence $e$ leading to $s$ in $M$ has a corresponding skip-free subsequence $e'$ leading to $s$, with identical functionality. $e'$ then also reaches $s$ in $S$.)

In the Alaris GP example, we can take $V$ as the set $\{vtbi\}$ of variables of the state invariant property that we wish to prove. Data-based slicing on this set simplifies the model, by eliminating variables such as *display*.

[2]In the case of state machines for operations, the empty sequence is the only case that needs to be considered for $e$.

The second definition $=_{sem}$ is more general, however it does not permit the use of the most powerful forms of state merging to reduce the size of state machines.

The third definition $=_{wsem}$ is relevant for hazard analysis of a system: if we want to identify how a potentially hazardous situation $V = v$ can occur in $M$, it is sufficient to apply the analysis to a $=_{wsem}$ slice $S$.

For example, in the infusion pump example, it should never be possible to have $rate = 0$ in the *infusing* state. Again, slicing using the set $V = \{rate\}$, according to $=_{wsem}$, substantially reduces the size of the model to be analysed.

Transformation (i) removes all states $x$ which cannot occur in paths from the initial state to the state $s$ of interest, together with their incoming and outgoing transitions.

The algorithm for this transformation is:

```
for x : States_M do
  if s ∉ x.pathTo
  then remove x and x.incoming and x.outgoing
  else if x ∉ initial_M.pathTo
  then remove x and x.incoming and x.outgoing
```

This transformation is semantically valid with respect to $=^s_{sem}$, $=^s_{ssem}$ and $=^s_{wsem}$ since no sequence of input events $e$ which reaches $s$ from the initial state can produce a path containing $x$ or its incident transitions. The time complexity of this process is of the order of $t_M * s_M$.

For data-based slicing, given a particular state $s$ in a state machine and a set $V$ of variables of interest in that state, we determine the data 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 variables such that: the value of the variables of $V_x$ in state $x$ may affect the value of a variable in $V$ in state $s$, but that no other variable in state $x$ can affect $V$ in state $s$.

Formally, to each state $x$ is assigned a set $V_x$ of variables, such that, 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 transition

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

the set $V_{s1}$ of variables of interest in $s1$ are augmented by all variables which appear in $Pre_{op}$ and $G$, and by all variables which may affect the value of $V_{s2}$ in $s2$ as a result of the class definition of $op(p)$, followed by *acts* (as described in Section 3). Dependencies for operation calls in *acts* are calculated from the definition of the called operation, as we describe in Section 5 below.

Variables of transition guards $G$ are not added to $V_{s1}$ if they cannot affect the values of variables in $V_{s2}$.

Figure 11 shows a simple example of this process, where *op* has *true* precondition and postcondition.

Here $x$ can be removed from the dependency set of $s1$.

The iteration is repeated until there is no change in any $V_x$ set. The algorithm can be defined as:

```
while any x.dependencySet changes do
  for tr : Transitions_M do
```

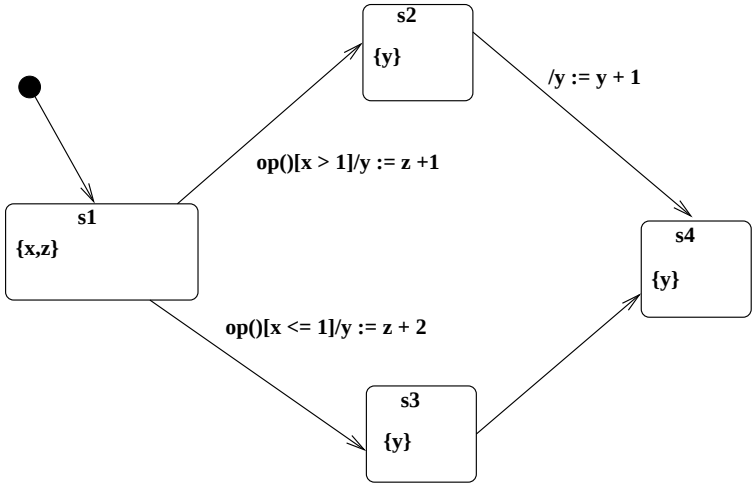



Figure 11 – Example of $V_x$ derivation

$$tr.source.dependencySet := tr.source.dependencySet \cup var(tr.guard) \cup \rho^{-1}_{op}(\!| dependents(tr.effect, tr.target.dependencySet) |\!)$$

$var(tr.guard)$ can be omitted if $tr$ is the only path from $s1 = tr.source$ to $s2 = tr.target$, $s1 \neq s2$:

$$\forall\, t : s2.incoming - \{tr\} \cdot s1 \notin t.source.pathFrom$$

Likewise for self-transitions on $x$ which do not update any variable of $V_x$.

There can be at most $s_M * v_M$ iterations of the *while* loop, so the worst case time complexity of this process is of the order of $t_M * s_M * v_M$.

Transformation (ii) uses the sets $V_x$ to slice individual transitions to remove 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. This step is close to the usual data-slicing of procedural program code to remove ineffective statements: $tr.effect$ is replaced by $slice(tr.effect, V_{tr.target})$ for each transition $tr$. In addition, $op$ can be data-sliced on $dependents(tr.effect, V_{tr.target})$, as described in Section 3.

This transformation is semantically valid wrt all three semantic equalities since only the values of the variables in $V_{s2}$ in $s2$ affect the values of $V$ in $s$. If a path for input sequence $e$ contains $tr$, then it has the same semantic effect on $V$ in $s$ as the path which replaces $tr$ by its sliced version where only the actions affecting $V_{s2}$ in $s2$ are retained. The time complexity of the transformation is linear in $t_M$, because the statements of the effect are basic statements.

An example of transition slicing is shown in Figure 12, the action $z := x + z$ can be deleted since it does not affect the value of $y$ in $s2$.

Transformation (iii) replaces a variable $v$ by a constant value $e$ throughout a state machine, if $v$ is initialised to $e$ on the initial transition $t_{init}$ of the state machine, and

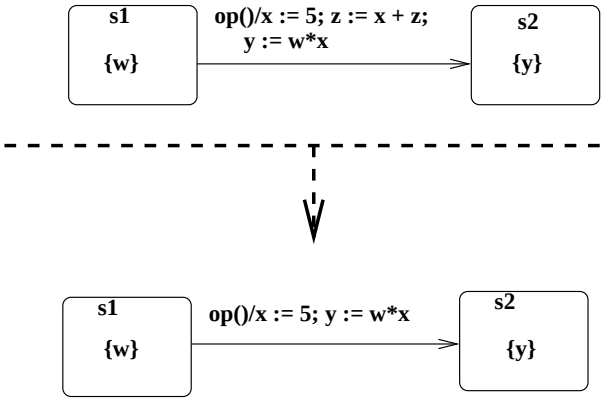



Figure 12 – Transition slicing example

is never subsequently modified. Expressions in guards and statements can then be simplified. Again this transformation is correct wrt the semantic relations because the original and transformed models have identical semantics. It is a well-known compiler optimisation strategy. The algorithm for this transformation is:

```
for v : Variables_M do
  if v ∈ t_init.effect.writes and
    v ∉ (Transitions_M − { t_init }).effect.writes
  then
    for t : Transitions_M − { t_init } do
      (t.guard := t.guard.substitute(v, e);
       t.effect := t.effect.substitute(v, e))
```

The worst case time complexity of this transformation is of the order of $v_M * (t_M + a_M)$ where $v_M$ is the number of variables of $M$ and $a_M$ is the number of basic transition action statements of $M$.

Transformation (iv) deletes all transitions with a *false* guard. Algebraic reduction is applied to simplify expressions such as $x < 0$ *and* $x \geq 0$ to *false*. Since such transitions cannot occur in any path, this transformation is semantically valid. The time complexity of this transformation is linear in $t_M$. This transformation is proved correct in [14].

Transformation (v) merges all pairs of transitions which have the same triggers, sources, targets and effects. The 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:

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

Again, algebraic simplification can be applied, to reduce expressions such as $x < 0$ *or* $x \geq 0$ to *true*, thus potentially reducing control dependencies.

The algorithm is as follows:

```
for x : States_M do
  for t1 : x.outgoing do
    for t2 : x.outgoing do
```

```
if t1 ≠ t2 and t1.target = t2.target and
   t1.trigger = t2.trigger and t1.effect = t2.effect
then
   (t1.guard := disjunction(t1.guard, t2.guard);
    delete t2)
```

This transformation is valid since the semantics of the original and transformed model are identical. It is of time complexity $s_M * t_M^2$.

The induced mapping of transitions is:

$$\sigma(tr) = \begin{array}{l} t1 \textit{ if } tr = t2 \\ tr \textit{ otherwise} \end{array}$$

if $t1$ and $t2$ are merged by the algorithm.

Transformation (vi) implements a generalised version of the R-merge algorithm of [10] for reduction of non-deterministic automata. It was also described, as 'Collecting transitions' in [14]. If two non-final states have equivalent sets of outgoing transitions then they can be merged. The incoming transitions of the resulting state are the union of the separate sets of incoming transitions. Conceptually, the states are being grouped together because they have identical behaviour.

The algorithm is:

```
for s1 : States_M do
  for s2 : States_M do
    if s1 ≠ s2 and s1 ∉ FinalState and s2 ∉ FinalState and
      sameOutgoing(s1, s2) and sameOutgoing(s2, s1)
    then
       for t : s2.incoming do t.target := s1;
       delete s2 and s2.outgoing
```

$sameOutgoing(s1, s2)$ returns *true* in the case that

$$\begin{array}{l} \forall\, t1 : s1.outgoing \cdot \exists\, t2 : s2.outgoing \cdot \\ \quad t2.trigger = t1.trigger \textit{ and } t2.effect = t1.effect \textit{ and} \\ \quad t2.guard = t1.guard \textit{ and} \\ \quad (t2.target = t1.target \textit{ or} \\ \qquad (t2.target = s1 \textit{ and } t1.target = s2) \textit{ or} \\ \qquad (t2.target = s2 \textit{ and } t1.target = s1)) \end{array}$$

The merged state is initial in the new model if either $s1$ or $s2$ were initial in $M$.

This transformation has time complexity bounded by $s_M * s_M * t_M$. It is semantically correct since any path which enters $s1$ or $s2$ in the original model as a result of input sequence $e$ enters $s1$ in the new model as a result of $e$, with the same values of variables. Paths within the set $\{s1, s2\}$ in the original model become paths with equivalent self-transitions on $s1$ in the new model. A transition exiting $s1$ or $s2$ and with a target distinct from either, say $s3$, has an equivalent transition from $s1$ to $s3$ in the new model.

The induced abstraction mapping $\sigma$ of states of $M$ to states of $S$ is:

$$\sigma(x) = \begin{array}{l} s1 \textit{ if } x = s2 \\ x \textit{ otherwise} \end{array}$$

Likewise, $\sigma(tr)$ of an outgoing transition of $s2$ in the original model is the corresponding outgoing transition of $s1$ in the transformed model. The state invariant of $s1$ is set to be the disjunction of the state invariants of $s1$ and $s2$.

Notice that the correctness of transformations (i) to (vi) does not depend on the assumption that the input sequence $e$ is skip-free. This assumption is needed for transformation (vii). In addition, the first six transformations are of polynomial time complexity in terms of the size of $M$.

Transformation (vii) merges a group $K$ of states into a single state $k$ if:

1. None of the states are final.
2. All transitions between the states of $K$ have no actions. These become self-transitions on $k$.
3. 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 be distinguished by disjoint guard conditions as transitions from $k$.
4. Each event $\alpha$ causing exit from $K$ cannot occur on states within $K$ which are not the explicit source of a transition triggered by $\alpha$.

The induced abstraction mapping $\sigma$ of states of $M$ to states of $S$ is:

$$\sigma(x) = \begin{array}{l} k \ \mathit{if}\ x \in K \\ x \ \mathit{otherwise} \end{array}$$

Similarly, the transitions $tr$ of $M$ have corresponding interpretations as transitions $\sigma(tr)$ of $S$. The state invariant of $k$ is set to be the disjunction of those of the states of $K$.

After this transformation is applied, the $V_x$ need to be recomputed, since the set of states has changed, and additional paths have been added to the state machine. The transformation is of exponential time complexity in the number of states of $M$. The algorithm for general state merging is as follows:

$$\begin{array}{l} \alpha_0 := \{\ t : \mathit{Transitions}_M \mid t.\mathit{effect}.\mathit{statements} = \mathit{Sequence}\{\}\ \}; \\ \mathit{sgroups} := \{\ \{\ t.\mathit{source},\ t.\mathit{target}\ \} \mid t \in \alpha_0\ \mathit{and}\ \mathit{validGroup}(\{\ t.\mathit{source},\ t.\mathit{target}\ \})\ \}; \\ \texttt{while}\ \mathit{sgroups}\ \mathit{changes}\ \texttt{do} \\ \quad \mathit{sgroups} := \mathit{sgroups} \cup \\ \qquad \{\ \mathit{sg1} \cup \mathit{sg2} \mid \mathit{sg1} \in \mathit{sgroups}\ \mathit{and}\ \mathit{sg2} \in \mathit{sgroups}\ \mathit{and} \\ \qquad\quad \mathit{sg1} \neq \mathit{sg2}\ \mathit{and}\ \mathit{sg1} \cap \mathit{sg2} \neq \{\}\ \mathit{and} \\ \qquad\quad \mathit{validGroup}(\mathit{sg1} \cup \mathit{sg2})\ \}; \\ \mathit{result} := \{\ \mathit{gg} \mid \mathit{gg} \in \mathit{sgroups}\ \mathit{and}\ \mathit{gg}\ \mathit{maximal}\ \}; \\ \mathit{mergeGroup}(\mathit{result} \rightarrow \mathit{any}()) \end{array}$$

*validGroup*($g$) applies the checks in items 1, 2, 3 and 4 above on $g$. The algorithm starts with basic groups composed of the end states of actionless transitions, then tries to merge these into larger groups until no more mergings are possible. The final step chooses a group that maximises the number of state mergings. This algorithm is an optimised version of that defined for G-merge in [1]: instead of considering all possible subsets of $\mathit{States}_M$, only the possible candidates for G-merging are considered.

This is considerably more efficient in the common case where only a few mergings are possible (the size of *sgroups* is under 10).

The transformation is valid with respect to $=^{s}_{ssem}$ because if $e$ is an input sequence with a path which traverses $K$ and ends with $s$, and has no implicit skips, then a corresponding path ending with $\sigma_M(s)$ in the transformed model can be constructed:

1. A transition to a state within $K$ from a state outside it becomes a transition with the same guard, source, trigger and actions, and with target $k$.

2. Explicit transitions of the path within $K$ have corresponding transitions in the new model, with identical trigger, guards and (skip) actions as self-transitions on $k$.

3. A transition $tr$ of the path which exits $K$ must either have a trigger distinct from the triggers of any of the internal transitions of $K$ (by condition 3 in the description of G-merging), or $tr$'s guard condition is disjoint from the guards of transitions triggered by $tr.trigger$ which are internal to $K$ (the generalisation of condition 3). In either case $tr$ is interpreted by a transition with source $k$ and unchanged trigger, guard, action and target in the new model.

The new path has identical functionality to the original path, and is a path for $e$ in the new model. The path ends with $s$ if $s \notin K$, otherwise it ends with $k$.

Figure 13 shows an example of this transformation.

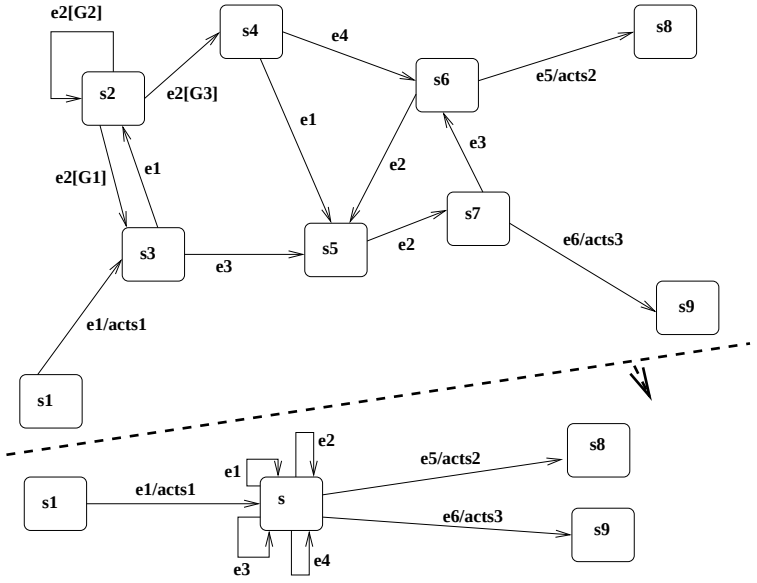



Figure 13 – Merging states transformation (1)

Skip-free input event sequences such as $e1$, $e3$, $e2$, $e6$ or $e1$, $e1$, $e2[G3]$, $e4$, $e2$, $e2$, $e3$, $e5$ have the same effect ($acts1$; $acts3$ and $acts1$; $acts2$ respectively) in the two models, and produce paths between the same start and end states ($s1$ to $s9$ and $s1$ to $s8$, respectively). This holds for any skip-free input event sequence of the original model.

Condition 3 can be generalised to allow events triggering internal transitions of $K$ to be the same as those triggering transitions that exit $K$, provided that the guards of the latter transitions are disjoint from any within $K$, for each such event. For such

events $e$, for each transition $t$ that exits $K$ and has $t.trigger = e$, there must be an equivalent transition (with the same trigger, target, guard and effect as $t$) from each state of $K$, and likewise the internal transitions triggered by $e$ must be duplicated on each state of $K$. Figure 14 shows an example of this case.

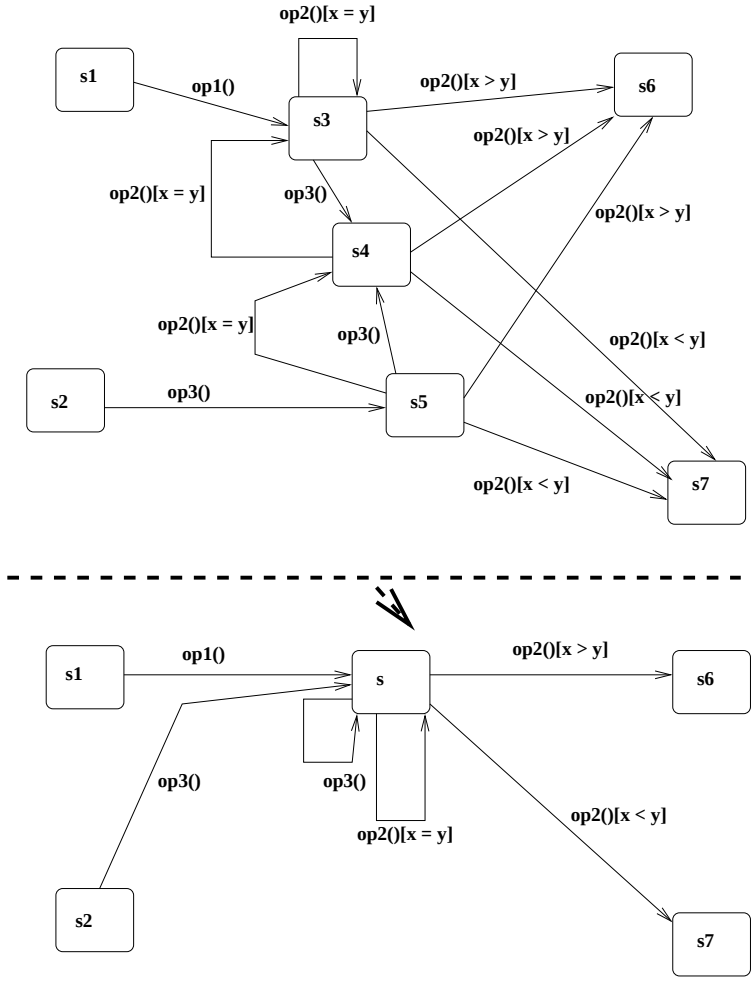



Figure 14 – Merging states transformation (2)

Finally, if (disjoint) state invariants are available for the source states of two transitions in $K$ with distinct source states, the same trigger, but different actions or target states but overlapping guards, the state invariants can be used to make the guards disjoint, to satisfy the generalised condition 3 (Figure 15).

The transformation is not valid for $=_{sem}$. Instead, all actionless self-transitions can be deleted: this step is valid only for $=_{sem}$.

The complete algorithm for data-based state machine slicing for $=_{ssem}$ is then:

```
while some reduction occurs in the model
do
 (compute reachability relations;
  remove unreachable states and transitions;
  compute dependency sets V_x;
```

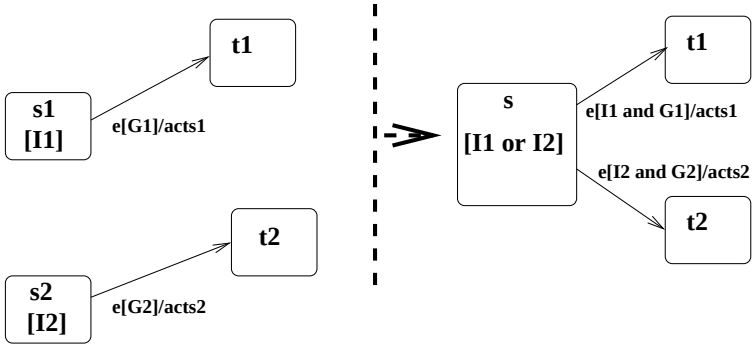



Figure 15 – Merging states transformation (3)

*data − slice transition actions*;
*replace unmodified variables by their values*;
*delete unexecutable transitions*;
*merge transitions*;
*R − merge states*;
*G − merge states*)

The order of transformations is chosen so that earlier steps may facilitate the application of later steps (for example, slicing transition actions and replacing variables may make explicit cases where two transitions have the same effect, enabling merging of the transitions). The most time-consuming steps are postponed to the end of each iteration.

The overall time complexity of the state machine slicing process is polynomial in the size of the source state machine $M$, if transformation (vii) is omitted, because the number of iterations of the transformation process is bounded by

$$Q(M) = v_M + s_M + t_M + a_M$$

since the iteration is only continued while $Q$ is decreased, and each transformation application reduces one of these quantities.

The techniques described in this section apply to any state machine in our subset of the UML state machine language. In particular, they can be applied directly to the product state machine $c1.SM_{C1} \times ... \times cm.SM_{Cm}$ of the state machines of components within a reactive system.

However, it is more practical to decompose the slicing of systems by individually slicing components of the system:

- Concurrent compositions of state machines, such as reactive system specifications, can be data-sliced by slicing each component state machine separately, provided there is no explicit communication between the machines.
- Client and supplier machines can be data-sliced separately to produce a data-slice of their composition.

In the first case, if state machine $M$ is composed of orthogonal regions $M_1$ and $M_2$, with disjoint sets of variables, $\alpha M_1$ is disjoint from $out(M_2)$ and $\alpha M_2$ is disjoint from $out(M_1)$, and there are no references to the states or data of $M_1$ from $M_2$, or

vice-versa, then if $M_1$ is sliced with respect to variables $V_1$ and state $s_1$ of $M_1$ to produce a slice $S_1$, this machine composed with $M_2$ forms a $V_1$, $V_2$ and $(s_1, s_2)$ slice for $M$, where $V_2$ are all features of $M_2$, and $s_2$ is any state of $M_2$.

An example of this situation is the composition of the state machines for *lf* : *LineFollower* and *ca* : *CollisionAvoidance* in the robot control system.

This holds for the general semantic equivalence $=^{s,V}_{sem}$. For strict equivalence $=_{ssem}$ the additional condition that $\alpha M_1$ and $\alpha M_2$ are disjoint is required.

In the second case, if state machine $M$ is composed of orthogonal regions $M_1$ and $M_2$, with disjoint sets of variables, $\alpha M_1$ is disjoint from $out(M_2)$ and $\alpha M_2$ is a subset of $out(M_1)$, and there are no references to the states or data of $M_1$ from $M_2$, or vice-versa, and $\alpha M_1$ and $\alpha M_2$ are disjoint, then if $M_1$ is sliced with respect to variables $V_1$ and state $s_1$ of $M_1$ to produce a slice $S_1$, and $M_2$ is sliced with respect to variables $V_2$ and state $s_2$ of $M_2$ to produce a slice $S_2$, $S_1$ composed with $S_2$ forms a $V_1$, $V_2$ and $(s_1, s_2)$ slice for $M$.

This holds for the general semantic equivalence $=^{s,V}_{sem}$ and for strict equivalence $=_{ssem}$.

The proofs are given in Appendix B.

Our notion of data and control dependency between states is simpler than the concepts of transition post-domination [27] used in graph-theoretic approaches to state machine slicing. Instead of computing a possibly very large data-and-control flow graph, our technique relies primarily on the determination of a set of variables for each state in the state machine.

# 5 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 $M2$, $M1$ is a client of $M2$) form an acyclic directed graph. The data of a state machine then also includes implicitly the data of all machines directly or indirectly subordinate to it in the client-supplier hierarchy (ie, the data of objects whose operations are invoked from the state machine).

## 5.1 Data-based slicing

For the lift system, we can separate out the door from the lift control class (Figure 16). In this version, the lift sends messages *door.atDest*(*fps* = *dest*) to the door when the door state needs to change because of lift events.

Figure 17 shows (on the left) the new version of the *arrive* operation state machine in this system, and the state machine of the invoked operation *atDest* (on the right).

The data-dependency calculation for each individual state machine is performed as in Section 4, except that input parameters of calls are linked to the possible features that could have been supplied as actual arguments to the call, in any call of the operation in a superordinate state machine. The data-dependency due to an invoked operation is calculated from the state machine of this operation, and the features used as actual parameters. This uses a similar technique as the calculation of inter-procedural data dependencies for the slicing of sequential programs [9].

In this example, the slice set $V = \{door.dm\}$ in the final state of the *arrive* operation state machine is used as the slice set in the final state of *atDest*, to calculate

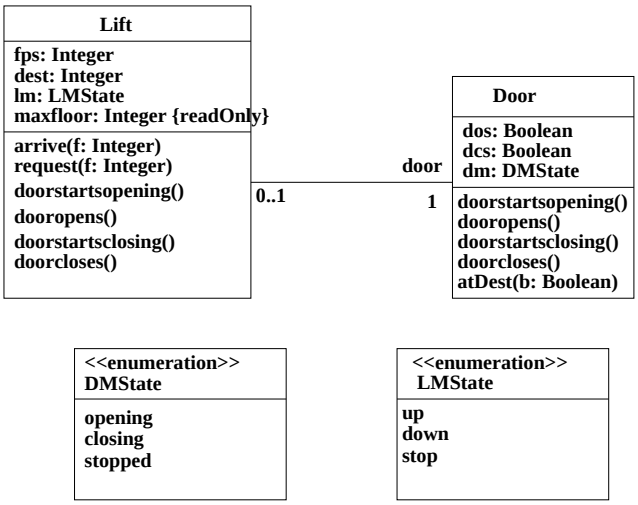



Figure 16 – Lift with subordinate Door

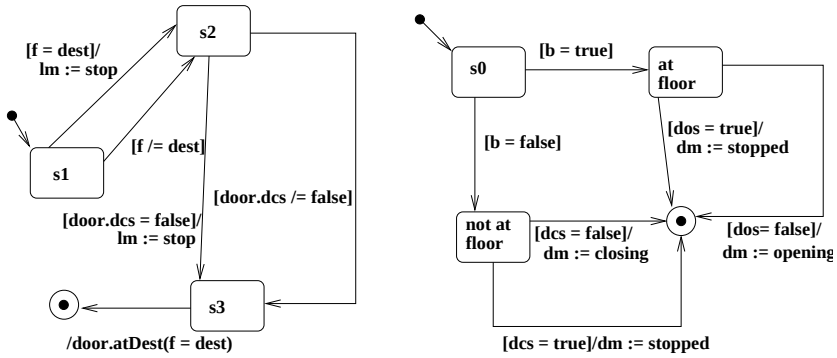



Figure 17 – *arrive* and *atDoor* state machines

the dependency of this invocation. This produces the set $\{b, door.dos, door.dcs\}$ at the initial state $s0$ of *atDest*. The only possible variables whose values can be used in the parameter $b$ are *dest* and $f$, so the actual dependencies from the call are $\{dest, f, door.dos, door.dcs\}$, and this is set as the dependency set in $s3$, it is also the dependency set at the start of *arrive*.

The general procedure for calculating $V' = dependents(c.op(e), V)$ for an invocation of a subordinate component operation $c.op(e)$ is as follows:

1. For each transition $tr$ of the state machine of $c$, with $tr.trigger = op$, compute

$$dependents(p := e, var(tr.guard) \cup \rho^{-1}_{c.op}(\!|\ dependents(tr.effect, V)\ |\!))$$

   where $p$ are the formal parameters of $op$.

2. If an implicit skip transition can occur for $c.op$ in $c$, include the set $V$.

3. Take $V'$ as the union of these sets.

The same concepts of syntactic $<$ and semantic relations $=$ can be used for composite state machines as for single state machines, but taken with respect to the full data of each state machine, including the data of subordinate (supplier) machines.

These structures of state machines are common in reactive systems, where the leaf components represent single devices (actuators) which are managed by supervisor controllers in a hierarchy: the top levels of the hierarchy may manage complete processes whilst the lower levels are responsible for steps within a process [25].

## 5.2 Event-based slicing

Slicing of an individual state machine $M$ within a hierarchy of communicating state machines can lead to further simplification of both subordinate (supplier) state machines and superordinate (client) state machines in the hierarchy, using event-based slicing.

For machines $M'$ directly subordinate to $M$, *input event slicing* can be used [21]. This simplifies a state machine by removing certain input events from its behaviour. The notation $M \upharpoonright E$ denotes $M$ restricted to a set $E$ of input events. If $M$ is the only client of $M'$, all events $e$ which are no longer generated by the slice $S$ of $M$ can be removed from the input alphabet of $M'$ to produce a slice $S'$. Transitions triggered by $e$ in $M'$ can be removed, and $M'$ simplified to $S'$ by using the transformations (i), (iii) to (vii) above. The same concepts of semantic equivalence are used, except that we restrict attention to those histories composed of sequences of events from $E$:

$$\begin{array}{l} \forall\, e : sf(M) \cap \mathrm{seq}(E);\ \forall\, v_0, v;\ \forall\, sq : \mathrm{seq}(out(M)) \cdot \\ \quad initial_M[Variables_M = v_0] \longrightarrow^M_e s[V = v, \underline{sent} = sq] \ \Rightarrow \\ \quad\quad \sigma(initial_M)[Variables_S = v_0'] \longrightarrow^S_e \sigma(s)[V = v, \underline{sent} = sq] \end{array}$$

for $=_{ssem}$ and similarly for $=_{sem}$ and $=_{wsem}$.

Likewise, if $M1$ has $M$ as its only supplier, and directly invokes operations of $M$, if the slice $S$ of $M$ no longer has any explicit transitions for certain input events $e$ of $M$, $M1$ can be *output event sliced*, by which invocations of $e$ are removed from the actions of transitions of $M1$. This could then enable transition merging or state merging transformations to simplify $M1$ to $S1$.

The semantic equality for output event slicing with respect to event set $E$ permits sent events to be omitted unless they are in $E$:

$$\begin{array}{l}\forall\, e : sf(M);\ \forall\, v_0, v;\ \forall\, sq : \mathrm{seq}(out(M)) \cdot \\ \quad initial_M[Variables_M = v_0] \longrightarrow_e^M s[V = v, \underline{sent} = sq] \ \Rightarrow \\ \quad\quad \sigma(initial_M)[Variables_S = v_0'] \longrightarrow_e^S \sigma(s)[V = v, \underline{sent} = sq \upharpoonright E]\end{array}$$

for $=_{ssem}$ and similarly for $=_{sem}$. $sq \upharpoonright E$ is the subsequence of $sq$ consisting of the events of $E$.

To calculate the restriction $MS \upharpoonright E$ of an acyclic hierarchy $MS$ of communicating state machines, we calculate $R \upharpoonright E$ for the machines $R$ without clients in $MS$, then for interior machines $Mx$, compute $Mx \upharpoonright \bigcup_{M \in clients(Mx)} out(M')$ where the $M'$ are the transformed forms of the clients $M$ of $Mx$ in $MS$.

More generally, input event slicing could be used when:

- We want to investigate the behaviour of the system when the input set of events is restricted to a subset (this could also include restricting the range of values of the parameters carried by these events).

  For example, in the robot control system it can be determined that the robot never turns left if input events for *lsright* are removed.

- We wish to show that the behaviour of the system on a subset of input events is identical to another system (such as an earlier version of the same system).

  For example, the infusion pump should behave in the same way as a previous version of the same device when used in the same manner, to avoid potentially hazardous mistakes by users familiar with the previous version. If the previous version had no bag selection facility *select* or operation $bt3$, input slicing on these operations reveals that the new version has inconsistent behaviour with the previous version for histories such as $bt2();\ bt2();\ bt1()$, and that states $setVTBI$ and *selectBag* cannot be merged.

- The state machine is to be reused in a new environment where certain events cannot occur.

  For example, removing unused events from the *Combinator* instances in the robot control system.

If a state machine is placed in an environment in which certain events (operations invoked on the state machine) cannot occur, we can simplify the state machine by omitting these events.

In the case that the values of input parameters are restricted, guards which use these values can be simplified by using these additional constraints. In some cases, guards may become simplified to *false*, so that the transition can be deleted.

The complete algorithm for input event state machine slicing is:

```
delete transitions whose trigger is not in E;
while some reduction occurs in the model
do
 (compute reachability relations;
  remove unreachable states and transitions;
  replace unmodified variables by their values;
  delete unexecutable transitions;
```

*merge transitions*;
*R − merge states*;
*G − merge states*)

The general motivations for output event slicing are:

- We want to inspect the state machine's modes and effect on one group of output devices, independently of its effect on certain others.
- We want to factor a large state machine into parallel controllers, which are each responsible for separate subsystems of a control system, to enable reuse and greater flexibility [28].

In particular, given a state machine $M$ which sends disjoint sets $E1$ and $E2$ of events to disjoint subordinate groups $M1$ and $M2$ of components, we can output slice $M$ on $E1$ to obtain a local controller $S1$ of $M1$, and output slice $M$ on $E2$ to obtain a local controller $S2$ of $M2$, then combine $S1$ and $S2$ in parallel to achieve the original effect of the monolithic controller $M$.

This is only appropriate for remodularisation if the groups of components in separate modules do not have required order relations on their actions relative to each other.

The complete algorithm for output event state machine slicing is:

*delete transition actions which are operation calls of excluded events*;
`while` *some reduction occurs in the model*
`do`
(*compute reachability relations*;
*remove unreachable states and transitions*;
*replace unmodified variables by their values*;
*delete unexecutable transitions*;
*merge transitions*;
*R − merge states*;
*G − merge states*)

As an example of output event slicing and refactoring, consider a control system for a simple gas burner, which has two sensors: a switch *sw*, flame detector *fd*, and three actuators: a gas valve *gv*, air valve *av* and ignitor *ig* (Figure 18).

In this example, the gas valve and air valve must be controlled by the same controller, because it is a safety requirement that the air valve is always on when the gas valve is on. The ignitor can be controlled by a separate state machine, however. Therefore we output slice on the actions of the valves, to produce one subcontroller (Figure 19), and on the actions of the ignitor to produce another subcontroller (Figure 20).

By using the third version of G-merging we can simplify these subcontrollers to produce the models of Figure 21.

## 5.3 Deadlock detection

Slicing can also be used to simplify state machines to facilitate the detection of potential deadlock situations. A potential deadlock occurs when the current state of the system has no available transitions for any event in the input event queue of the system. We extend the semantic model of state machines $M$ to include a queue

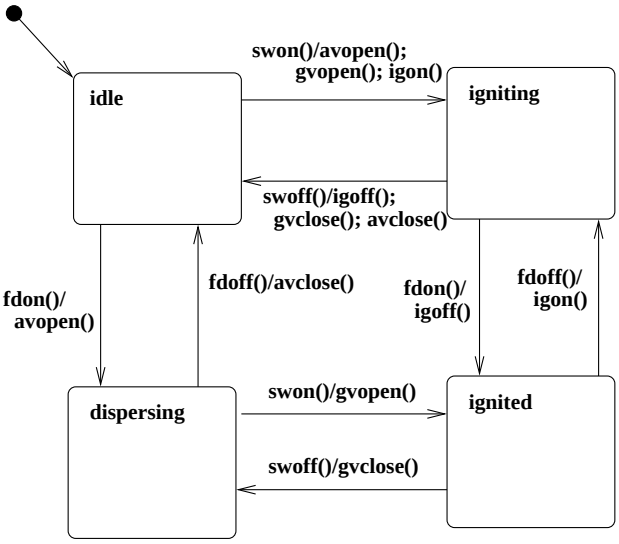



Figure 18 – Original gas burner state machine

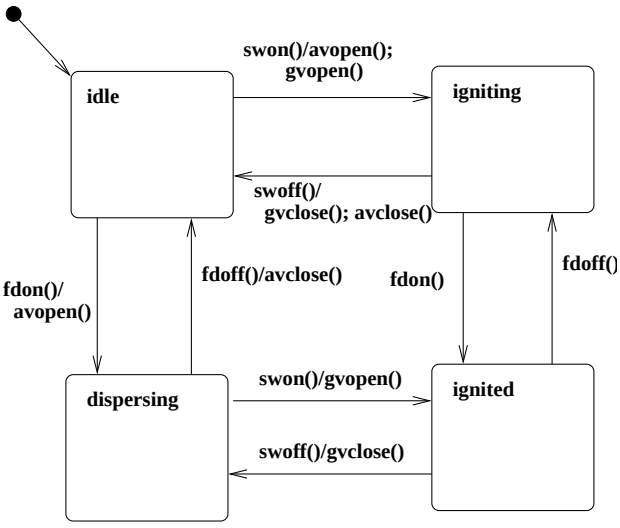



Figure 19 – Valves controller

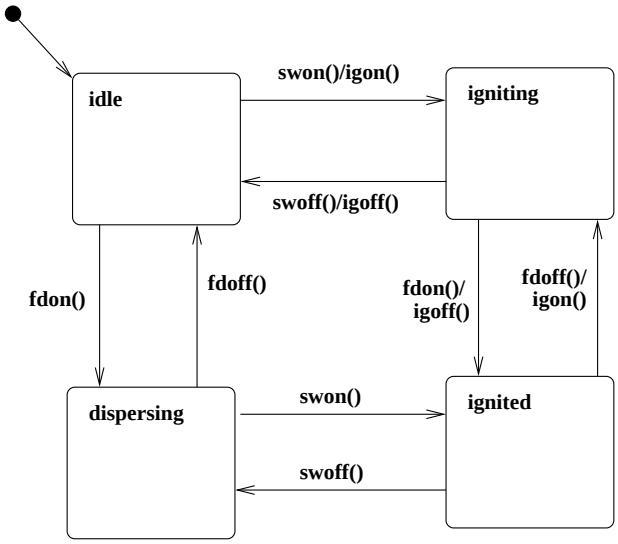



Figure 20 – Controller for ignitor

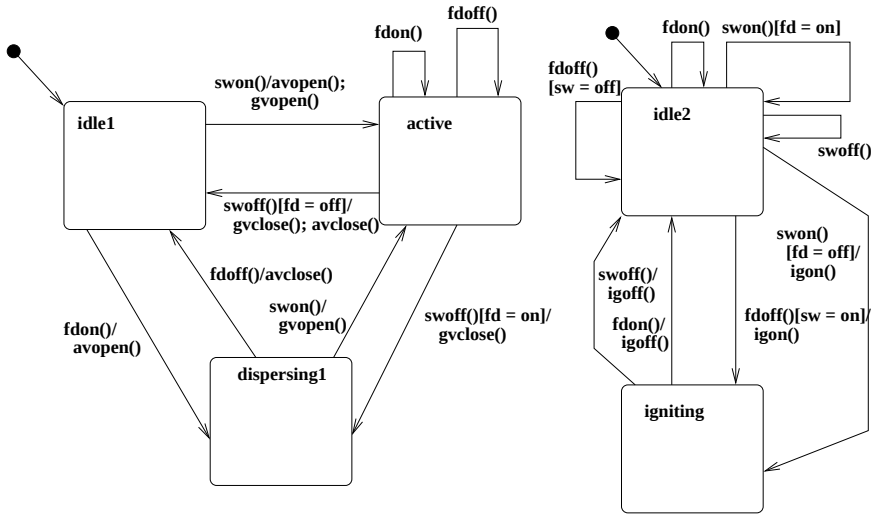



Figure 21 – Controllers after state merging

$waiting_M : Bag(\alpha M)$ of input events which have been received by $M$ but not yet accepted (referred to as the event pool in [23], page 569). No order is assumed for the events in the input event queue, however a specific policy such as 'first come, first served' may be used, which enforces that already-received events must be accepted before later arrivals.

States may list events as deferred, meaning that the events cannot be accepted in the state. $s.deferrableTrigger$ is the set of deferred events of state $s$.

A potential deadlock arises when $waiting_M$ is non-empty, and

$$waiting_M \subseteq s.deferrableTrigger$$

for the currently active state $s$ of $M$.

An appropriate slicing definition for simplifying the analysis of potential deadlocks is: $S <_{dsyn} M$ is $S <_{syn} M$ together with the condition that

$$\forall\, st : States_M \cdot st.deferrableTrigger = \sigma_M(st).deferrableTrigger$$

$S =^{s}_{dsem} M$ is $S =^{s,\{\}}_{sem} M$, together with the condition that $waiting_M$ at state $s$ equals $waiting_S$ at state $\sigma_M(s)$.

We can therefore deduce that, for potential deadlock detection:

1. Only states with *deferrableTrigger* non-empty need to be considered for $s$.
2. Transformations (i), (iii), (iv) and (v), deleting unreachable states, deleting transitions with false guards, transition merging and replacing variables by constants preserve potential deadlocks.
3. Only variables that affect which state is active need to be considered: that is, we can take the initial slice set $V_x$ (for data-based slicing) for each state $x$ equal to the set of variables in the guards of its outgoing transitions.
4. R-merging of two states is valid provided the additional condition that they have the same *deferrableTrigger* sets is true.
5. Non-blocking leaf components $L$ in a reactive system can be removed (that is, components whose states all have empty *deferrableTrigger* sets), provided their data and states are not referred to in their clients. Clients of $L$ can then be output event sliced on $\alpha L$.

We can deduce that: A state $s$ of $M$ has a potential deadlock in $M$ iff $\sigma(s)$ has a potential deadlock in a slice $S$ of $M$ computed using these transformations.

We illustrate these transformations on the classic 'dining philosophers' example. Figure 22 shows the class diagram of philosopher, fork and bell components, and Figure 23 their state machines.

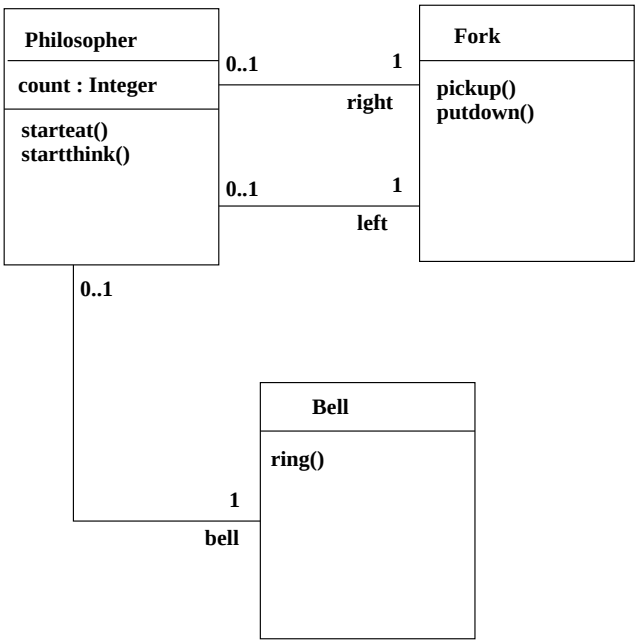



Figure 22 – Dining philosophers system

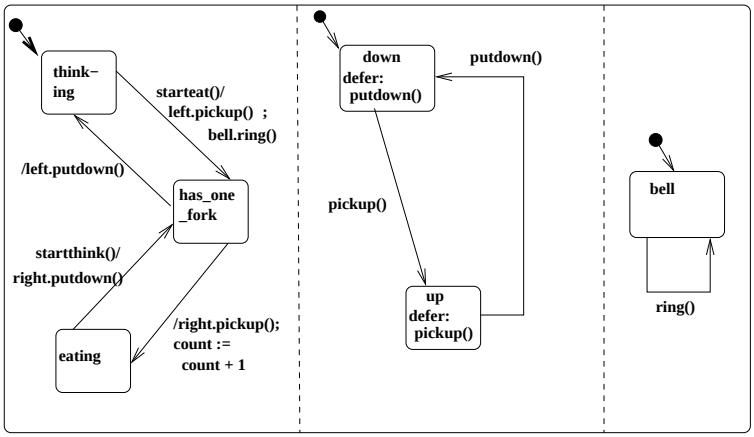



Figure 23 – Dining philosophers state machines

Consider a simple configuration with two instances of each class:

$b1 : Bell$
$b2 : Bell$
$f1 : Fork$
$f2 : Fork$
$p1 : Philosopher$
$p2 : Philosopher$
$p1.bell = b1$
$p2.bell = b2$
$p1.left = f1$
$p1.right = f2$
$p2.left = f2$
$p2.right = f1$

An input event sequence

$p1.starteat(), p2.starteat()$

from the initial system state can lead to the potential deadlock state where both philosophers are in the *has_one_fork* state, and both forks are in the *up* state with *pickup*() in their event queues.

Identification of this system state can be equivalently carried out on the sliced version of the system. The local variable *count* of *Philosopher* can be sliced away, as can the non-blocking leaf component *Bell* and its events.

## 6 Evaluation

State machine slicing for state machines of objects and operations has been implemented in the UML-RSDS tools [17, 22], using the algorithms defined in Sections 3 and 4.

To test the efficiency of the algorithms presented here, we carried out the data and event-based slicing of a concurrent composition $M$ of multiple copies of a component $N$ with three states (Figure 24 shows the data slice of this machine for $V = \{\}$, and two copies of $N$).

The state space of $M$ therefore has $3^n$ states, for $n$ copies. There are $n * \#Transitions_N * (\#States_N)^{n-1}$ transitions in $M$, ie, $n * 3^n$ transitions. This is a 'worst case' test of data-slicing, because almost all states can be G-merged, if only one variable is retained in the slice: the resulting sliced state machine should have two states as in Figure 24.

Table 1 shows the size of these test cases, and execution time for data and input event slicing. There are approximately $2^{s_M}$ sets of states which are connected only by actionless transitions, and are therefore considered by the G-merge algorithm. However, this number can be bounded by merging sets of states only up to some limited size at each cycle of the algorithm: iteration of merging eventually achieves the same effect as attempting to identify and merge a large group of states in the original state machine.

These tests were executed on a Windows XP laptop with Pentium 4M 2GHz processor, and 256MB RAM. The final test case produced an out-of-memory error for data slicing. Test cases 4 and 5 are larger than any considered in [1], and the efficiency of our G-merge algorithm is higher.

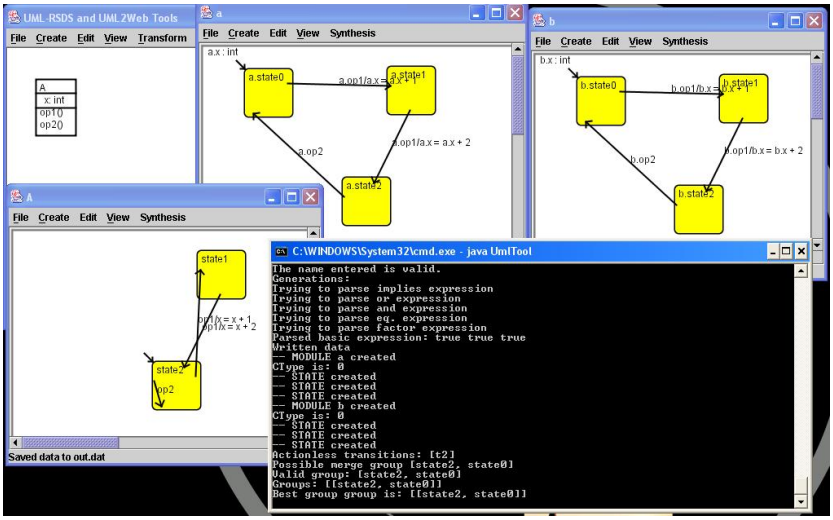



Figure 24 – Test case state machine

| *Test case* | *States* | *Transitions* | *Data slicing (ms)* | *Event slicing (ms)* |
|---|---|---|---|---|
| 1 | 3 | 3 | 0 | 0 |
| 2 | 9 | 18 | 20 | 20 |
| 3 | 27 | 81 | 3619 | 30 |
| 4 | 81 | 324 | 67,348 | 151 |
| 5 | 243 | 1215 | – | 359 |

Table 1 – Test cases

Data-slicing the flattened state machine $M$ on $a.x$ for a particular component $a$ reduces $M$ to the two-state data slice of $a.N$. Likewise, input event slicing to retain the events $a.op1$, $a.op2$ also reduces $M$ to the slice of $a.N$. In this respect, slicing is an inverse to flattening, and can extract the components of a flattened product of state machines from the product.

Further examples of event-based slicing have been carried out on industrial applications, with substantial reductions in model sizes [1].

# 7 Related Work

Transformations for simplifying UML state machines and class diagrams are defined in [13, 14], and transformations to simplify activity diagrams described in [7]. These are not based upon a reduction of the set of variables or system events, but rely on semantic equivalence to restructure systems without such reduction: slicing techniques are more powerful than such transformations in principle since a weaker condition of semantic equivalence is required.

Similar concepts for event-based slicing of hierarchically organised reactive systems are described in [5], although individual components are not simplified by this process, only their interfaces are reduced.

In contrast to the work of [30, 12] on state machine slicing, we do not require the sliced system of state machines to be equivalent to the original for general temporal logic properties, but only with regard to the values of features in a particular state. Thus substantial reduction techniques such as G-merging can be used, and our state machine slices may be potentially much simpler than the original state machines.

In [19] two additional state machine reduction transformations are described: (i) path contraction and (ii) AND-factoring. Path contraction is applicable if a state has a single incoming and outgoing transition, it can then be removed and the transitions combined, if the outgoing transition has a completion trigger (Figure 25).

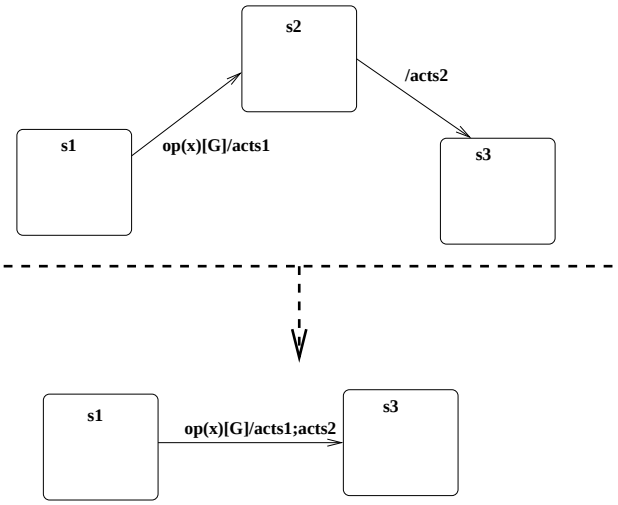



Figure 25 – Path contraction

This may be generalised to the case of multiple outgoing transitions from the second state.

In principle, both of these transformations could also be used in our state machine slicing process, since they preserve the semantic relations whilst reducing syntactic complexity.

## 8 Summary

We have defined systematic techniques for the slicing of UML class diagram and state machine models of reactive systems. These techniques support the reduction of models to syntactically smaller but semantically related models, either on the basis of a set of features of interest, or (for state machines) on the basis of a subset of input or output events of interest. Properties of the original model can be deduced from those of the sliced model. In addition, models can be factored on the basis of groups of features or events. Extension of this work to activity diagrams and sequence diagrams is planned.

## References

[1] K. Androutsopoulos, D. Binkley, D. Clark, M. Harman, K. Lano et al, *Model projection: simplifying state-based models in response to restricting the environment*, ICSE 2011, doi: `http://doi.acm.org/10.1145/1985793.1985834`.

[2] R. A. Brooks, *Intelligence without representation*, Artificial Intelligence (47), pp. 139–159, 1991.

[3] I. Bruckner, H. Wehrheim, *Slicing Object-Z Specifications for Verification*, ZB 2005, LNCS 3455, Springer-Verlag, pp. 414–433, 2005.

[4] Cardinal Health Inc., *Alaris gp volumetric pump*, technical report, Cardinal Health, 1180 Rolle, Switzerland, 2006.

[5] S. Cheung, J. Kramer, *Context constraints for compositional reachability analysis*, ACM Transactions on Software Engineering and Methodology, 5(4), October 1996.

[6] D. Clark, *Amorphous Slicing for EFSMs*, PLID' 07, 2007.

[7] R. Eshuis, R. Wieringa, *Tool support for verifying UML activity diagrams*, IEEE Transactions on Software Engineering, 30 (7): pp. 437–447, 2004.

[8] M. Harman, D. Binkley, S. Danicic, *Amorphous Program Slicing*, Journal of Systems and Software, 68 (1): 45 – 69, October 2003, doi: `http://dx.doi.org/10.1016/S0164-1212(02)00135-8`.

[9] S. Horwitz, T. Reps, D. Binkley, *Interprocedural Slicing using Dependence Graphs*, Trans on Prog. Lang. and Syst., 12 (1): 26–60, 1990.

[10] 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.

[11] 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.

[12] S. Langenhove, A. Hoogewijs, *$SV_tL$: System verification through logic tool support for verifying sliced hierarchical statecharts*, LNCS, *Recent Trends in Algebraic Development Techniques*, pp. 142–155.

[13] K. Lano, J. Bicarregui, *Semantics and Transformations for UML Models*, UML 98, Mulhouse, France, June 1998, Springer-Verlag LNCS vol. 1618, 1998, pp. 107–119.

[14] K. Lano, J. Bicarregui, *UML Refinement and Abstraction Transformations*, ROOM 2 workshop, Bradford University, May 1998.

[15] K. Lano, *Logical Specification of Reactive and Real-Time Systems*, Journal of Logic and Computation, vol. 8, no. 5, pp. 679–711, 1998, doi: `http://dx.doi.org/10.1093/logcom/8.5.679`.

[16] K. Lano, D. Clark, K. Androutsopoulos, *UML To B: Formal Verification of Object-oriented Models*, IFM 2004, Springer-Verlag LNCS vol. 2999, pp. 187–206, doi: http://dx.doi.org/10.1007/978-3-540-24756-2_11.

[17] K. Lano, *Constraint-Driven Development*, Information and Software Technology, 50, 2008, pp. 406–423, doi: http://dx.doi.org/10.1016/j.infsof.2007.04.003.

[18] K. Lano, *A Compositional Semantics of UML-RSDS*, SoSyM, vol. 8, no. 1, February 2009, pp. 85–116, doi: http://dx.doi.org/10.1007/s10270-007-0064-x.

[19] K. Lano, *Slicing of UML state machines*, AIC '09 Proceedings, pp. 63–69, Moscow, August 2009.

[20] K. Lano (ed.), *UML 2 Semantics and Applications*, Wiley, 400 pages, 2009.

[21] K. Lano, *Event slicing of communicating state machines*, Dept. of Computer Science, King's College London, October 2009.

[22] K. Lano, S. Kolahdouz-Rahimi, *Specification and Verification of Model Transformations using UML-RSDS*, IFM 2010, doi: http://dx.doi.org/10.1007/978-3-642-16265-7_15.

[23] OMG, *UML superstructure, version 2.3. OMG document formal/2010-05-05*, 2009.

[24] OMG, *Model-Driven Architecture*, http://www.omg.org/mda/, 2004.

[25] F. J. Ortiz, D. Alonso, B. Alvarez, J. A. Pastor, *A Reference Control Architecture for Service Robots Implemented on a Climbing Vehicle*, Ada Europe 2005, Springer LNCS vol. 3555, pp. 13–24.

[26] Praxis Ltd., *The SPADE Program Analyser*, 2008.

[27] 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, doi: http://doi.acm.org/10.1145/1275497.1275502.

[28] 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.

[29] G. Snelting, T. Robschink, J. Krinke, *Efficient path conditions in dependence graphs for software safety analysis*, ACM Transactions on Software Engineering and Methodology, vol. 15, no. 4, October 2006, pp. 410–457, doi: http://doi.acm.org/10.1145/1178625.1178628.

[30] J. Wang, W. Dong, Z.-C. Qi, *Slicing hierarchical automata for model checking UML statecharts*, ICFEM, 2002.

[31] M. Weiser, *Program slicing*, IEEE Transactions on Soft. Eng., 10, July 1984, pp. 352–357.

[32] F. Wu, T. Yi, *Slicing Z Specifications*, ACM Sigplan, vol. 39 (8), August 2004.

## About the authors

**Kevin Lano** is the Reader in Software Engineering at King's College London. He is the author of over 100 conference and journal papers, and several books. His research covers the fields of formal specification, semantics of software modelling languages, model transformation and model verification.

**Shekoufeh Kolahdouz-Rahimi** is a PhD student in the Department of Informatics at King's College London. Her research interests are in the comparison and analysis of model transformation languages and approaches.

**Acknowledgments** The work carried out in this paper was supported by the SLIM EPSRC project. In particular, David Clark, Kelly Androutsopoulos, David Binkley and Mark Harman have assisted us in the refinement of the state machine slicing concepts presented here.

# A Definition of slicing and data dependency for statements

Write and read frames for statements can be defined as follows:

$$wr(x := e) = \{x\}$$
$$rd(x := e) = var(e)$$

$$wr(S1;\ S2) = wr(S1) \cup wr(S2)$$
$$rd(S1;\ S2) = rd(S1) \cup rd(S2)$$

$$wr(if\ E\ then\ S1\ else\ S2) = wr(S1) \cup wr(S2)$$
$$rd(if\ E\ then\ S1\ else\ S2) = var(E) \cup rd(S1) \cup rd(S2)$$

The function

$$dependents : Statement \times \mathbb{F}(Variable) \to \mathbb{F}(Variable)$$

gives the set $V'$ of variables whose values at the start of an execution of the statement may affect the values of the supplied variables at the end of the execution.

$$dependents(x := e,\ V) = \begin{cases} V & if\ x \notin V \\ rd(e) \cup (V - \{x\}) & otherwise \end{cases}$$

$$dependents(S1;\ S2, V) = dependents(S1, dependents(S2, V))$$

$$dependents(if\ E\ then\ S1\ else\ S2, V) = rd(E) \cup dependents(S1, V) \cup dependents(S2, V)$$

$$dependents(c.op(e), V) = \bigcup_{tr:Transitions_N \wedge tr.trigger = op} dependents(p := e, (var(tr.guard) \cup \rho^{-1}_{c.op}(\!|\ dependents(tr.effect, V)\ |\!)))$$

where $N$ is the state machine which supplies $c.op$ to the caller, and $p$ are the formal parameters of $op$. If implicit skip transitions of $N$ can occur in response to $c.op$, then $V$ is also added to the result set.

The function

$$slice : Statement \times \mathbb{F}(Variable) \to Statement$$

computes the slice of the statement with respect to the variables $V$.

$$slice(x := e,\ V) = \begin{cases} skip & if\ x \notin V \\ x := e & otherwise \end{cases}$$

$$slice(S1;\ S2, V) = slice(S1, dependents(S2, V));\ slice(S2, V)$$

$$slice(\mathit{if}\ E\ \mathit{then}\ S1\ \mathit{else}\ S2, V) = \\ slice(S1, V)\ \ \mathit{if}\ slice(S1, V) = slice(S2, V) \\ \mathit{if}\ E\ \mathit{then}\ slice(S1, V)\ \mathit{else}\ slice(S2, V)\ \ \mathit{otherwise}$$

Weakest preconditions are defined as follows:

$$wpc(x := e,\ P) = P[e/x]$$

$$wpc(S1;\ S2, P) = wpc(S1, wpc(S2, P))$$

$$wpc(\mathit{if}\ E\ \mathit{then}\ S1\ \mathit{else}\ S2, P) = \\ (E \Rightarrow wpc(S1, P)) \wedge \\ (\neg E \Rightarrow wpc(S2, P))$$

Notice that $dependents(Stat, V) = var(wpc(Stat, V = v'))$ where the $v'$ are considered as constants, for statements without operation calls.

# B Proofs of theorems

## B.1 Transitivity of semantic equivalence

$<_{syn}$ is transitive because if $R <_{syn} S$ and $S <_{syn} M$, then

$$Q(R) < Q(S) \wedge Q(S) < Q(M)$$

so $Q(R) < Q(M)$.

The abstraction mappings

$$\tau_M : States_M \to States_R$$

of states and

$$\tau_M : Transitions_M \to Transitions_R$$

of transitions are defined as the compositions of the abstraction mappings $\sigma_M$ from $M$ to $S$ and $\sigma_S$ from $S$ to $R$.

In addition, $\alpha R = \alpha M$ and $Variables_R \subseteq Variables_M$.

Therefore $R <_{syn} M$.

For $\overset{s,V}{sem}$, assume that $S =^{s,V}_{sem} M$ and $R =^{\sigma_M(s),V}_{sem} S$, where $V \subseteq Variables_R$.

Then:

$$\forall\, e : \mathrm{seq}(\alpha M);\ \forall\, v_0, v \cdot \\ initial_M[Variables_M = v_0] \longrightarrow^M_e s[V = v]\ \Rightarrow\ \sigma_M(initial_M)[Variables_S = v_1] \longrightarrow^S_e \sigma_M(s)[V = v]$$

where $v_1$ are the initial values for $Variables_S$ defined in $v_0$, and

$$\forall\, e' : \mathrm{seq}(\alpha S);\ \forall\, w_0, w \cdot \\ initial_S[Variables_S = w_0] \longrightarrow^S_{e'} s'[V = w]\ \Rightarrow\ \sigma_S(initial_S)[Variables_R = w_1] \longrightarrow^R_{e'} \sigma_S(s')[V = w]$$

where $s' = \sigma_M(s)$.

Then if $e \in \mathrm{seq}(\alpha M)$, and

$$initial_M[Variables_M = v_0] \longrightarrow^M_e s[V = v]$$

take $w = v$, $w_0 = v_1$ in the second definition, and $e' = e$, then:

$$initial_S[Variables_S = v_1] \longrightarrow^S_e \sigma_M(s)[V = v]$$

since $\sigma_M(initial_M) = initial_S$, so

$$\sigma_S(initial_S)[Variables_R = w_1] \longrightarrow^R_e \tau_M(s)[V = v]$$

as required. Likewise if sent messages *sent* are also considered.

## B.2 Slice factorisation

Let $M$ be a state machine with two orthogonal regions $M_1$ and $M_2$, with disjoint sets of variables, and with no reference between these regions to the state or data of the other region.

Select any state $s_2$ of $M_2$.

If $S_1$ is an $M_1$ slice for state $s_1$ of $M_1$ and set $V_1$ of variables of $M_1$, with respect to $=_{sem}$, then any input event sequence $e$ of events of $M$ with

$$(initial_{M_1}, initial_{M_2})[Variables_M = v_0] \longrightarrow_e^M (s_1, s_2)[V = v]$$

also has

$$initial_{M_1}[Variables_{M_1} = w_0] \longrightarrow_{e\restriction \alpha M_1}^{M_1} s_1[V_1 = v_1]$$

in $M_1$, where $V = V_1, V_2$ and $V_2$ is the set of all variables of $M_2$, so

$$\sigma_{M_1}(initial_{M_1})[Variables_{S_1} = w_0'] \longrightarrow_{e\restriction \alpha M_1}^{S_1} \sigma(s_1)[V_1 = v_1]$$

since $S_1$ is a $=_{sem}^{s_1, V_1}$ slice of $M_1$. But then

$$(\sigma_{M_1}(initial_{M_1}), initial_{M_2}) \longrightarrow_e^S (\sigma(s_1), s_2)[V = v]$$

because the events of $\alpha M_2 - \alpha M_1$ in $e$ have no effect upon the state or data of $M_1$. Therefore $S_1$ composed with $M_2$ is a $=_{sem}$ slice of $M$.

This proof needs modification for $=_{ssem}$ slices, because if an event is in both $\alpha M_1$ and $\alpha M_2$ then $e \restriction \alpha M_1$ may not be skip-free in $M_1$, even if $e$ is skip-free for $M$.

The additional hypothesis that $\alpha M_1$ and $\alpha M_2$ are disjoint is therefore necessary for $=_{ssem}$.

For client-supplier factorisation, assume that $S_1 =_{sem}^{s_1, V_1} M_1$ and $S_2 =_{sem}^{s_2, V_2} M_2$:

$$\begin{array}{l} \forall\, e : \text{seq}(\alpha M_1);\ \forall\, v_0, v;\ \forall\, sq : \text{seq}(out(M_1)) \cdot \\ \quad initial_{M_1}[Variables_{M_1} = v_0] \longrightarrow_e^{M_1} s_1[V_1 = v, \underline{sent} = sq] \ \Rightarrow \\ \quad\quad \sigma_1(initial_{M_1})[Variables_{S_1} = v_1] \longrightarrow_e^{S_1} \sigma_1(s_1)[V_1 = v, \underline{sent} = sq] \end{array}$$

and

$$\begin{array}{l} \forall\, e' : \text{seq}(\alpha M_2);\ \forall\, w_0, w;\ \forall\, r : \text{seq}(out(M_2)) \cdot \\ \quad initial_{M_2}[Variables_{M_2} = w_0] \longrightarrow_{e'}^{M_2} s_2[V_2 = w, \underline{sent} = r] \ \Rightarrow \\ \quad\quad \sigma_2(initial_{M_2})[Variables_{S_2} = w_1] \longrightarrow_{e'}^{S_2} \sigma_2(s_2)[V_2 = w, \underline{sent} = r] \end{array}$$

If we take $e' = sq \restriction \alpha M_2$ in the second inference, we can conclude that

$$\begin{array}{l} \forall\, e : \text{seq}(\alpha M);\ \forall\, u_0, u \cdot \\ \quad (initial_{M_1}, initial_{M_2})[Variables_M = u_0] \longrightarrow_e^M (s_1, s_2)[(V_1, V_2) = u, \underline{sent} = r] \ \Rightarrow \\ \quad\quad (\sigma_1(initial_{M_1}), \sigma_2(initial_{M_2}))[Variables_S = u_1] \longrightarrow_e^S (\sigma_1(s_1), \sigma_2(s_2))[(V_1, V_2) = u, \underline{sent} = r] \end{array}$$

for $M$ and $S$, as required, where $u_0 = (v_0, w_0)$, $u_1 = (v_1, w_1)$, $u = (v, w)$.

This holds since $M_2$ receives input events only from $M_1$, and $\alpha M = \alpha M_1,\ \alpha M_2 \subseteq out(M_1)$.

# C Summary of UML and OCL notations used

UML-RSDS is a model-based development and analysis method for software systems. It uses the following UML models:

- Use case models

- Simplified class diagram models
- State machine models
- OCL constraints
- Structured activities.

Communication between objects and between state machines is point-to-point from a specific client to a specific supplier, not broadcast.

The semantics of UML-RSDS is expressed in real-time action logic [15]. For each UML-RSDS model $M$ we can define (i) a logical language $\mathcal{L}_M$ that corresponds to $M$, and (ii) a logical theory $\Gamma_M$ in $\mathcal{L}_M$, which defines the semantic meaning of $M$, including any internal constraints of $M$.

The first-order language $\mathcal{L}_M$ consists of type symbols for each type defined in $M$, including primitive types such as integers, reals, booleans and strings which are normally included in models, and semantic types $\mathsf{C}$ for each classifier $C$ defined in $M$. There are attribute symbols $\mathsf{att(c : C)}$: $\mathsf{T}$ for each property $att$ of type $T$ in the feature set of a classifier $C$. There are attributes $\overline{\mathsf{C}}$ to denote the set of currently existing instances of each classifier $C$, ie, its extent. This corresponds to $C.allInstances()$ in OCL. There are action symbols $\mathsf{op(c : C, p : P)}$ for each operation $op(p : P)$ in the features of $C$ [18]. Collection types (sets, ordered sets, sequences, bags) and operations on these and the primitive types are also included. The OCL logical operators *and*, *or*, *implies*, *not* are semantically represented by $\wedge$, $\vee$, $\Rightarrow$, $\neg$ . $\cup$ denotes union of two collections, returning a collection of the same kind. $\{x : X \mid \varphi(x)\}$ denotes the subcollection of collection $X$ consisting of the elements that satisfy $\varphi(x)$.

The theory $\Gamma_M$ includes axioms expressing the multiplicities of association ends, the mutual inverse property of opposite association ends, deletion propagation through composite aggregations, the existence of generalisation relations, and the logical semantics of any explicit constraints in $M$, including pre/post specifications of operations. For example, if classifier $C$ generalises classifier $D$, this is expressed by the axiom $\overline{\mathsf{D}} \subseteq \overline{\mathsf{C}}$.

For a sentence $\varphi$ in $\mathcal{L}_M$, there is the usual notion of logical consequence:

$$\Gamma_M \vdash \varphi$$

means the sentence is provable from the theory of $M$, and so holds in $M$.

The theory $\Gamma_P$ of a particular configuration $P$ of a reactive system includes constants $\mathsf{c}$: $\overline{\mathsf{C}}$ for each component instance $c$ of class $C$, together with semantic representations of its attributes (ie., the variables of $P$) and operations.