Slicing of UML State Machines

KEVIN LANO
CREST
Department of Computer Science
King’s College London
Strand, London, UK
kevin.lano@kcl.ac.uk

Abstract: This paper defines techniques for the slicing of UML state machine models, to produce smaller or simpler models which have the same behaviour as the original model on selected data. Slicing enables more effective analysis and comprehension than the complete model, and can also be used to factor a model.

Key–Words: Slicing, UML State Machines

1 Introduction

Slicing of programs has been a widely-used analysis technique for many years [12] 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 traces back through the program, 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 become significant as a means of detecting flaws and in restructuring these models.

Slicing in the formulation of Harman and Danicic [2] is close to the concept of a quality improvement model transformation: a slice is considered as a transformed version \( S \) of an artifact \( C \) which has a lower value of some complexity measure, but an equivalent semantics with respect to the sliced data:

\[
S \prec_{\text{syn}} C \land S =_{\text{sem}} C
\]

The slicing may be structure-preserving, so that \( S \) has the same structure as \( C \) although containing only a subset of its elements, or may be amorphous, with a possibly completely different structure.

We will consider techniques for the structure-preserving and amorphous slicing of state machines. Model transformations which preserve \( =_{\text{sem}} \) and \( \prec_{\text{syn}} \) will be used to compute slices. In general, applying slicing at a high level of abstraction simplifies the calculation of the slice, and means it is possible to detect specification flaws at an early development stage, thus reducing development costs. By showing that slicing is preserved by certain kinds of refinement (the slice of a refinement is equal to the refinement of a slice), we can ensure that specification-level slicing is adequate to also analyse properties of the refinements of the specification.

2 State Machine Slicing

Slicing techniques for state machines based on control and data-flow analysis have been defined by Korel [3] and Clark [1]. These are generalisations of concepts from program slicing, to deal with the special features of state machines: that there is not necessarily an end state or other termination point of the behaviour represented by the state machine, and that control structures may be arbitrarily complex, with multiple entry/exit loops and other unstructured control flows.

Our approach to state machine slicing is based instead upon the notion of path-predicates, as used in static analysis tools such as SPADE [8]. 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.

The criteria for slicing state machines \( M \) are then: \( S \prec_{\text{syn}} M \) if \( S \) has fewer elements (states, transitions, transition actions, etc) than \( M \). \( S =_{\text{sem}} M \) if for all possible input sequences of events, starting from \( S \) and \( M \) in their initial states, and for all possible initial values of attributes in those states, the state \( s \) of interest is reached by \( S \) as a result of the input sequence whenever it is reached by \( M \) as a result of the same sequence, and then the value of the variables \( V \) of interest in the state \( s \) of interest are the same in the two
models.

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

Care must be taken concerning the state machine notation considered, and the semantics adopted, since the computation of the slice will differ from version to version. UML defines two sublanguages of state machine notation:

- Protocol state machines: describe the allowed or intended histories of objects of a class. Their transitions have postconditions instead of actions, and their states may have invariants.

- Behaviour state machines: describe behaviour of objects or operations. Their transitions have actions, their states do not have invariants.

We will consider behaviour state machines for objects and operations, but permit state invariants, since these are useful for reasoning about algorithm correctness, eg, by weakest-preconditions [10].

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 [7]:

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.

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.

We will assume skip semantics (which is normally assumed for behaviour state machines), but identify where blocking or precondition semantics would make a difference to the slicing definitions.

Figure 1 shows an example state machine of an ATM system, based on an example from [3]. The state PINError can be removed from the slice for state Accessing Current Account and variable cb, since no path exists from it to Accessing Current Account.

Given a particular state s in a state machine and a set V of variables of interest in that state, the path predicates for each path that terminates in state s can, in principle, be derived by an inductive computation over the set of transitions in the state machine. We initialise the algorithm with the predicates P being v' = v in s, the conjunction of each equality v_i' = v_i for v_i in V, and true in each other state. v_i' is a new variable denoting the value of v_i in s.

For each transition

\[ tr : s_1 \rightarrow_{op(x)} G/acts s_2 \]

the predicate P_{s1} is strengthened by the additional conjunct

\[ G \Rightarrow wpc(op(x); acts, P_{s_2}) \]

where wpc is the weakest-precondition operator.

Figure 2 shows a simple example of this calculation.

In the case that there are loops in the control flow, the path predicate calculation does not necessarily converge. If state invariants exist for loop nodes, these can be conjoined to the P_i predicates, and this may assist in achieving convergence.

An alternative way of determining the slice is to compute 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 $s$. $V_x$ will be a superset of the variables of $P_x$.

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. Given a transition

$$tr : s1 \rightarrow_{\text{op}(p)} G1/\text{acts} s2$$

the set $V_{s1}$ of variables of interest in $s1$ are augmented by all variables which appear in $Pre_{\text{op}}$ and $G$, and by all variables which may affect the value of $V_{s2}$ in $s2$ as a result of the code of $\text{op}(p)$ followed by $\text{acts}$. This iteration is repeated until there is no change in any $V_x$ set.

These dependencies can be simplified, and the $V_x$ sets made smaller, by omitting variables of the guards in cases where the values of variables in $V_x$ always depend on the $V_x$ in the same way, regardless of the paths taken from $x$ to $s$ because of different guards.

In terms of path conditions, if we have a condition of the form

$$G \Rightarrow P$$

resulting from the paths starting from $x$ with guard $G$ true, and condition

$$\text{not}(G) \Rightarrow P$$

for the paths starting with $G$ false, then the overall condition is simply $P$, independent of $G$.

In particular, guard variables (of $G3$) can be omitted from the data dependency set of $s1$ in situations such as those shown in Figure 3: if there is only one outgoing transition from $s1$ which leads to $s$, and if all self-transitions of $s1$ have no actions.

In the case of skip semantics, omitting the actionless self-transitions has no effect on the meaning of the model: the models have the same set of paths. In the case of blocking semantics the set of paths is reduced by removing the self-transitions, however every path in the new model which leaves the state by means of the retained transition (possibly after being blocked waiting for its condition to become true) could be considered equivalent to any path in the original model which busy-waits by repeatedly taking the self transitions.

Using the sets $V_x$, individual transitions can now be sliced to remove actions which cannot contribute to the values of the variables $V$ in state $s$. For a transition

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

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

An example of transition slicing is shown in Figure 4, the action $z := x + z$ can be deleted since it does not affect the value of $y$ in $s2$.

![Figure 4: Transition slicing example](image)

Figure 5 shows the ATM example after transition slicing:

![Figure 5: ATM after transition slicing](image)

Further transformations can then be applied to simplify the model: transition merging and state merging.
Transitions can be merged if their sources, targets and actions are the same:

\[ tr_1 : s_1 \xrightarrow{\text{op}(x)[G_1]/\text{acts}_1} s_2 \]

and

\[ tr_2 : s_1 \xrightarrow{\text{op}(x)[G_2]/\text{acts}_2} s_2 \]

can be replaced by:

\[ tr : s_1 \xrightarrow{\text{op}(x)[G_1 \text{ or } G_2]/\text{acts}_1} s_2 \]

Another case of transition merging is 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 6).

Another transformation to remove states is shown in Figure 7: if one state \( s_2 \) is auxiliary to another \( s_1 \), with the only incoming and outgoing transitions of \( s_2 \) being between itself or between itself and \( s_1 \), and being actionless, then we can merge \( s_2 \) into \( s_1 \), removing the new self-transitions which are derived from the original transitions between the states and on \( s_2 \).

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

States can be merged if their incoming and outgoing transitions are identical – modulo a bisimulation on the source and target states.

More precisely, let \( \sim \) be a relation between states such that \( s_1 \sim s_2 \) implies that \( V_{s_1} = V_{s_2} \), and:

\[
s_1.\text{outgoing}\rightarrow\forall (tr_1 | \text{s2.outgoing}\rightarrow\exists (tr_2 | \text{tr1.target} \sim \text{tr2.target} \text{ and } \text{tr1.trigger} = \text{tr2.trigger} \text{ and } \text{tr1.guard} = \text{tr2.guard} \text{ and } \text{tr1.effect} = \text{tr2.effect}))
\]

and

\[
s_1.\text{incoming}\rightarrow\forall (tr_1 | \text{s2.incoming}\rightarrow\exists (tr_2 | \text{tr1.source} \sim \text{tr2.source} \text{ and } \text{tr1.trigger} = \text{tr2.trigger} \text{ and } \text{tr1.guard} = \text{tr2.guard} \text{ and } \text{tr1.effect} = \text{tr2.effect}))
\]

and conversely. Then the states which are in the same equivalence groups under \( \sim \) can be merged. Their new invariant is the disjunction of the invariants of the individual states.

Figure 8 shows the ATM example after state merging of AccessingAccount and AccessingSaverAccount.

Notice that additional paths may be introduced into the slice \( S \) which were not present in the original model \( M \): by merging states it is now possible for paths to include sequences of events which were not possible in the original model, thus increasing the cost of model-checking [6, 11], although the reduction in state space size should outweigh this cost in most cases. The primary application of the form of amorphous slicing described here is to assist the compre-
hension of a state machine by restricting its behaviour to a part of the data which it operates on.

3 Preservation of State Machine Slicing by Refinement

As with class diagrams, it is possible to synthesise a PIM or PSM level state machine description from an abstract CIM specification consisting of type (class and enumeration) declarations and constraints [4].

In the CIM there should be one system or controller class, from which all other classes are reachable by navigation, and the multiplicity of association ends should always be of bounded maximum cardinality. Sensor features should be attributes.

Constraints may be operational, or non-operational. Operational constraints are of the form

\[ L \text{ implies } R \]

where \( R \) is a formula such as \( f = e \), \( f \rightarrow \text{includes}(e) \), \( f \rightarrow \text{excludes}(e) \), \( f \rightarrow \text{includesAll}(e) \), \( f \rightarrow \text{excludesAll}(e) \). \( f \) is a feature name, possibly with a selector \(-\text{at}(g)\) in the first case, for some expression \( g \), in the case of an ordered role \( f \). \( f \) is called the writable feature of the constraint. The data dependencies of each constraint \( \psi \) are \( x \mapsto f \) where \( x \) occurs in \( \psi \) and \( f \) is its writable feature. The data dependency relation \( \rho \) of the CIM is calculated from the operational constraints, as the transitive closure of the dependencies \( x \mapsto f \) of each constraint.

We assume that the set \( Vbls_{C} \) of data features of a CIM specification \( C \) are partitioned into three sets \( Sen_{C} \) of sensor (input) features, \( Internal_{C} \) of internal features, and \( Act_{C} \) of actuator (output) features. Each class may have a mixture of these features. Sensor features cannot occur as the writable feature \( f \) of an operational constraint. The types of sensor attributes must be finite sets.

An example, of a simple reactive control system, is shown in Figure 9. In this system \( v1 \) is an inlet valve for fluid entering a tank, \( v2 \) is an outlet valve, and \( s1 \) and \( s2 \) low and high-level sensors detecting the fluid level. \( s1 = 1 \) means that \( s1 \) detects fluid at its level, and \( v1 = 1 \) means that \( v1 \) is open.

The state machine \( SM_{C} \) synthesised for a set \( Cons \) of operational constraints for a CIM \( C \) has state space the cartesian product of the sensor attribute types, with one copy of a type included for the maximum number of objects with the attribute which may exist. There are transitions for each change in an individual sensor attribute value on one object between these states. For each state \( s \) in \( SM_{C} \), corresponding to the tuple \((v_{1},...,v_{n})\) of values for sensor attributes \((s_{1},...,s_{n})\), the invariant

\[ s_{1} = v_{1} \text{ and } ... \text{ and } s_{n} = v_{n} \]

holds. In addition, for each \( \psi \in Cons \),

\[ \psi[v_{1}/s_{1},...,v_{n}/s_{n}] \]

is an invariant of \( s \). From these latter constraints, the entry action of \( s \) can be determined, and hence the actions on every transition which is targeted at \( s \).

Figure 10 shows the synthesised state machine for the valve controller CIM.

If \( C \) is sliced on variables \( V \subseteq Act_{C} \cup Internal_{C} \) to produce a new specification \( S \), the operational constraints of \( S \) are all those operational constraints

\[ L \text{ implies } R \]

of \( C \) which have writable feature in \( \rho^{-1}[\{ V \}] \cup V \).
The refinement $SM_S$ of $S$ may therefore have reduced state space compared to that of $C$, since only the set $Sen_1$ of those sensors which occur in at least one constraint of $S$ need to be considered.

For example, the slice for the valve controller on $v1$ only has two states, for the possible values of $s1$, since $s2$ has been removed from the slice.

If the refinement $SM_C$ of $C$ is also sliced on $V$, the individual state invariants $Inv_x$ do not depend on $Sen_C$, and their data dependency relation $\sigma_x$ will be $\rho$ restricted to $Vbisc_C - Sen_C$, so that

$$\sigma_x^{-1}(V) = \rho^{-1}(V) - Sen_C$$

A state invariant operational constraint $\psi[v1/s1, ..., v_n/s_n]$ will only be included in the slice if its writable feature is in $\sigma_x^{-1}(V) \cup V$, and therefore also in $\rho^{-1}(V) \cup V$ since sensor attributes cannot be writable features. Thus state invariants in the refinement of $S$ and the slice of $SM_C$ are identical.

States of $SM_C$ which differ only in sensor values for sensor attributes not in $\rho^{-1}(V)$ will have identical state invariants in the slice of $SM_C$, and hence can be merged. The result is the state space of $SM_S$, showing that refinement and slicing can be equivalently performed in either order.

4 Slicing by refactoring

Model transformations can be used to simplify the structure of a state machine, and to factor out groups of states and their associated data. The two most relevant transformations are:

- Introduce OR-state: group together states which have the same behaviour as substates of a new OR state.
- Introduce AND-state: factor out a repeated structure of states into a region of an AND state.

The first transformation is described in detail in [5].

Introduction of an AND state can be used if a state machine $M$ can be partitioned into disjoint sets $G_1, ..., G_n$ of states, such that:

1. The $G_i$ all have the same size.
2. They are isomorphic with respect to the transitions within each $G_i$. That is, for $i, j : 1..n$, $i \neq j$, for each transition $t : s \to_{op_j}[\mathcal{E}] / \text{acts} s'$ within $G_i$, there is a corresponding transition $t' : s \to_{op_j}[\mathcal{E}] / \text{acts} s'$ within $G_j$.
3. Transitions between states in different $G_i$ groups are between isomorphic states.

$M$ can then be refactored into two regions of an AND state, one region $G$ is a copy of $G_1$, the other, $R$, is formed from $M$ by replacing each $G_i$ by a new state $s_i$, the transitions of $R$ are those of $M$ with the internal transitions of the $G_i$ removed, and additional guards introduced where necessary to express that a transition is only enabled in a particular state of $G$.

The initial state of $G$ is that state which is initial in $M$, likewise the initial state of $R$ is the $s_i$ corresponding to the $G_i$ in which the initial state of $M$ occurs.

Figure 11 shows this transformation.

In cases where $R$ does not depend on $G$ (no references to states of $G$ are made in the guards of $R$’s transitions), $G$ can be removed from the model to yield $R$ as a valid slice of $M$, where only states and data used in $R$ are of interest.

5 Verification

The following forms of state machine verification [6] are potentially facilitated by slicing:

- Proof of pre/post condition relationships between a state and the selected state of the slice: since every path in the slice $S$ is equivalent in its functionality on the slice variables to a path in the original model $M$, if a pre-post relationship can be established in the slice then also it can be established in the original model.
- Preservation of state invariants: if it can be proved that a predicate of a state $x$, which uses only features from $V_x$, is a state invariant of $x$ in
the slice, then it remains a state invariant of \( x \) in the original model.

Some forms of verification cannot be carried out on the slice, however, due to the enlargement in the set of possible paths:

- Proof of absence of deadlocks: absence of deadlocks in the slice does not establish the same property for the original model, since operations may execute in more states and under weaker conditions in the slice.
- Liveness analysis: a state may be visited infinitely often on all infinite paths in the slice without this being true for the original model, again due to the weakening of transition guard conditions, and deletion of states.
- Reaching an invalid state: this may not be possible in the slice, due to deletion of states, but may be possible in the original model.

In order that these forms of analysis could be carried out usefully on the slice, a stricter definition of semantic equivalence \( S =_{\text{sem}} M \) is necessary. In particular, that each path of \( M \) is equivalent to some path in \( S \), in addition to the converse property. State merging can only be carried out in this case for path contraction, where the second transition has no guard, otherwise guards of transitions must be preserved. For liveness, states which cannot reach the selected state should not be deleted, as these may be states which cause a failure of liveness (that the selected state occurs infinitely often on each infinite path).

Our slicing technique is as effective as the Korel approach in reducing the size and complexity of a state machine when slicing. Our notion of data and control dependency between states is simpler than the concepts of transition post-domination [9] used by Korel, and more appropriate to reactive systems, where there may be no termination states, and where there may be no control dependencies in the Korel sense: since there may be paths from each transition to any other. Our definition is non-termination insensitive, since the data-dependency calculation does not take account of cases where an infinite loop of states may arise, preventing the target state from being reached. Instead of computing a possibly very large data-and-control flow graph, our technique relies only on the determination of a set of variables for each state in the state machine.

Another advantage of our approach is that we avoid the introduction of non-determinism: the sliced state machines are deterministic if the original state machines are. We explicitly base our work upon UML state machines and the skip semantics for these state machines.

State machine slicing for behaviour state machines of objects and operations has been implemented in the UML2Web tool [4].

6 Summary

We have defined techniques for slicing of UML state machine models. These enable the models to be simplified and factored on the basis of groups of features. Extension of this work to composite states in state machines, and to activity diagrams and sequence diagrams is ongoing.

References: