SLICING OF UML MODELS

K. Lano and S. Kolahdouz-Rahimi
Department of Computer Science, King's College London, London, U.K.

Keywords: UML, Model transformations, Software verification.

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

1 INTRODUCTION

In this paper we show how slicing techniques for specification languages such as Z (Wu and Yi, 2004) and Object-Z (Bruckner and Wehrheim, 2005) can be combined with the semantic slicing of (Lano, 2009a) to slice UML class diagrams and state machines.

A slice can be considered to be a transformed version **S** of an artifact **M** which has a lower value of some complexity measure, but an equivalent semantics with respect to the sliced data (Harman et al., 2003):

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

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.

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

2 SLICING OF CLASS DIAGRAMS

We assume that the client-supplier relation between different classes in a class diagram forms a tree structure. The class at the root of this tree is termed the *controller* class of the system: it will usually serve as the access point to the services of the system for external users. Operations are assumed to be deterministic.

The *data features* of such a system are all the attributes of classes in the system, including association ends owned by the classes.

Slicing will be carried out upon class invariants and operation pre and postconditions by considering the *predicates* **P** of which they are composed. A predicate is a truth-valued formula, whose main operator is not **and**. Class invariant predicates can be either *assertions*: properties which are expected to be invariant for objects of the class, but which do not contribute to the effect of operation postconditions, or *effective*: implicitly conjoined to the postcondition of each update operation.

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

L implies R

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

Other predicates are termed *non-operational*, they may (in the case of postcondition predicates) consist of a mixture of features in pre form and post form, all

Lano K. and Kolahdouz-Rahimi S. (2010).
SLICING OF UML MODELS.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 259-262
DOI: 10.5220/0002913202590262

the features in post form are assumed to be writable in this case, and to be mutually data-dependent.

For any system there are also implicit operational constraints which relate the two ends of binary associations. Assertion constraints arise from the multiplicity restrictions of association ends, ie, a 0..n multiplicity end **br** has a constraint $\mathbf{br}{\rightarrow}\mathbf{size}() \leq \mathbf{n}$.

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

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

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

The first step in producing a slice of a class **C** is to normalise the class invariant, and each pre and post condition, so that these are all in the form of conjunctions of predicates. Then the effective invariant predicates are copied to the postconditions of each update operation.

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

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

  $$\mathbf{dep}(\mathbf{p}) = \mathbf{rd}(\mathbf{p}) \times \mathbf{wr}(\mathbf{p})$$

  and for a non-operational predicate

  $$\mathbf{dep}(\mathbf{p}) = (\mathbf{rd}(\mathbf{p}) \cup \mathbf{wr}(\mathbf{p})) \times \mathbf{wr}(\mathbf{p})$$

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

The predicates in the postcondition of an operation are assumed to be control dependent on the predicates in the precondition (Bruckner and Wehrheim, 2005).

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

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

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

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

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

In order to analyse dependencies between data in different operations, we need to consider the possible life histories of objects of the class. A UML class **C** may have a protocol state machine $\mathbf{SM_C}$ as its **classifierBehavior** (Lano, 2009b), this state machine defines what sequences of operations can be applied to the object, and under what conditions. $\mathbf{SM_C}$ can be used as the basis of the data and control flow graph $\mathbf{G_C}$ of the class **C**. We carry out normalisation of the pre and post-conditions of each transition of **t**, so that these are conjunctions of predicates.

The primary nodes of $\mathbf{G_C}$ are:

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

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

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

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

The dependency relation $\rho_{\mathbf{C}}$ is then the transitive closure of the union of the data-dependency and control-flow relations on predicates.

The following algorithm is used to compute a slice, with respect to a state **s** and set **V** of features, from the graph $\mathbf{G_C}$. We associate a set $\mathbf{V_x}$ of features to each basic state node of the data and control flow graph.

1. Initialise each $\mathbf{V_x}$ with the empty set of features, except for the target state **s**, which has the set **V** of features.
2. For each transition $\mathbf{t} : \mathbf{s}1 \rightarrow \mathbf{s}2$, add to $\mathbf{V_{s1}}$ the set $\rho^{-1}_{\mathbf{op}}(\mid \mathbf{V_{s2}} \mid)$ of features upon which $\mathbf{V_{s2}}$ depends, via the version of the operation **op** executed by this transition (with precondition and postcondition formed from both the class and state machine predicates for **op** and **t**).

   Mark as included in the slice those predicates of $\mathbf{pre_t}$, $\mathbf{post_t}$ which are in $\rho^{-1}_{\mathbf{C}}(\mid \mathbf{ps} \mid)$ where **ps** is the set of predicates contained in $\mathbf{post_t}$ which have write occurrences of features in $\mathbf{V_{s2}}$. The set of features (with **pre** removed) used in the marked predicates are also added to $\mathbf{V_{s1}}$.

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

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

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

3 STATE MACHINE SLICING

Slicing can be carried out for UML 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 a selected state of the machine (Lano, 2009a).

The following transformations are used to slice state machines:

- Delete states which cannot occur in paths from the initial state to the selected state, and delete the transitions incident to the deleted states.
- Slice transition actions to remove assignments which cannot affect the value of the variables of interest in the selected state.
- Delete transitions with a **false** guard.
- Merge two transitions which have the same sources, targets and actions. The guard of the resulting transition is the disjunction of the original guards.
- Replace a feature **v** by a constant value **e** throughout a state machine, if **v** is initialised to **e** on the initial transition of the state machine, and is never subsequently modified.
- Merge a group **K** of states into a single state **k** if

the states are connected only by actionless transitions and all transitions which exit **K** are triggered by events distinct from any of the events that trigger internal transitions of **K**.

State machine slicing for behaviour state machines of objects and operations has been implemented in the UML2Web tool (Lano, 2008).

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 $\mathbf{M}1 \rightarrow \mathbf{M}2$ (**M**1 sends messages to **M**2) form a tree structure.

4 SUMMARY

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

ACKNOWLEDGEMENTS

The work described here has been funded by the UK EPSRC project SLIM.

REFERENCES

Bruckner, I. and Wehrheim, H. (2005). Slicing Object-Z specifications for verification. In *ZB 2005, LNCS 3455*, pages 414–433. Springer-Verlag.

Harman, M., Binkley, D., and Danicic, S. (2003). Amorphous program slicing. *Journal of Systems and Software*, 68(1):45 – 69.

Lano, K. (2008). Constraint-driven development. *Information and Software Technology*, 50:406–423.

Lano, K. (2009a). Slicing of UML state machines. In *AIC '09*.

Lano, K. (2009b). *UML 2 Semantics and Applications*. Wiley.

Wu, F. and Yi, T. (2004). Slicing Z specifications. *ACM Sigplan*, 39(8).