Helsinki University of Technology Laboratory for Theoretical Computer Science
Technical Reports 25
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tekninen raportti 25
Espoo 2007 HUT-TCS-B25

A SLICER FOR UML STATE MACHINES

Vesa Ojala

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Technical Reports 25
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tekninen raportti 25
Espoo 2007 HUT-TCS-B25

A SLICER FOR UML STATE MACHINES

Vesa Ojala

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK, FINLAND
Tel. +358 9 451 1
Fax. +358 9 451 3369
E-mail: lab@tcs.tkk.fi
URL: http://www.tcs.tkk.fi/

ISBN 978-951-22-9195-3
ISSN 0783-540X

Multiprint Oy
Espoo 2007

ABSTRACT: This document describes the data structures and algorithms used in an implementation of a slicer for UML state machines developed in the SMUML project.

KEYWORDS: UML, state machines, slicing

CONTENTS

1 INTRODUCTION

Concurrent and distributed systems can be modelled with UML state machines. With model checking techniques a set of UML state machines working in parallel, accessing variables in the UML classes, and sending signals to other UML state machines can be checked against a specified property. The usage of slicing techniques (see, for example, the survey [6]) to the state machines might ease the model checking task by removing triggers, guards, and effects that do not affect the functionality of interest. This document introduces a way to implement a UML state machine slicer.

Slicing is done in three parts, first we construct a control flow graph from the UML model (Section 3), then we calculate various dependencies between the control flow graph's nodes (Section 4), and in the third part we calculate the slice using the control flow graph and the calculated dependencies as a guide (Section 5). The slice is always generated with respect to some slicing criterion, in our case this is a set of transitions in a collection of UML state machines.

2 SUPPORTED FEATURES OF UML STATE MACHINES AND ACTION LANGUAGE

We use UML 1.4 as our UML version [4]. In this document it is assumed that the reader is familiar with basic UML terminology. We use Jumbala [2] as our action language for guards and effects.

We support UML state machines that have only one composite state (top), otherwise composite states are forbidden. Simple and final states are supported. From pseudostates, only initial and choice states are supported. Every other state type is unsupported. States cannot have any internal behaviour like entry, exit or doActivities.

Only SignalEvents are supported as triggers of transitions in UML state machines. Also, only primitive Jumbala data types (int and boolean) are supported with the exception of targets in 'send' statements which have to be references to objects. Currently arrays are not supported.

Effects of transitions are Jumbala statements with at most one primitive operation (i.e., an assignment, an assertion, or a Jumbala 'send' statement).

In this work, we assume a UML model to have a fixed set of classes C and a fixed set of variable names (variables) V. By *defining* a variable, we mean that the variable is assigned with a new value. *Referencing* a variable means that a variable is read. For example in the Jumbala expression '$x = a + b;$', x is defined, a and b are referenced.

A variable is *local* in an expression if it is accessed simply by its name, for example x and y are local in the expression '$x = y + 1;$'. A variable is not local in an expression if it is accessed using a dereference. For example $a.x$ is a dereference to attribute x in object a. Thus $a.x$ is not local in the expression '$a.x = y + 1;$'. All parameters in triggers are assumed to be local.

We encapsulate a variable (the variable name and the class to which the variable belongs) and the information whether it is local in an expression in the attributes of a new structure VAR. We will have a VAR structure for every variable identifier in an expression. The variable name $v \in V$ associated with a VAR structure can be accessed using the attribute *var* and the boolean attribute *local* tells whether the variable is local in the expression to which the VAR structure belongs. The attribute *class* contains the class $c \in C$ to which the variable belongs. With VAR structures we are not capable of distinguishing between the attributes of different instances of the same class because VAR structures do not have an attribute for the object to which the variable associated with the VAR structure belongs. Such an attribute is not needed (and would not be easily computable) because we are using only static analysis methods for dependency calculation.

A VAR structure a is said to *represent* its associated variable $a.var \in V$. Two VAR structures a and b are *equal* (denoted $a = b$) if $a.var = b.var$ and $a.class = b.class$. If we have a set W of VARs, the set of variables represented by the members of W is $vars(W) = \{x.var \mid x \in W\}$. A VAR is said to be referenced in an expression if the variable it is representing is referenced in the expression the VAR is associated with and a VAR is said to be defined if the variable it is representing is defined in the expression the VAR is associated with.

3 CONTROL FLOW GRAPH

In the first phase of our implementation, a control flow graph, CFG, is constructed. It captures all the possible executions of the UML state machines from which it is generated.

A control flow graph $CFG = (N, E)$ is a directed graph, where N is a set of nodes in the graph and $E \subseteq N \times N$ is a set of directed edges in the graph. A node $s \in N$ is a *predecessor* of a node $t \in N$ iff there is an edge $(s, t) \in E$. Node $t \in N$ is a *successor* of node $s \in N$ iff there is an edge $(s, t) \in E$. We define $n.preds$ to be the set of predecessors of node n and a set $n.succs$ to be the set of successors of n. An *end node* is a node with no successors.

In the CFG, different parts of a UML state machine's transition are broken up into multiple CFG nodes. We have three different kinds of CFG nodes, BRANCH, SIMPLE, and SEND. Triggers and guards are represented by BRANCH nodes, effects are represented by SIMPLE and SEND nodes. SIMPLE and SEND nodes have one successor while BRANCH nodes can have any number of successors. Because our UML state machine states do not have internal behavior, we have to consider only transitions in the CFG construction.

A CFG is generated from a UML model by traversing all (UML) state machines in the model. For each state s, the following CFG nodes are generated to represent the transitions leaving state s. Nodes generated (wholly or partly) from the transitions in the slicing criterion constitute the slicing criterion in the CFG.

3.1 Branch Nodes

For every state in the UML state machine, there is one BRANCH node. The BRANCH node contains one ELEMENT substructure for every outgoing transition in the original UML state machine. The ELEMENT contains trigger and guard parts from the transition. Every ELEMENT has one successor which represents the effect part of the transition. Even if the effect part is empty, there is an empty CFG node corresponding to that effect. The successors of the individual ELEMENTs form the successors of the BRANCH node itself. For final states, a BRANCH f with zero ELEMENTs is generated. Therefore, f does not have any successors.

In every ELEMENT there is a trigger part and a guard part. Only the trigger part can contain variable definitions and these definitions correspond to the trigger parameters.

To be able to replace unimportant signal parameters by dummy ones, we have to keep a record of parameters that might be important with respect to our slicing criterion. The trigger includes information about every signal parameter using PARAMETER structures that are part of the trigger part of the ELEMENT.

Figure 1 illustrates the relationship between a UML state machine's state and a BRANCH node in the CFG.

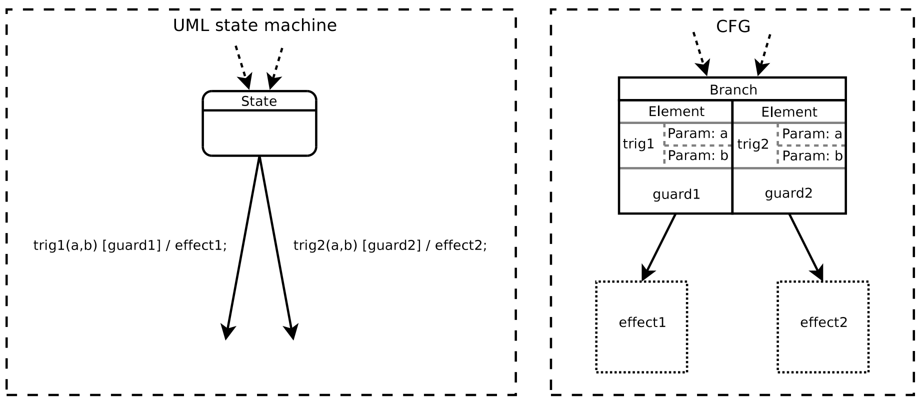

Figure 1: BRANCH node constructed from UML state machine's state. Effects in the picture can contain one or more CFG nodes. Nodes corresponding to effects are discussed in more detail in Section 3.2

When the execution arrives in the BRANCH node, execution proceeds to the next node through one of the ELEMENTs in the node. An ELEMENT is *enabled* if its guard is true and a correct type of signal specified by the trigger is received. A missing trigger (completion transition) or guard is interpreted as if the correct type of signal is received or the guard is true. If every ELEMENT in the node has a guard or a trigger, there is a chance for a situation where no ELEMENT is enabled. This causes the execution to be halted in that state machine until one of the ELEMENTs is enabled, which might be never. In such cases a self-loop to the BRANCH node has to be added to ensure that our algorithms produce the correct outcome. Adding a self-loop allows infinite control flow paths where the execution stays indefinitely at the BRANCH after it has arrived to the branch. Adding a self-loop is done by adding one extra ELEMENT whose successor is the BRANCH node to which it belongs. This ELEMENT does not have anything as a trigger or a guard. Because transitions starting from choice states do not have triggers and one of these transitions is guaranteed to be enabled whenever the execution is in such a state (as required by the UML specification [4]) we do not add self-loops to BRANCHes that represent choice states. Hatcliff et al. solved a similar issue arising in concurrent Java programs with a new dependence called ready depencence [3] but the resulting slice can be shown to be the same with both approaches.

Every PARAMETER p has a set $p.def = \{v\}$, where v is the VAR p defines. Every ELEMENT has sets *def* and *ref*. The *def* of ELEMENT e is a set of VARs defined in e. The only place in ELEMENTs where variables are defined is the PARAMETERs of e, because UML specification forbids side-effects in the guard. Therefore, $e.def = \{v \mid v \in p.def$ for some PARAMETER p in $e\}$. The *ref* of ELEMENT e is the set of VARs referenced in e. Because no variables can be referenced in the trigger of a transition, all the references in e come from the guard. The *def* of a BRANCH node is the union of the *def*s of its ELEMENTs and the *ref* of the BRANCH node is the union of the *ref*s of its ELEMENTs.

3.2 CFG Nodes Corresponding to Effects of Transitions

If the effect of a transition sends a signal, then we add one SEND node for the actual sending operation and a SIMPLE node for every signal parameter to take into account the possible side effects of the parameter evaluation. Figure 2 clarifies the construction.

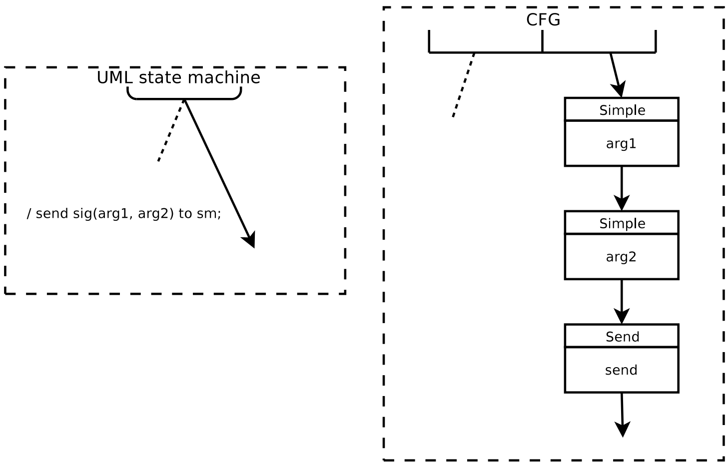

Figure 2: CFG nodes constructed from signal sending transition effect

The first successor of the corresponding BRANCH is a SIMPLE node which *evaluates* the first signal parameter, followed by a node that evaluates the second signal parameter which is followed by the node that evaluates the third one and so forth. After the node evaluating the last signal parameter is the SEND node which corresponds to the actual signal sending operation. Its successor is the BRANCH node corresponding to the UML state machine's state that is the target of the corresponding UML state machine's transition.

If there are no signal parameters, then only the SEND node is generated.

For non-sending effects, a SIMPLE node is generated in the CFG. Its predecessor is the ELEMENT corresponding to the transition to which the effect belongs. The successor of the node is the BRANCH corresponding to the transition's target.

If the transition does not have an effect, an empty SIMPLE node is generated as an effect.

Every SIMPLE and SEND node has sets *def* and *ref* which contain a set of VARs, representing variables defined in that node and a set of VARs for variables referenced in that node, respectively.

4 DEPENDENCY RELATIONS

To be able to deduce on which parts of the model the nodes in our slicing criterion are dependent, we calculate different dependencies between parts of the model. Data dependencies and interference dependencies describe the possibility of an action in a state machine to define a variable that is later referenced in a context belonging to the slice, thus making the location of the definition an important one with respect to our slicing criterion. Control dependencies are concerned with whether a node is part of an execution or not.

Sections 4.1 and 4.2 describe how data dependencies are calculated. Section 4.3 describes interference dependencies and Section 4.4 control dependencies.

4.1 Reaching Definitions

Data dependencies are calculated in two parts: first we calculate reaching definitions and based on that information we calculate actual data dependencies.

A *data flow path* from a_1 to a_i is a sequence $a_1, a_2, \ldots, a_{i-1}, a_i$, where every $a_j \in \{a_1, \ldots, a_i\}$ is either of the form n, where n is a SIMPLE, SEND or end node (the type of end node is BRANCH), or $b.e$, where b is a BRANCH node and e is an ELEMENT in b. For every consecutive pair (a_j, a_{j+1}) in the path, the following must hold:

- if both a_j and a_{j+1} are SIMPLE, SEND or end nodes (only a_{j+1} can be an end node) there must be an edge from a_j to a_{j+1}
- if a_j is a SIMPLE or SEND node and $a_{j+1} = b.e$, where b is a BRANCH node and e is an ELEMENT in b, there must be an edge from a_j to b
- if $a_j = b.e$, where b is a BRANCH node and e is an ELEMENT in b and a_{j+1} is a SIMPLE, SEND, or end node, a_{j+1} must be a successor of e
- if $a_j = b_j.e_j$, where b_j is a BRANCH node and e_j is an ELEMENT in b_j, and $a_{j+1} = b_{j+1}.e_{j+1}$, where b_{j+1} is a BRANCH node, and e_{j+1} is an ELEMENT in b_{j+1}, b_{j+1} must be a successor of e_j

For example, in figure 3 there is a data flow path $a, b.d, e, f.h$ between the SIMPLE node a and the ELEMENT h belonging to the BRANCH node f.

Every node n has a set $n.\mathit{dfpreds}$ which contains the SIMPLE nodes, SEND nodes and ELEMENTs that have n as their successor.

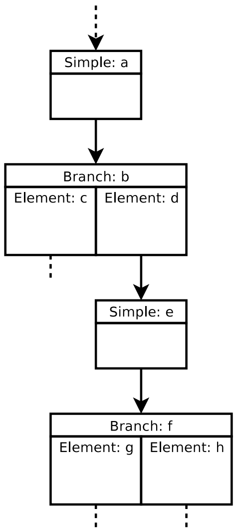

Figure 3: A part of a CFG which has a data flow path $a, b.d, e, f.h$ between the SIMPLE node a and the ELEMENT h

A definition of a variable v in a SIMPLE node s *reaches* a SIMPLE, SEND or end node t (written $(v, s) \overset{def}{\rightsquigarrow} t$) if there is a data flow path $s = a_1, \ldots, a_j = t$ from s to t in the CFG and $v \notin \bigcup_{a \in \{a_{i+1}, \ldots, a_{j-1}\}} vars(a.def)$ and $v \in vars(s.def)$.

A definition of a variable v in a SIMPLE node s *reaches* an ELEMENT e in a BRANCH node b (written $(v, s) \overset{def}{\rightsquigarrow} b.e$) if there is a data flow path $s = a_1, a_2, \ldots, a_j = b.e$ from s to $b.e$ in the CFG and $v \in vars(s.def)$ and $v \notin \bigcup_{a \in \{a_{i+1}, \ldots, a_{j-1}\}} vars(a.def)$.

A definition of a variable v in a PARAMETER p in an ELEMENT e in a BRANCH node b *reaches* a SIMPLE, SEND or end node t (written $(v, p) \overset{def}{\rightsquigarrow} t$) if there is a data flow path $b.e = a_1, a_2, \ldots, a_j = t$ in the CFG and $v \in vars(p.def)$ and $v \notin \bigcup_{a \in \{a_{i+1}, \ldots, a_{j-1}\}} vars(a.def)$.

A definition of a variable v in a PARAMETER p in an ELEMENT e_1 in a BRANCH node b_1 *reaches* an ELEMENT e_2 in a BRANCH node b_2 (written $(v, p) \overset{def}{\rightsquigarrow} b_2.e_2$) if there is a data flow path $b_1.e_1 = a_1, a_2, \ldots, a_j = b_2.e_2$ from $b_1.e_1$ to $b_2.e_2$ in the CFG and $v \notin \bigcup_{a \in \{a_{i+1}, \ldots, a_{j-1}\}} vars(a.def)$ and $v \in vars(p.def)$.

We calculate *defs_in* sets for every node in the CFG. The *defs_in* set contains information about the definitions that reach the node. The definitions that reach a BRANCH node are the same that reach every ELEMENT in the BRANCH node. We calculate *defs_out* sets for SIMPLE and SEND nodes in the CFG as well as for ELEMENTs in every BRANCH node in the CFG. The *defs_out* set contains information about the definitions that reach the successor of the node or ELEMENT by an edge from the node or ELEMENT to its successor.

Formally, for a node t in the CFG $t.defs\_in = \{(v, a) \mid (v, a) \overset{def}{\rightsquigarrow} t\}$, where v is a variable and a is either a SIMPLE node or a PARAMETER. For SIMPLE and SEND nodes $n.defs\_out = \{(w, a) \mid (w, a) \in n.defs\_in \wedge w \notin vars(n.def)\} \cup \{(v, n) \mid v \in vars(n.def)\}$. For BRANCH nodes *defs_out* sets are defined for every ELEMENT e in the BRANCH node n:

$$\begin{aligned} e.defs\_out \;=\; & \{(v,p) \mid v \in vars(p.def) \wedge p \text{ is a PARAMETER in } e\text{'s trigger}\} \cup \\ & \{(w,a) \mid (w,a) \in n.defs\_in \wedge w \notin vars(e.def)\} \end{aligned}$$

Algorithm 1 calculates reaching definitions.

Algorithm 1 Reaching Definitions

SIMPLE and SEND nodes have *defs_in* and *defs_out* sets that contain (v,a) pairs, where v is a variable and a is a node or a PARAMETER that defines v.
BRANCH nodes have *defs_in* sets just like the other nodes but they may have multiple *defs_out* sets, one for each ELEMENT.
All *defs_in* and *defs_out* sets are initially empty.

repeat
    for all $node \in N$ **do**
        $node.defs\_in \leftarrow \bigcup_{pred \in node.dfpreds} pred.defs\_out$
        if *node* type is SIMPLE or SEND **then**
            $A \leftarrow \{(v,n) \mid (v,n) \in node.defs\_in \wedge v \notin vars(node.def)\}$
            $B \leftarrow \{(v,node) \mid v \in vars(node.def)\}$
            $node.defs\_out \leftarrow A \cup B$
        if *node* type is BRANCH **then**
            for all ELEMENTs e in *node* **do**
                $A \leftarrow \{(v,n) \mid (v,n) \in node.defs\_in \wedge v \notin vars(e.def)\}$
                $B \leftarrow \{(v,p) \mid v \in vars(p.def) \wedge p \text{ is a PARAMETER in } e\}$
                $e.defs\_out \leftarrow A \cup B$
until *defs_in* and *defs_out* sets do not change from the last iteration

After the execution of the algorithm, every node has a set *defs_in* that contains pairs (v,a) where a is a SIMPLE node or a PARAMETER and v is a variable. Let $m \in N$ be a node in the CFG. The set $\{a \mid (v,a) \in m.defs\_in\}$ is the set of SIMPLE nodes and PARAMETERs that may have defined the current value of the variable v when entering the node m. Let l be a SIMPLE or SEND node or ELEMENT in the CFG. The set $\{a \mid (v,a) \in l.defs\_out\}$ is the set of SIMPLE nodes and PARAMETERs that may have defined the current value of the variable v when exiting the node or ELEMENT l.

4.2 Data Dependencies

A SIMPLE node t is *data dependent* on a SIMPLE node s, if for some $v \in vars(t.ref)$, $(v,s) \overset{def}{\rightsquigarrow} t$. A SIMPLE node t is *data dependent* on a PARAMETER p in an ELEMENT e in a BRANCH b, if for some $v \in vars(t.ref)$, $(v,p) \overset{def}{\rightsquigarrow} t$. A BRANCH node b is *data dependent* on a SIMPLE node s, if for some ELEMENT e in b and $v \in vars(e.ref)$, $(v,s) \overset{def}{\rightsquigarrow} b.e$. A BRANCH node b_2 is *data dependent* on a PARAMETER p in an ELEMENT e_1 in a BRANCH b_1, if for some ELEMENT e_2 in b_2 and $v \in vars(e_2.ref)$, $(v,p) \overset{def}{\rightsquigarrow} b_2.e_2$.

A PARAMETER has always an associated signal sig. A PARAMETER p is data dependent on a node q if q evaluates an expression whose value gets assigned to the PARAMETER p when a signal sig is received. Here, node q is one of the CFG nodes generated when transforming a signal send operation[1] into a sequence of SIMPLE nodes as described in Section 3.2. In figure 4, data dependencies marked with (c) are obtained in this way.

When we already have calculated reaching definitions, we can get data dependencies by algorithm 2. As it can be seen from the algorithm, the whole BRANCH b node becomes data dependent on a if at least one of the ELEMENTs belonging to b is data dependent on a.

Algorithm 2 Data Dependencies

1: Every node and PARAMETER has a set *dd* which will contain the nodes and the PARAMETERs on which it is data dependent.

2: **for all** $n \in N$ **do**
3:     **if** n type is SIMPLE **then**
4:         $n.dd \leftarrow \{m \mid (v, m) \in n.defs\_in \wedge v \in vars(n.ref)\}$
5:     **if** n type is BRANCH **then**
6:         $n.dd \leftarrow \{m \mid (v, m) \in n.defs\_in \wedge v \in vars(n.ref) \wedge v \notin vars(n.def)\}$
▷ all definitions come from PARAMETERs
7:         **for all** ELEMENTs e in n **do**
8:             **for all** PARAMETERs p in e **do**
9:                 **if** $vars(p.def) \subseteq vars(e.ref)$ **then**
10:                     $n.dd \leftarrow n.dd \cup \{p\}$
11:             ▷ PARAMETERs depend on their evaluators before send
12:             **for all** PARAMETERs p in e **do**
13:                 $p.dd \leftarrow \{m \mid m$ evaluates p's value before signal send$\}$

In line 4 in algorithm 2, data dependencies for SIMPLE nodes are calculated. In line 6, dependencies to nodes that define variables that are referenced in a BRANCH node, but are not defined by any signal parameters, are calculated. The for-loop beginning from line 8 calculates data dependencies to PARAMETERs in node n that define variables referenced in the same node. Finally, for all PARAMETERs, data dependencies are calculated in the for-loop beginning from line 12. Figure 4 illustrates the data dependence calculation for BRANCH nodes. Because there are no variables referenced in SEND nodes, only SIMPLE and BRANCH nodes need to be considered here.

4.3 Interference Dependencies

When the variable v belonging to a class c is referenced in a node s and defined in node t, node s is *interference dependent* on node t if s and t are in different state machines or in different instances of the same state

[1] These signal send operations can happen in other UML state machines but they can happen also in the same state machine. Because we may have multiple instances of one class, the sends in the same state machine have to be taken into account.

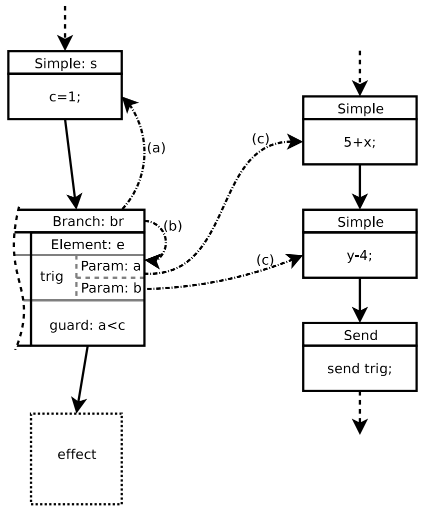

Figure 4: (a) br is data dependent on s, because the definition of c in s reaches br and there is no signal parameter defining c in br. (b) BRANCH br is data dependent on PARAMETER a because it defines a variable that is referenced in br. (c) PARAMETERs are data dependent on their evaluators.

machine but still can access the same variable v in class c. This can happen if the access to v is not local either in s or in t. The same concept applies also if instead of node t PARAMETER p defines the variable v. It should be remembered though that in PARAMETERs the occurrences of the variables are always local as described in Section 2.

Formally, a node s is interference dependent on a node or a PARAMETER t if $v \in s.ref$, $w \in t.def$, $v = w$, and $\neg v.local \vee \neg w.local$. Hatcliff et al. defined interference dependency between threads in a Java program in the same way [3]. Algorithm 3 calculates interference dependencies.

4.4 Control Dependencies

A node n_i is control dependent on a node n_j if the execution of n_j decides whether n_i is executed or not. In figure 5, the node d is control dependent on the node a because the branch taken at node a decides whether we execute node d or not.

Conventional algorithms for control dependency calculation assume that programs have unique start and end nodes and that only paths ending in the unique end node need to be considered. In reactive systems, these assumptions do not always hold. Because we want our slicer to work correctly with reactive systems we are using two dependencies introduced by Ranganath et al. and algorithms for calculating them [5]. These dependencies are non-termination sensitive control dependencies and decisive control dependencies and we shall examine those more closely in Sections 4.4 and 4.4, respectively.

For discussion of control dependencies we define a *control flow path*. A control flow path from n_1 to n_j in a CFG is a sequence of nodes $n_1, n_2, \ldots, n_j$ such that for every consecutive pair of nodes (n_k, n_{k+1}) in the path there is an edge $(n_k, n_{k+1}) \in E$. A control flow path is *maximal* if it is infinite or it

Algorithm 3 Interference Dependencies

every node has a set id that contains nodes and PARAMETERs on which the node is interference dependent.

for all $n \in N$ **do**
  $n.id \leftarrow \emptyset$
  for all $v \in n.ref$ **do**
    $n.id \leftarrow n.id \cup$ FIND_DEFS(v)

function FIND_DEFS(v)
  $deps \leftarrow \emptyset$
  for all $m \in N$ **do**
    if m type is SIMPLE **then**
      for all $w \in m.def$ **do**
        if $v = w \wedge (\neg v.local \vee \neg w.local)$ **then**
          $deps \leftarrow deps \cup \{m\}$
    if m type is BRANCH **then**
      for all ELEMENTs e in m **do**
        for all PARAMETERs p in e **do**
          pick $w \in p.def$ $\triangleright$ $|p.def| = 1$
          if $v = w \wedge \neg v.local$ **then**
            $deps \leftarrow deps \cup \{p\}$
  return $deps$

ends in an end node.

Node n strictly precedes any occurrence of node m in a control flow path if n occurs in the control flow path and either node m does not occur in the control flow path or the first occurrence of node n in the control flow path is earlier than the first occurrence of node m in the control flow path.

Non-Termination Sensitive Control Dependencies

Node n_i is *non-termination sensitive control dependent* on node n_j iff n_j has at least two successors, n_k and n_l, such that

- for all maximal control flow paths from n_k, n_i always occurs and either $n_j = n_i$ or n_i strictly precedes any occurrence of n_j, and
- there exists a maximal control flow path from n_l on which either n_i does not occur, or n_j strictly precedes any occurrence of n_i.

In figure 5, node a has two successors, b and c. In all maximal control flow paths from c, d always occurs and it strictly precedes any occurrence of a, so the first condition is fulfilled. In all maximal control flow paths from b, a strictly precedes any occurrence of d, so the second condition is fulfilled and we can conlude that d is non-termination sensitive control dependent on a. Note that there are maximal control flow paths from b where d occurs but there is always an occurrence of a before the occurrence of d. So, the examination is limited between two successive visits in a (or n_j).

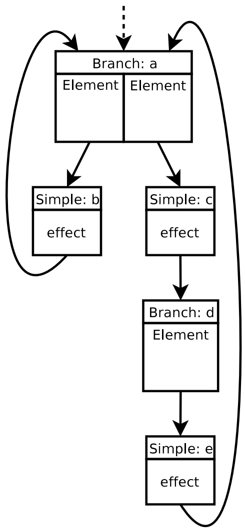

Figure 5: Nodes b, c, d, and e are non-termination sensitive control dependent on node a

Algorithm 4 is used for calculating non-termination sensitive control dependencies and it is based on the algorithm presented by Ranganath et al. [5]. In the algorithm, notation $S[a, b]$ is used to represent a value obtained from a dictionary S using the pair (a, b) as key value. $\langle a, b\rangle$ represents a unique symbolic value for the pair (a, b).

In algorithm 4, values $\langle a, b\rangle$ are used for marking all maximal control flow paths beginning from node a where the second node in the control flow path is the node b. The dictionary S contains a set of these values for every pair (c, d), where c is a node in the CFG and d is a BRANCH node in the CFG. The values in the set represent a set of maximal control flow paths which start at node $d = a$ and contain node c.

The algorithm begins by initializing the dictionary S in a way that for every BRANCH node b in the CFG, for every successor s of b, $S[s, b] \leftarrow \{\langle b, s\rangle\}$. Then the algorithm propagates these symbols from the node to the successor of the node if appropriate.

Finally, every node n is made to be non-termination sensitively control dependent on a BRANCH node b if b has more successors than there are symbols in the set $S[n, m]$ and $S[n, m] \neq \emptyset$. This is because every maximal control flow path starting from b passes through one of b's successors s (represented by $\langle b, s\rangle$). When $0 < |S[n, b]|$, there is a maximal control flow path including n, starting from b, that passes through one of b's successors s_1, and therefore $\langle b, s_1\rangle \in S[n, b]$. On the other hand, when $|S[n, b]| < |b.succs|$, there is a maximal control flow path that does not include n, starting from b, that passes through one of b's successors s_2, and therefore $\langle b, s_2\rangle \notin S[n, b]$.

Decisive Order Dependencies

In figure 6, node g is data dependent on nodes a and h. The choice on node b decides whether the assertion on node g holds, so g has to be control dependent on node b in some way. We check whether g is non-termination sensitive control dependent on node b.

Algorithm 4 Non-Termination Sensitive Control Dependencies

Every node has a set $ntscd$ which contains nodes on which it is ntsc-dependent.
$workbag \leftarrow \emptyset$
S : empty dictionary which will contain symbols representing a pair of nodes

for all $n \in N$, where $|n.succs| > 1$ **do**
    for all $s \in n.succs$ **do**
        $S[s, n] \leftarrow \{\langle n, s \rangle\}$
        $workbag \leftarrow workbag \cup \{s\}$
while $workbag \neq \emptyset$ **do**
    pick $n \in workbag$
    $workbag \leftarrow workbag \setminus \{n\}$
    if n has a unique successor s and $s \neq n$ **then**
        ▷ all maximal control flow paths starting from n include also s
        for all $m \in N$, where $|m.succs| > 1 \wedge S[n, m] \setminus S[s, m] \neq \emptyset$ **do**
            $S[s, m] \leftarrow S[s, m] \cup S[n, m]$
            $workbag \leftarrow workbag \cup \{s\}$
    if $|n.succs| > 1$ **then**
        for all $m \in N$ **do**
            if $|S[m, n]| = |n.succs|$ **then**
                ▷ all maximal control flow paths starting from n also include m
                for all $p \in \{q \in N \mid |q.succs| > 1 \wedge q \neq n\}$ **do**
                    if $S[n, p] \setminus S[m, p] \neq \emptyset$ **then**
                        $S[m, p] \leftarrow S[m, p] \cup S[n, p]$
                        $workbag \leftarrow workbag \cup \{m\}$
for all $n \in N$ **do**
    for all $m \in N$, where $|m.succs| > 1$ **do**
        if $0 < |S[n, m]| < |m.succs|$ **then**
            $n.ntscd \leftarrow n.ntscd \cup \{m\}$

Node b has two successors, c and d. For all maximal control flow paths from c, g always occurs and strictly precedes any occurrence of b. Therefore the first condition of non-termination sensitive control dependence is fulfilled. But there exists no control flow path from d where either g does not occur or b strictly precedes any occurrence of g, and thus g is not non-termination sensitive control dependent on node b. The same goes for node h. If our slicing criterion is the set $\{g\}$, node b does not belong to our slice even though it should: Because b does not belong to the slice, guards from the transitions corresponding to node b are removed, and thus the sliced state machine has executions where the assertion at node g is false even though there were no such executions in the original state machine. What node b actually decides is the order of execution of nodes e, f, g, and h. The calculation of a slice is discussed in more detail in Section 5.

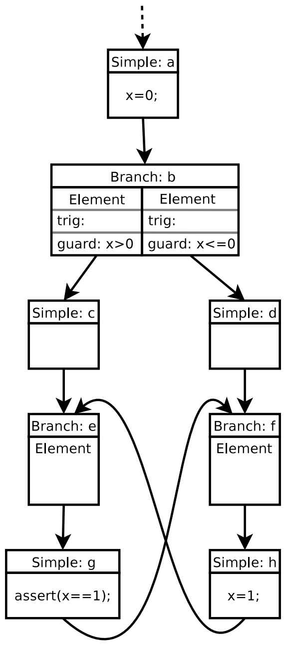

Figure 6: Nodes g and h are decisively order dependent on node b

Decisive order dependence captures the dependence where the order of execution of two nodes depends on the choice made in a third node. In any control flow graph, two nodes m and p are *decisively order dependent on* n if:

- all maximal control flow paths from n contain both m and p, and
- n has a successor from which all maximal control flow paths contain m before any occurrence of p, and
- n has a successor from which all maximal control flow paths contain p before any occurrence of m.

In figure 6, nodes g and h are decisively order dependent on node b. Now, because g belongs to our slicing criterion and because g is data dependent on node h, also node b belongs to the slice[2].

Algorithm 6 is used for calculating decisive order dependencies. It is based on the algorithm presented by Ranganath et al. [5]. Function ALLREACH(a,b) (presented in algorithm 5) returns true if b can be reached from a by all maximal control flow paths. This all-path reachability is calculated beforehand by ALLREACH_INIT and ALLREACH only looks from the dictionary whether the value should be true or false.

In lines 5 and 6 in algorithm 6 the three conditions for decisive control dependence are checked. The first condition is checked by the two ALLREACH calls in the line 5. The second and the third condition are checked by the DEPEND function called in the line 6. The DEPEND function colors nodes of the CFG so that m and p are colored white and black, respectively. The

[2] If h is not in the slice, b would not be in the slice either because both g and h are needed for decisive order dependency.

Algorithm 5 All-path reachability

$reachability$: dictionary - contains for all nodes a set of nodes from which the node is reachable by all control flow paths

function ALLREACH_INIT(G)
    $workbag \leftarrow N$
    for all $n \in N$ **do**
        $reachability[n] \leftarrow \{n\}$
    while $|workbag| > 0$ **do**
        pick $n \in workbag$
        $workbag \leftarrow workbag \setminus \{n\}$
        $new\_set \leftarrow \bigcap_{m \in n.succs} reachability[m]$
        if $(new\_set \not\subseteq reachability[n])$ **then**
            $reachability[n] \leftarrow reachability[n] \cup new\_set$
            $workbag \leftarrow workbag \cup n.preds$

function ALLREACH(n,m)
    if $m \in reachability[n]$ **then**
        return $true$
    else
        return $false$

predecessor of a node is colored white (black) if all its successors are colored white (black), the nodes that have children of different colors or uncolored children are uncolored. After the coloring process, if n has a white child and a black child, nodes m and p are decisively order dependent on node n.

5 CALCULATING THE SLICE

Given the slicing criterion as a set of nodes in the CFG (see Section 3), the slice is calculated as the smallest set of nodes which includes all nodes in the slicing criterion and is closed under the dependencies described earlier.

The slice is a set of nodes and PARAMETERs. If a BRANCH node belongs to the slice it does not directly imply that the PARAMETERs belonging to the BRANCH are also included in the slice. This prevents the inclusion of unnecessary signal parameters in the slice. A signal parameter is added to the slice only if a node in the slice is actually dependent on it. Although the inclusion of a BRANCH node in the slice does not force all of its PARAMETERs to be included in the slice, the inclusion of a PARAMETER in the slice does effectively imply the inclusion of the PARAMETER's corresponding BRANCH node to which the PARAMETER belongs in the slice through function PART_IN_SLICE.

The function PART_IN_SLICE(*node*) returns *true* if *node* is in the slice or if a PARAMETER that belongs to *node* is in the slice, otherwise it returns *false*. In algorithm 7 the slice of the CFG is calculated.

Algorithm 6 Decisive Order Dependencies

every node n has a set *dod*, which includes node pairs (a, b), where b is the node on which n and a are decisively order dependent.
for all $n \in N$, where $|n.succs| > 1$ **do**
    for all $m \in N$ **do**
        for all $p \in N \setminus \{m\}$ **do**
            if ALLREACH$(m, p) \land$ ALLREACH(p, m) **then**
                if DEPEND(n, m, p) **then**
                    $m.dod \leftarrow m.dod \cup \{(p, n)\}$
                    $p.dod \leftarrow p.dod \cup \{(m, n)\}$

function DEPEND(n, m, p)
    color: dictionary indexed with nodes, initially filled with *uncolored*
    $color[m] \leftarrow white$
    $color[p] \leftarrow black$
    $visited \leftarrow \{m, p\}$
    COLORED_DAG$(n, color, visited)$
    return *true* if n has a black and a white child, otherwise *false*

function COLORED_DAG$(n, color, visited)$
    if $n \notin visited$ **then**
        $visited \leftarrow visited \cup \{n\}$
        if $|n.succs| > 0$ **then**
            for all $succ \in n.succs$ **do**
                COLORED_DAG$(succ, color, visited)$
            $c \leftarrow color$[pick $t \in n.succs$]
            for all $succ \in n.succs$ **do**
                if $color[succ] \neq c$ **then**
                    $c \leftarrow uncolored$
                    break
            $color[n] \leftarrow c$
    return

Algorithm 7 Slicing

```
slicingcrit: slicing criterion (a set of nodes in the CFG)

slice ← slicingcrit
repeat
    old_slice ← slice
    for all n ∈ N do
        if PART_OF_SLICE(n) then
            slice ← slice ∪ n.dd ∪ n.ntscd ∪ n.id
            ▷ elements in (n.dd ∪ n.id) either PARAMETERs or SIMPLEs
            slice ← slice ∪ {b ∈ N | ∃m ∈ slice : (m, b) ∈ n.dod}
            if n type is BRANCH then
                for all ELEMENTs e in n do
                    for all i ∈ N do
                        if i sends a signal matching to e's trigger then
                                        ▷ SEND nodes are added to the slice here
                            if i ∉ slice then
                                slice ← slice ∪ {i}

        ▷ handling of PARAMETERs
        if n type is BRANCH then
            for all ELEMENTs e in n do
                for all PARAMETERs p in e do
                    if p ∈ slice then
                        slice ← slice ∪ p.dd
until old_slice = slice
```

After we have calculated the slice of a CFG, we still have to get the slice of the original UML model. When we constructed the CFG from the UML state machines in section 3, we also kept record of the relationship between CFG nodes and UML state machine transitions.

We will remove all parts of transitions whose counterparts in the CFG are not in the slice. We will also replace unused signal arguments with dummy ones. For simplicity, we do not remove any transitions, we modify only their triggers, guards and effects.

For every BRANCH node that is not in the slice, we will remove all triggers and guards from the corresponding UML state machine's state. For sending effects in transitions, we will remove the entire effect if the corresponding SEND node in the CFG is not in the slice. Otherwise we will only replace all unused signal arguments with *NONE*, that is, all arguments whose evaluator node is not in the slice. *NONE* is used as a dummy value for arguments whose values will not be used.

Algorithm 8 maps the slice from the CFG to the original UML model and constructs the sliced UML model by removing uninteresting behavior from the original UML model.

Algorithm 8 From CFG slice to UML slice

```
for all states s in UML model do
    b ← BRANCH representing s
    if ¬PART_OF_SLICE(b) then
        remove all triggers and guards of transitions leaving s
    for all outgoing transitions t from s do
        if effect of t is send then
            if corresponding SEND is not in slice then
                remove the effect from t
            else
                replace all arguments that are not in slice with NONE
        else
            if SIMPLE representing the effect is not in slice then
                remove the effect from t
```

6 IMPLEMENTATION

The algorithms presented in the previous sections have been implemented as a part of the SMUML project to obtain a slicer for UML state machines. The implementation uses the Python API provided by the Coral metamodeling tool [1] to process UML 1.4 models in XMI format. As output the slicer generates another XMI file representing the slice of the model given. The slicing criterion is defined in the input file by adding a new taggedValue named 'slicingCrit' to each transition belonging to the slicing criterion. The dataValue of the taggedValue slicingCrit has no meaning to our implementation.

The sliced model can be analysed with other tools developed in the SMUML project or it can be processed further, for example abstracted using a model abstractor implemented in the SMUML project.

REFERENCES

[1] Marcus Alanen and Ivan Porres. Coral: A metamodel kernel for transformation engines. In *Proc. Second European Workshop on Model Driven Architecture (MDA)*, number 17-04 in Tech. Report, pages 165–170. Computing Laboratory, Univ. of Kent, 2004.

[2] J. Dubrovin. Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2006.

[3] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Static Analysis Symposium (SAS 1999)*, volume 1694 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1999.

[4] OMG unified modelling language specification, version 1.4. Object Management Group, 2001.

[5] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. Technical Report #2004-8, SAnToS Laboratory, Kansas State University, 2006.

[6] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
TECHNICAL REPORTS

HUT-TCS-B12 Kimmo Varpaaniemi
On Computing Symmetries and Stubborn Sets. April 1994.

HUT-TCS-B13 Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, Tino Pyssysalo
PROD Reference Manual. August 1995.

HUT-TCS-B14 Tuomas Aura
Modelling the Needham-Schröder authentication protocol with high level Petri nets. September 1995.

HUT-TCS-B15 Eero Lassila
ReFlEx — an Experimental Tool for Special-Purpose Processor Code Generation. March 1996.

HUT-TCS-B16 Markus Malmqvist
Methodology of Dynamical Analysis of SDL Programs using Predicate/Transition Nets. April 1997.

HUT-TCS-B17 Tero Jyrinki
Dynamical Analysis of SDL Programs using Predicate/Transition Nets. April 1997.

HUT-TCS-B18 Tommi Syrjänen
Implementation of Local Grounding for Logic Programs With Stable Model Semantics. October 1998.

HUT-TCS-B19 Marko Mäkelä, Jani Lahtinen, Leo Ojala
Performance Analysis of a Traffic Control System Using Stochastic Petri Nets. December 1998.

HUT-TCS-B20 Eero Lassila
A Tree Expansion Formalism for Generative String Rewriting. June 2001.

HUT-TCS-B21 Annikka Aalto
Automatic Translation of SDL into High Level Petri Nets. November 2004.

HUT-TCS-B22 Maarit Hietalahti, Mikko Särelä, Antti Tuominen, Pekka Orponen
Security Topics and Mobility Management in Hierarchical Ad Hoc Networks (Samoyed): Final Report. December 2007.

HUT-TCS-B23 Jori Dubrovin, Tommi Junttila
Symbolic Model Checking of Hierarchical UML State Machines. December 2007.

HUT-TCS-B24 Jori Dubrovin, Tommi Junttila, Keijo Heljanko
Symbolic Step Encodings for Object Based Communicating State Machines. December 2007.

HUT-TCS-B25 Vesa Ojala
A Slicer for UML State Machines. December 2007.

ISBN 978-951-22-9195-3
ISSN 0783-540X