A

State-Based Model Slicing: A Survey

KELLY ANDROUTSOPOULOS, University College London
DAVID CLARK, University College London
MARK HARMAN, University College London
JENS KRINKE, University College London
LAURENCE TRATT, King's College London

Slicing is a technique, traditionally applied to programs, for extracting the parts of a program that affect the values computed at a statement of interest. In recent years authors have begun to consider slicing at the model level. We present a detailed review of existing work on slicing at the level of finite state machine-based models. We focus on state based modelling notations because these have received sufficient attention from the slicing community that there is now a coherent body of hitherto unsurveyed work. We also identify the challenges that state based slicing present and how the existing literature has addressed these. We conclude by identifying problems that remain open either because of the challenges involved in addressing them or because the community simply has yet to turn its attention to solving them.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*

General Terms: Design

Additional Key Words and Phrases: Slicing, finite state machines

1. INTRODUCTION

Program slicing is a source code analysis and manipulation technique, in which a subprogram is identified based on a user-specified slicing criterion. The criterion captures the point of interest within the program, while the process of slicing consists of following dependencies to locate those parts of the program that may affect the slicing criterion [Weiser 1979]. Some flavours of slicing merely highlight the identified subprogram within the larger program, while others actively rewrite the program based upon the identified subprogram.

As an increasing portion of software production is done with models – particularly specification and design – researchers have moved from considering only program slicing to model slicing. The need for model slicing is strong: models convey many types of information better than programs, but become unwieldy in scale far quicker.

Software modelling encompasses a number of different languages, with UML – the *de facto* standard modelling language – containing several distinct sub-languages. Each different modelling language needs to be considered differently, with the challenges facing class models (describing static structure) being different than object

Author's address: K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, University College London, Malet Place, London, WC1E 6BT, United Kingdom.
L. Tratt, King's College London, Strand, London WC2R 2LS, United Kingdom
Kelly Androutsopoulos is the primary author.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

models (describing specific instance patterns) or collaboration models (describing behaviour), for example.

In this paper we focus on the modelling notation to which slicing has been most often applied: State-Based Models (SBMs). We use the term SBMs as an umbrella term for a wide-range of related languages (e.g. Extended Finite State Machines, UML statecharts, STATEMATE statecharts and RSML). These languages, typically graphical, are based on finite state machines, often with additional features (e.g. stores, structuring / hierarchical constructs, or explicit parallelism constructs).

1.1. Why slicing state-based models is interesting and useful

Initially, it may seem possible to use program slicing to achieve SBM slicing. However, since this would lead to results too poor for a human to interpret, SBM slicing needs to be considered as a distinct research area. There are two chief reasons for this, which we now address.

1.1.1. Syntactic. SBMs are (visual) graphs whereas programs are sequences of (textual) statements. Program slicing often operates at the most natural human-orientated level of granularity—a line of code. Program slices are thus typically subsets of the lines of code in the original program. SBMs do not have an equivalent level of granularity for slicing to be applied at—an individual node may represent the equivalent of several lines of code, or several nodes may represent the equivalent of a single line of code. Because of this inherent, and unfixed, difference of granularity, translating SBMs into programs may thus lead to slices that make little sense to a modeller.

SBM's graph-based nature also necessitates a totally different approach to rewriting SBMs after slicing. Where program slicing can simply remove lines and be left with a program, SBMs must be 'rewired' to prevent nodes being orphaned; as the literature shows, achieving a good rewriting is non-trivial.

1.1.2. Semantic. An important semantic difference between SBMs and programs is that the majority of state based modelling languages allow non-determinism (i.e. when, in a given state in a SBM, more than one transition can be validly taken), whereas programming languages go out of their way to avoid non-determinism. Translating a non-deterministic state based model into a deterministic programming language requires encoding. Even assuming that an accurate encoding can be found, the program slicing algorithm will have no understanding of it—it is as likely to slice a small part of the encoding as it is any other part of the program. Translating the sliced (encoded) program back into a state based model would then lead to bizarre state machines which would appear to bear little resemblance to the original.

Indeed, the specific set of features present in SBMs presents challenges which, when combined together, have yet to be tackled in program slicing. If one were to view the task that confronts an approach for slicing SBMs through the eyes of traditional program slicing, then the problem would resemble that of slicing a non–deterministic set of concurrently executed procedures with arbitrary control flow. Such a combination of characteristics is not addressed by the current literature on slicing [Binkley and Gallagher 1996; Binkley 2007; Binkley and Harman 2004; Harman and Hierons 2001; Tip 1995].

1.1.3. The need for a survey. In summary: SBM slicing is in many ways substantially different from program slicing; yet important challenges remain unresolved even in common areas between the two. Because of this, many authors have tackled various aspects of SBM slicing. The body of knowledge on SBM slicing is wide, and spread over many different and sometimes disjoint research communities. This paper is the

first survey of SBM slicing, integrating together disparate knowledge and highlighting open problems.

We start with an overview of program slicing (Section 2). We then give an overview of the SBM languages (Section 3). In Section 4 we introduce the running example, an ATM. We then discuss the slicing approaches according to their type (Section 5) before discussing the applications of SBM slicing (Section 6). Finally, we discuss open issues and untackled problems in SBM slicing (Section 7).

2. BACKGROUND: PROGRAM SLICING

Most research into slicing has considered slicing at the program level; we therefore present a brief overview of this 'parent' subject area, as most SBM slicing work builds upon it, directly or indirectly.

Weiser observed that programmers build mental abstractions of a program during debugging; slicing is his formalisation of that process [Weiser 1979]. Weiser defined a slice as any subset of the program which maintains the effect of the original program on the slicing criterion, a pair $c = (s, V)$ consisting of a statement s in the program and a subset V of the program's variables. We now call such a slice an *executable* slice. Slicing has many applications including program comprehension [Harman et al. 2003], software maintenance [Gallagher and Lyle 1991], testing and debugging [Binkley 1998; Harman et al. 2004], virus detection [Lakhotia and Singh 2003], integration [Binkley et al. 1995], refactoring [Komondoor and Horwitz 2000], reverse engineering and reuse [Canfora et al. 1998]. Also, slicing has been used as an optimisation technique for reducing program models or other program representations extracted from programs for the purpose of verification via model checking [Corbett et al. 2000; Jhala and Majumdar 2005; Dwyer et al. 2006].

Since Weiser's seminal work, program slicing has been developed in many ways to include: forward and backward formulations [Horwitz et al. 1990; Binkley and Harman 2005; Fox et al. 2001]; static, dynamic, hybrid formulations [Korel and Laski 1988; Agrawal and Horgan 1990; Gupta et al. 1992]; conditioned formulations [Canfora et al. 1998; Field et al. 1995; Harman et al. 2001; Fox et al. 2004]; and amorphous formulations [Harman et al. 2003; Ward 2003; Ward and Zedan 2007]. Much work has also been conducted on applications of slicing, and algorithmic techniques for handling awkward programming language features [Agrawal et al. 1991; Ball and Horwitz 1993; Harman and Danicic 1998] and for balancing the trade offs of speed and precision in slicing algorithms [Gupta and Soffa 1995; Mock et al. 2002; Binkley et al. 2007]. This body of knowledge has been developed over several hundred papers; interested readers may find it easier to start with one of the survey papers on the area [Binkley and Gallagher 1996; Binkley and Harman 2004; De Lucia 2001; Harman and Hierons 2001; Tip 1995; Venkatesh 1991; Xu et al. 2005; Silva 2012].

Consider the example program in Figure 1 (a), taken from [Tip 1995], that computes the product `p` and the sum `s` of integer numbers up to a limit `n`. With a slicing criterion of (line 10, {`p`}) (i.e. we are only interested in the computation of the product and its output in line 10) then the slice, illustrated in Figure 1 (b), still computes the product correctly. This is a *static* slice because it is independent of the program's inputs and computes `p` correctly for all possible executions. Alternatively, if we are interested only in the statements which have an impact on the criterion for a *specific* execution, we can compute a *dynamic* slice. The slicing criterion for static slices is extended with a third item, the inputs to the program. In Figure 1 (c) a dynamic slice is shown for the execution where the input to variable `n` is 0.

A common approach to program slicing uses reachability analysis in program dependence graphs (PDGs) [Ferrante et al. 1987]. Nodes in a PDG represent program states, with edges representing dependence. Dependence comes in two forms: the simple form

```
 1 read(n)
 2 i := 1
 3 s := 0
 4 p := 1
 5 while (i <= n)
 6   s := s + i
 7   p := p * i
 8   i := i + 1
 9 write(s)
10 write(p)
```

(a) Original program

```
 1 read(n)
 2 i := 1
 3
 4 p := 1
 5 while (i <= n)
 6
 7   p := p * i
 8   i := i + 1
 9
10 write(p)
```

(b) Static Slice for (10, p)

```
 1
 2
 3
 4 p := 1
 5
 6
 7
 8
 9
10 write(p)
```

(c) Dynamic Slice for (10, p, n = 0)

Fig. 1. A program and two slices taken from [Tip 1995]

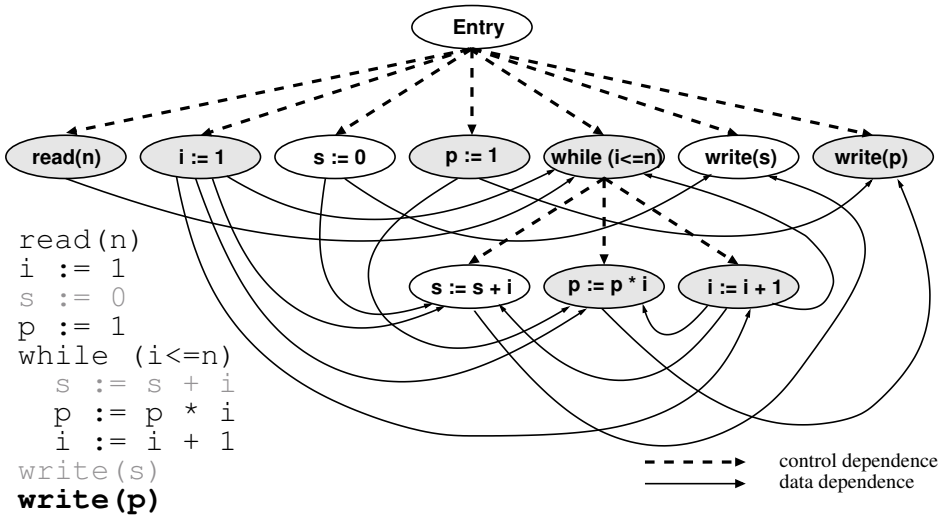

Fig. 2. The program dependence graph of the program from Figure 1. The slice for the criterion "write(p)" is highlighted in the graph and in the source text.

data dependence between statements S and S' exists if S' references a variable defined or assigned to in S; the complex form *control dependence* between statements S and S' exists if S determines whether S is executed or not. Data dependence is relatively easily calculated; as we shall later see, control dependence comes in many different forms, depending on the desired effect.

Using PDGs, static slices of programs can be computed by identifying the nodes that are reachable from the node corresponding to the criterion. The underlying assumption is that all paths through the dependence graph are *realisable*. This means that, for every path through the dependence graph a possible execution of the program exists that executes the statements corresponding to the nodes on the path in the same order as on the path. In the presence of procedures, paths are considered realisable only if they obey the calling context (i.e. called procedures always return to the correct call site). Ottenstein and Ottenstein [1984] were the first to suggest the use of PDGs to compute Weiser's slices.

An example PDG is shown in Figure 2, taken from [Tip 1995], where control dependence is drawn in dashed lines and data dependence in solid ones. In the Figure, a slice

is computed for the statement "`write(p)`". The statements "`s := 0`", "`s := s+i`", and "`write(s)`" have no direct or indirect influence on the criterion and are thus not part of the slice.

3. STATE-BASED MODELS (SBMS)

SBMs are used to model the behaviour of a wide variety of systems, such as embedded systems. They consist of a finite set of states (a non-strict subset of which are start states), a set of events (or 'inputs') and a transition function that, based on the current state and event, determines the next state (i.e. performs transitions between states). The start state indicates the state in which computation starts; transitions are then performed based on the transition function. This basic definition has many variants; for example, Moore machines [Moore 1956] extend state machines with labels on states, while Mealy machines [Mealy 1955] have labels on transitions.

Figure 3 illustrates a simple SBM with two states $S1$ and $S2$, and a labelled transition $T1$. $S1$ is a start state (indicated by the incoming edge from the filled in circle). We write $source(T1) = S1$ to indicate the source state for $T1$ and $target(T1) = S2$ for its target. Transition labels are of the form $e[g]/a$, where each part is optional: e is the event necessary to trigger a possible change of state; g is the guard (i.e. a boolean expression) that further constrains a possible change of state; and a is a sequence of actions (chiefly updates to variables in the store, or generation of events) to be executed when a change of state is about to take place. A transition is executed when its source state is the current state, its trigger event occurs and its guard is true.

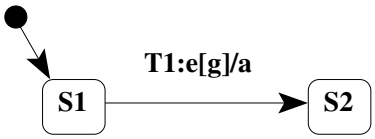

Fig. 3. A simple state machine.

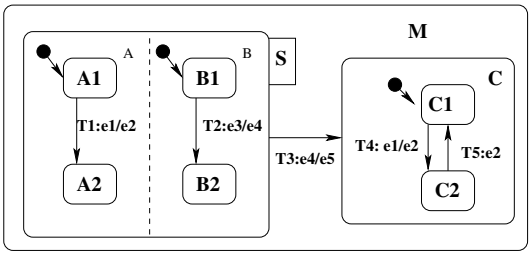

Fig. 4. A hierarchical and concurrent state machine.

SBMs can satisfy many different properties. The two most commonly references are as follows:

— **Non-determinism.** In a *deterministic* SBM, for each pair (state, event) only a single matching transition can validly be taken; in a *non-deterministic* SBM, any of a set of matching transitions can be taken.
— **Non-termination.** A SBM is non-terminating if there is a path from each state to every other state. A terminating SBM has at least one *exit* state that has no outgoing transitions.

Basic SBM languages have long been extended to augment their expressive power or to allow better structuring of SBMs. The major features are as follows:

— **Store.** SBMs can have a *store*, a set of variables (that can be of type real) which can be updated by actions. For example, in Figure 5 (via Table II) the store is $\{pin, d, w, sb, cb, p, attempts, l\}$.
— **Parameterised events.** Basic SBM events are opaque: one can determine only their 'type'. For more realistic purposes, events need to come with further information about the specific instance of the event. Parameterised events fulfil this purpose. For

Table I. Feature comparison of SBM languages used in slicing.

SBM Variant	Slicing Approaches	C[a]	H[b]	SC[c]
Extended Finite State Machines (EFSMs)	Korel et al. [2003]	×	×	×
	Androutsopoulos[†] et al. [2009]			
	Androutsopoulos et al. [2011]			
UML State Machines	Colangelo et al. [2006]	√	×	S[d]
	Lano and Kolahdouz-Rahimi [2011]	√	√	S
Timed Automata [Alur and Dill 1990]	Janowska and Janowski [2006]	√	×	S
Input/Output Symbolic Transition Systems (IOSTs) [Gaston et al. 2006]	Labbé and Gallois [2008]	√	×	S
Extended Automata	Bozga et al. [2000]	√	×	A[e]
UML Statecharts v1.4	Ojala [2007]	√	×	A
Statecharts [Harel 1987]	Fox and Luangsodsai [2005]	√	√	S
Argos [Maraninchi 1991]	Ganapathy and Ramesh [2002]	√	√	S
Requirements State Machine Language (RSML) [Leveson et al. 1994]	Heimdahl and Whalen [1997]	√	√	S
	Chan et al. [1998][*]			
Extended Hierarchical Automata (EHA)	Wang et al. [2002]	√	√	S
	Langenhove [2006]			
Rhapsody [Harel and Kugler 2004], Stateflow [Hamon 2005], UML statecharts	Guo and Roychoudhury [2008]	√	√	S

C^a Concurrency H^b Hierarchy SC^c Synchronisation or Communication
S^d Synchronous A^e Asynchronous
[*]Also, applies to Statecharts [Harel 1987].

example, in Figure 5 (via Table II) event *PIN* has the parameter p that represents the specific PIN number entered by the user.

— **Event generation.** Events can be generated by the state-machine itself in actions. In Figure 4, transition $T4$ generates event $e2$ which then triggers transition $T5$. Generated events are also known as *internal* events or *outputs*, while events that are generated by the environment are known as *external* events or *inputs*.

— **State Hierarchy.** Hierarchical states are an abstraction mechanism for hiding low-level details. *Basic states* are 'atomic', where *composite states* ('OR-states' in statecharts [Harel 1987]) contain other states. In Figure 4 $C1$ and $C2$ are basic states, while C is a composite. A *superstate* is the parent state of a *nested state* (e.g. in Figure 4 the superstate of $C1$ and $C2$ is C).

— **Concurrency and Communication.** Basic SBMs are purely sequential; concurrency constructs (known as 'AND-states' in statecharts [Harel 1987]) allows different superstates to execute independently or in parallel with each other. For example, in Figure 4, A and B are concurrent states (divided by a dashed line). Communication between concurrent SBMs is synchronous (the SBM blocks until the receiver consumes the event) or asynchronous (non-blocking).

— **Time.** Some SBM variants add features for modelling time. For example, timed automata [Alur and Dill 1990] model clocks using real-valued variables.

There are far too many SBM languages for this paper to capture; instead we concentrate on graphical SBM languages used in SBM slicing. Table I gives an overview of these languages.

4. A RUNNING EXAMPLE

We model the Automatic Teller Machine (ATM) system using state machines in two different ways and use these as running examples. This is because we want to illustrate how the differences between SBM variants affect slicing. The first example models the

Table II. The transitions of the ATM system as illustrated in Figures 5. See Table II for the transition labels.

Transition	Label
T1	Card(pin,sb,cb)/print(“Enter PIN”); attempts = 0
T2	PIN(p)[(p ! = pin) and (attempts $<$ 3)]/print(“Wrong PIN, Re-enter”); attempts = attempts+1
T3	PIN(p)[(p ! = pin) and (attempts == 3)]/ print(“Wrong PIN, Ejecting card”)
T4	PIN(p)[p==pin]/print(“Select a Language English/Spanish”)
T5	English/l=‘e’; print(“Savings/Current”)
T6	Spanish/l=‘s’; print(“Ahorros/Corriente”)
T7	Current
T8	Savings
T9	Done
T10	Done
T11	Balance[l=‘s’]/print(“Balanza=”,cb)
T12	Balance[l=‘e’]/print(“Balance=”,cb)
T13	Deposit(d)/cb=cb+d
T14	Withdrawal(w)/cb=cb-w
T15	Receipt[l=‘e’]/print(“Balance=”,cb); print(“Savings/Current”)
T16	Receipt[l=‘s’]/print(“Balanza=”,cb); print(“Ahorros/Corriente”)
T17	Withdrawal(w)/sb=sb-w
T18	Deposit(d)/sb=sb+d
T19	Balance[l=‘e’]/print(“Balance=”,sb)
T20	Balance[l=‘s’]/print(“Balanza=”,sb)
T21	Receipt[l=‘e’]/print(“Balance=”,sb); print(“Savings/Current”)
T22	Receipt[l=‘s’]/print(“Balanza=”,sb); print(“Ahorros/Corriente”)
T23	Exit/print(“Ejecting card”)

ATM using a SBM variant that has no concurrency or state hierarchy and is deterministic with a unique exit state. The second example introduces concurrency, state hierarchy and event generation. In order to be consistent, we have used a standard graphical notation, as illustrated in Figure 3 and 4.

The first example, illustrated in Figure 5, models the ATM as described by Korel et al. [2003] for EFSMs. EFSMs extend FSMs with a store. The ATM system allows a user to enter a card and a correct PIN. The user is allowed a maximum of three attempts to enter a correct PIN. The PIN is verified by matching it against a PIN that is stored on the card. Once the PIN has been verified, the user can withdraw, deposit, or check balance, on either their current or savings account. Figure 5 has parameterised events *Card(pin, sb, cb)* and *PIN(p)* (see Table II). The event *Card* has three parameters denoting information stored on the card, i.e., *pin* that represents the value of the PIN, *sb* that represents the balance of the savings account, and *cb* that represents the balance of the current account. The event *PIN* has a parameter *p* that represents the value for the PIN entered at the ATM by the user.

Figure 6 shows the second ATM variant, which is hierarchical, concurrent, and has generated events (we assume STATEMATE semantics [Harel and Naamad 1996]). It consists of the hierarchical state $DispensingMoney$ that has two sub-states, $s2$ and $s3$ and the concurrent states atm and $bank$. The atm concurrent state models the behaviour of the ATM at a higher level of abstraction than that shown in Figure 5, i.e., a user can withdraw or deposit money for a single account. Also, a variable representing the current balance of an account is not given in Figure 6 because it requires to be updated based on a parameterised event, such as $T13$ in Figure 5 and some FSM variants do not support parameterised events. The $bank$ concurrent state models the bank’s behaviour as described in [Knapp and Merz 2002]. It shows how a card and a PIN that is entered into the ATM is verified by the bank. It has two key stages of verification (modelled, by the concurrent states c and b): the bank needs to verify that

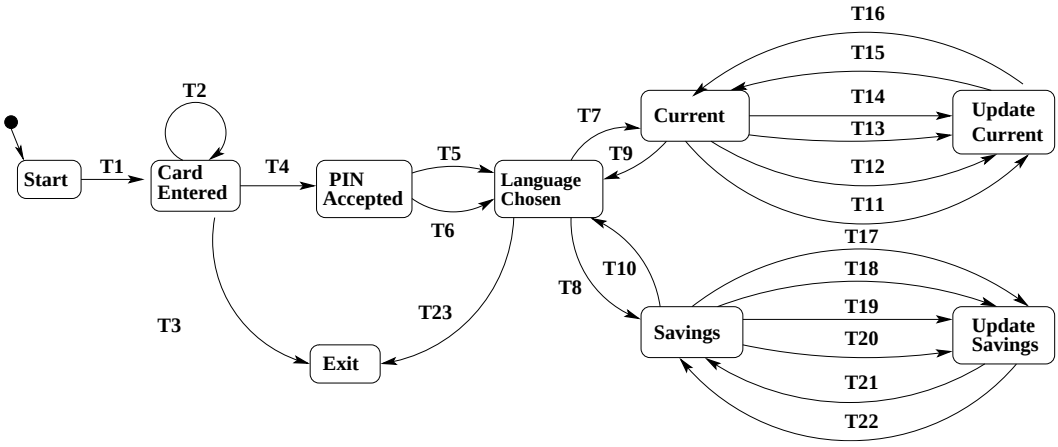

Fig. 5. The ATM system as modelled by Korel et al. [2003], © IEEE 2003 Proceedings of International Conference of Software Maintenance, for EFSMs with a unique exit state. See Table II for the transition labels.

the card is valid (i.e. it is not some arbitrary card); and the PIN entered is correct, and if not the user is given three attempts to enter a correct PIN.

5. TYPES OF SBM SLICING

We describe all of the SBM slicing approaches according to their type and produce a slice from the running example where possible. For each type of slicing, we summarise its goal and list its main applications.

5.1. Static Slicing

Most SBM slicing techniques are static [Weiser 1979], meaning that the slice considers any possible input event sequence. An *executable* slice is a subset of the original model, where elements not in the slice have been removed, that maintains the effect of the original model on the slicing criterion. A *closure* slice is given by marking elements in the slice on the original model. Slices can be *backward* or *forward*, depending on the direction in which models are traversed from the slicing criterion. Backward slicing determines all the elements in the model that could influence the slicing criterion. Forward slicing determines how modifying one part of the model will affect other parts of the model. Computing SBM slices requires some dependence analysis to determine which elements in the model depend on the slicing criterion. In program slicing, typically a data structure is used to make the dependencies for each statement in a program explicit (e.g. a PDG, see Figure 2) and slicing is defined as a reachability problem on this graph. In SBM slicing, some approaches use dependence graphs for static slicing but others compute dependencies directly on the model. While to compute other types of dependence, intermediate representations of the model are required to make relationships between elements explicit. This is the case for hierarchical and concurrent SBM variants where these additional features were introduced to make models more concise.

The algorithms in [Androutsopoulos† et al. 2009; Korel et al. 2003] for slicing EFSMs both slice with respect to a transition T and a set of variables at T. They first construct a dependence graph by using data and control dependence relations. A dependence graph is a directed graph where nodes represent transitions and edges represent data

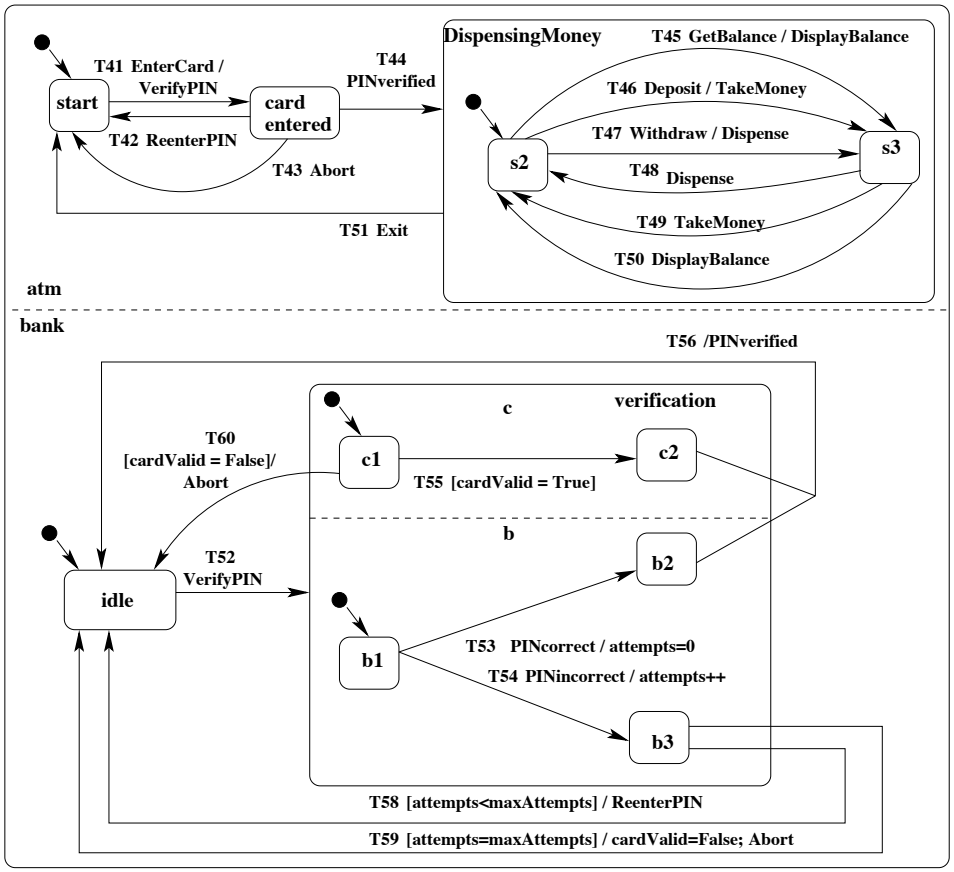

Fig. 6. The ATM system modelled by a hierarchical and concurrent state machine with generated events.

and control dependencies between transitions. Then, the algorithm starts from the node in the dependence graph representing the slicing criterion and marks all nodes (i.e. transitions) that are backward reachable from the slicing criterion in the dependence graph. Once the transitions in the slice have been marked, Androutsopoulos† et al. [2009] and Korel et al. [2003] have implemented different algorithms for automatically reducing the size of an EFSM slice, and we discuss each respectively. Note that the dependence graphs generated by [Androutsopoulos† et al. 2009] and [Korel et al. 2003] differ because they use different definitions of control dependence. Control dependence in [Korel et al. 2003] is non-termination sensitive (intermediate loops are kept in slices) and applies only to state machines with a unique "exit state" (a state with no outgoing transitions). While, control dependence in [Androutsopoulos† et al. 2009] is non-termination insensitive (intermediate loops are sliced away) and can be applied to any state machine, including those with no exit state. This means that the slicing algorithm described in [Korel et al. 2003] cannot be applied to a non-terminating EFSMs. For example, if the ATM system shown in Figure 5 had a transition whose source state is $Exit$ and target state is $Start$, then the slicing algorithm in [Korel et al. 2003] cannot be applied.

Korel et al. [2003] describe two slicing algorithms for automatically reducing the size of the EFSM slice. The first slicing algorithm produces slices that are syntax pre-

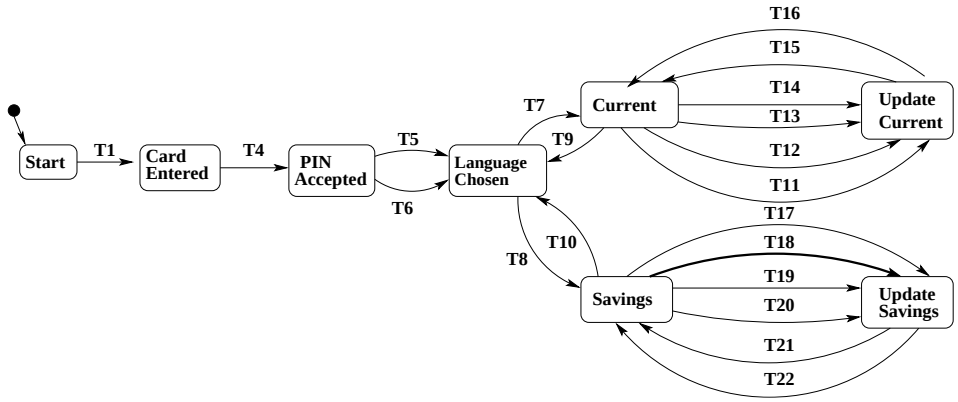

Fig. 7. The slice generated for the ATM system, shown in Figure 5, with respect to $(sb, T18)$ (highlighted) using Korel et al. [2003] first algorithm. The transition labels are given in Table II.

serving, i.e., they are executable sub-models of the original EFSMs and thus are not much smaller than the original. Consider the ATM system shown in Figure 5. The slice obtained using the first algorithm, as described in [Korel et al. 2003], with the slicing criterion $(sb, T18)$ is illustrated in Figure 7. This slice could be produced by just applying a reachability algorithm. It is not minimal as it contains more than the transitive dependencies (e.g. in the ATM the transitive dependencies with respect to $(sb, T18)$ are: $T1, T4, T8, T17, T18$), which this algorithm cannot remove without breaking the connectivity of the state machine. The second slicing algorithm is an amorphous slicing approach and is discussed in Section 5.5.

The algorithm in [Androutsopoulos† et al. 2009] anonymises all unmarked transitions i.e., they have empty labels. A slice with unmarked transitions may introduce non-determinism where none previously existed. Consider the ATM system shown in Figure 5. If the slicing criterion is $(sb, T18)$, the slice produced is shown in Figure 8, where ε represents unmarked transitions. Non-determinism is introduced at any state where there is more than one outgoing transition with an empty label because if an event occurs that does not trigger an event of a transition with a label, then any one of the transitions with the empty label can be taken.

The slicing algorithms in [Androutsopoulos† et al. 2009; Korel et al. 2003] cannot be applied to the ATM system shown in Figure 6 because the EFSMs considered don't have generated events.

Labbé and Gallois [2008] have presented polynomial algorithms for slicing Input/Output Symbolic Transition Systems (IOSTSs). The slicing criterion is a set of transitions. The algorithm is similar to the algorithms in [Androutsopoulos† et al. 2009; Korel et al. 2003] whereby a dependence graph is constructed and transitions that are backward reachable from the slicing criterion are marked. The slice produced is a closure slice, where transitions in the slice are marked, i.e., similarly to [Androutsopoulos† et al. 2009]. Figure 9 shows the slice produced when applied to Figure 5. This algorithm differs from [Androutsopoulos† et al. 2009] as it applies a different control dependence definition, one that is sensitive to non-termination[1], i.e. loops are kept in the slice as infinite execution of a loop may prevent some transitions from occuring. Also, it can be applied to communicating automata and Labbé and Gal-

[1]Labbé et al.'s definition of control dependence in [Labbe et al. 2007] differs slightly from [Labbé and Gallois 2008], so we evaluate the most recent.

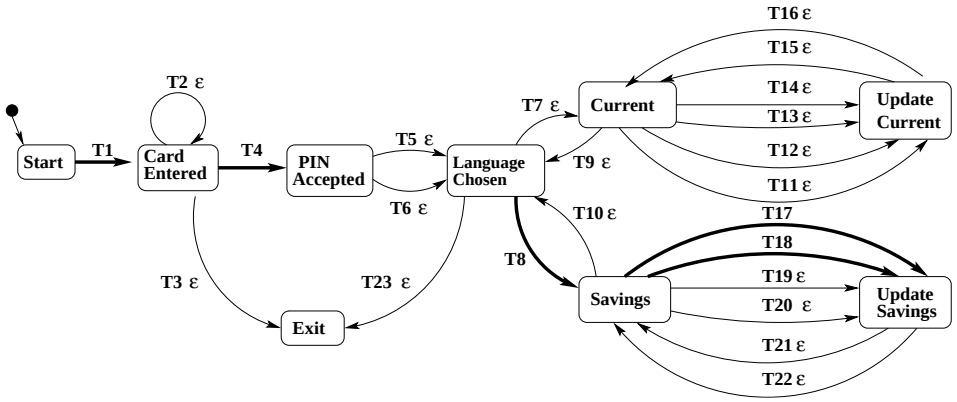

Fig. 8. The slice generated for the ATM system, shown in Figure 5, with respect to $(sb, T18)$ using the algorithm by [Androutsopoulos† et al. 2009]. The labels of marked transitions (highlighted) are given in Table II, while unmarked transition have the label ε indicating an empty label.

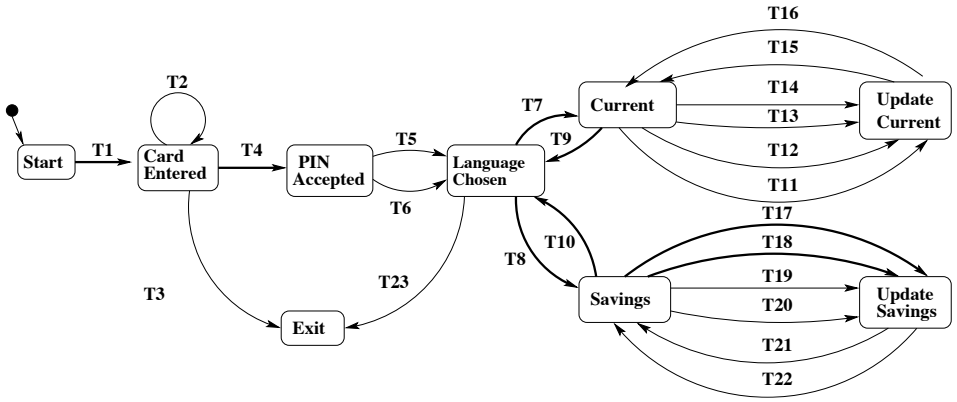

Fig. 9. The slice generated for the ATM system (shown in Figure 5) with respect to $(sb, T18)$ using Labbé and Gallois [2008]. The highlighted transitions indicate the transitions in the slice. The transition labels are given in Table II.

lois [2008] define communication dependence with respect to channels to capture the dependencies produced by communication actions. However, the algorithm cannot be applied to the ATM system in Figure 6 because communication is not described via channels. In their prototype tool, Labbé and Gallois [2008] give the option of reducing the slice further by removing transitions not in the slice and reconnecting the graph by applying two different algorithms based on τ-reduction of labelled transition systems and ε-reduction of non-deterministic finite automata (NFA) with ε-transitions. The process of ε-reduction of an NFA with n states and alphabet size p can lead to an NFA with $O(n^2p)$ transitions [Hromkovic and Schnitger 2007]. They do not give the details in [Labbé and Gallois 2008] of the algorithms or how they overcome this problem.

Fox and Luangsodsai [2005] have defined And-Or dependence graphs that are used to slice statecharts [Harel 1987]. The And-Or dependence graphs are based on dependence graphs as in [Kuck et al. 1981] but augmented to record And-Or dependencies. They consist of nodes that represent any statechart element that can be depended on

or can depend on (i.e. states, actions, events and guards), and edges that represent potential dependence. The slicing criterion is a collection of states, transitions, actions and variable names. Slicing is static and backward and it is defined as a graph reachability problem over the And-Or dependency graph with respect to the slicing criterion. Elements not in the slice are deleted. This slicing approach cannot be applied to the ATM system in Figure 5 as details of how to extend the dependence graph to include variable dependencies have not been given. Also, it cannot be applied to the ATM in Figure 6 because the approach does not yet deal with hierarchical states.

Ojala [2007] has presented a slicing approach for UML state machines (specifically UML 1.4 [OMG 2001]). The guards and actions of transitions are expressed in Jumbala [Dubrovin 2006], which is an action language for UML state machines. Actions have at most one primitive operation, i.e., an assignment, an assertion or a Jumbala "send" statement. The slicing criterion is a set of transitions in a collection of UML state machines. The slicing algorithm constructs a CFG from the UML state machines, keeping a record of the mapping between UML transitions and CFG nodes. Three types of CFG nodes are defined: BRANCH which are used to represent triggers and guards, SIMPLE and SEND, both are used to represent actions. BRANCH nodes can have more than one successor and SIMPLE and SEND have only one successor. Then, using the CFG, four types of dependencies are computed. The CFG slice is the smallest set of nodes and event parameters, including the nodes of the slicing criterion, that are closed under the four dependencies. From the CFG slice, the slice for the UML model is computed by removing all parts of the transitions in the UML model whose counterparts in the CFG are not in the slice. Also, unused parameters are replaced with a dummy value. Fox and Luangsodsai [2005] and Ojala [2007] are the only that have defined slicing approaches that can remove parts of transitions, i.e., trigger events, or guards, or actions, rather than just the actions of a transition or the entire transition (or label). These differ in the way that the dependencies are computed. Ojala [2007] define four dependence relations between transition elements while in [Fox and Luangsodsai 2005] every action depends on its trigger, guard and source state. Table V compares the slicing approaches according to the elements that they remove. This slicing approach [Ojala 2007] cannot be applied to any of the running examples because the language used for expressing the guards and actions is not Jumbala.

Lano and Kolahdouz-Rahimi [2011] define a slicing approach for a restricted subset of UML that is used for developing reactive systems. Their approach makes use of the semantic concept of path-predicates (as used in SPADE [Praxis Ltd 2008]). A predicate is assigned to each path which defines how the values of the variables at the end of the path relate to the values of the start state, over all executions of the path. Computing path predicates for state machines with loops is impractical and thus Lano and Kolahdouz-Rahimi [2011] only provide an algorithm for loop-free state machines. The slicing criterion is a tuple of variables of interest. Actions that cannot affect the values of the variables in the target state of the transition are deleted. Lano and Kolahdouz-Rahimi [2011] have also described another slicing approach that is environment-based (discussed in Section 5.6) that uses a number of algorithms. They claim that these algorithms can be used for this type of slicing too, but require the data dependencies to be recalculated because the set of states and paths may have changed. These algorithms can be re-applied until the state machine can be no longer reduced. This slicing approach cannot be applied to any of the state machines defined for the ATM system (Figure 6 and Figure 5) because they all contain loops.

Objective: To produce a slice that shows what elements in the SBM influence a given set of elements, either variables, transitions, states or actions, or some combination of these.

Applications: Model comprehension,

5.2. Proposition-based Slicing

Proposition-based slicing [Hatcliff et al. 2000] was defined for reducing the size of a program with respect to a linear temporal logic (LTL) formula in order to reduce the size of the corresponding transition system for model checking. Model checking is often very costly for large and complex programs.

We classify a SBM slicing approach as being proposition-based if its objective is to reduce the model with respect to a property ϕ (expressed as a temporal logic formula) that is to be model checked. The slicing criterion typically consists of elements from ϕ, such as, the set of states and transitions in ϕ, or the set of variables in ϕ. The slice produced must preserve the behaviour of those parts of the model that affect the truth of ϕ. The first proposition-based slicing approach [Chan et al. 1998] was defined for RSML (Requirements State Machine Language) and statechart [Harel 1987] specifications for model checking. RSML [Leveson et al. 1994] is a requirements specification language that combines a graphical notation that is based on statecharts [Harel 1987] and a tabular notation, i.e., AND/OR tables. The slicing criterion consists of the states, events, transitions or event parameters that appear in a property to be model checked. Initially the slicing criterion will be in the slice. The algorithm recursively applies the following rules until a fixed point is reached. If an event is in the slice, then so are all the transitions that generate it. If a transition is in the slice, then so are its trigger event, its source state as well as all the elements in the guarding condition. If a state is in the slice, then so are all of its transitions (both in and out), as well as its parent state. In fact, the algorithm describes a search of the dependence graph and its time complexity is linear to the size of the graph.

We manually apply this slicing algorithm to the hierarchical and concurrent statechart of the ATM system in Figure 6. Given the LTL safety property $G(\neg(\mathrm{Abort} \wedge \mathrm{PINverified}))$, which states that a card cannot be both aborted and verified by the bank, the slicing criterion consists of the events Abort and $\mathrm{PINverified}$. The slice produced is shown in Figure 10. The nested states and transitions of DispensingMoney have been removed. Given the LTL property $G(\mathrm{Withdraw} \Rightarrow F(\mathrm{TakeMoney}))$, which states that if the user asks to withdraw money, he/she will eventually take it, then the slicing criterion consists of the events Withdraw and TakeMoney. The slice will only consist of these events, as these are external events that do not influence any other element.

The slicing algorithm [Chan et al. 2001] is not minimal and may include false dependencies, i.e., elements are shown to be dependent on each other when they should not be. Not only do false dependencies increase the size of the slice but they can mislead as to which elements actually affect the slicing criterion.

Wang et al. [2002] use slicing for reducing the state space of UML statecharts when model checking. UML statecharts are translated into Extended Hierarchical Automata (EHAs) [Dong et al. 2001] and then sliced with respect to the slicing criterion, which is extracted from a given LTL_x (without the next operator) property ϕ. An EHA is composed of a set of sequential automata, which is a 4-tuple, consisting of a finite set of states, an initial state, a finite set of labels and a transition relation. The slicing criterion consists of the states and transitions described in ϕ as well as the states and transitions that generate any event found in ϕ. Four dependence relations are defined, which are able to handle hierarchy, concurrency and communication. A slice consists of sequential automata. If a state or transition in a sequential automaton is determined to be in the slice, then all of the states and transitions in this automaton are also in the slice. After the algorithm terminates, if a state is not dependent on any elements, then a sub-EHA and actions of this state will be deleted from the slice. If a transition is not dependent on any elements, its action will be deleted. This is an improvement

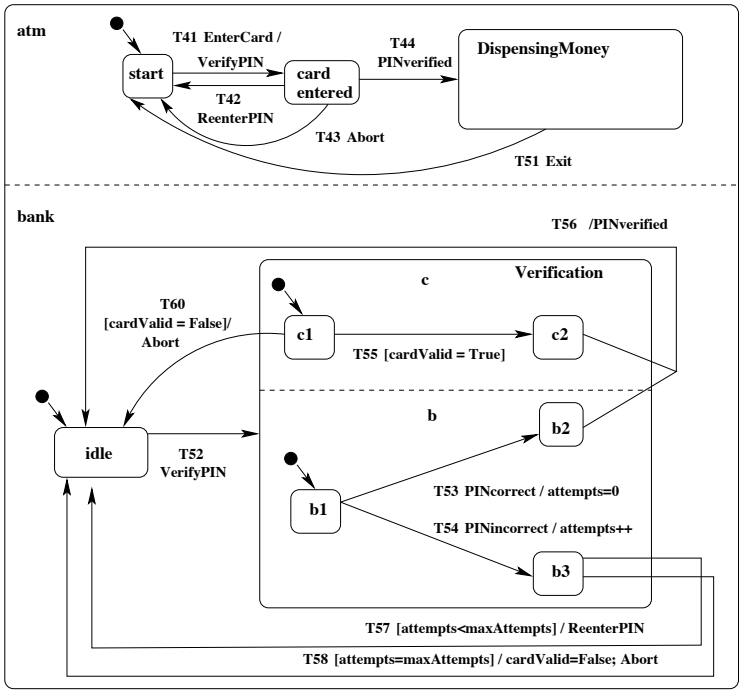

Fig. 10. The slice produced by applying the algorithm in [Chan et al. 1998] to the ATM system in Figure 6 with the slicing criterion consisting of events Abort and PINverified.

on the algorithm described by Ganapathy and Ramesh [2002] that only deletes states and the transitions associated with that state, but not parts of transitions.

Langenhove and Hoogewijs [2007] have defined two new slicing algorithms, as part of the SV$_t$L (System Verification through Logic) framework. The first algorithm is an extension of the algorithm defined in [Wang et al. 2002] for slicing a single statechart. It removes false parallel data dependencies by taking into account the execution chronology and defining a Lamport-like [Lamport 1978] happens-before relation on statecharts that follows from the internal broadcasting (synchronisation) mechanism for communication between concurrent states/transitions.

The second algorithm is a parallel algorithm for slicing a collection of statechart models. A collection of statecharts is often used when describing a system in UML, i.e., a class diagram is defined, where each class has a corresponding statechart. Figure 11 shows an example of a bank and ATM system modelled in UML as two threads of control with their classes and collection of statecharts. Slices are extracted across all statecharts in order to keep the object-oriented structure of the model. Langenhove and Hoogewijs [2007] define global dependence relations in terms of global variables and events that statechart diagrams use to communicate. The algorithm uses these relations to connect the statecharts to each other by drawing a global directed edge for each global dependence. The result is a graph-like structure, which is similar to the one in [Ganapathy and Ramesh 2002], but draws edges between statecharts rather than statechart elements. Then SV$_t$L starts running an instance of the slicing algorithm for a statechart, e.g., *BankVerifier* in Figure 11. If a global dependence edge is encountered, then a second instance of the slicing algorithm is started that runs in parallel, e.g., if there is a global dependence between *BankVerifier* and *ATM* in Fig-

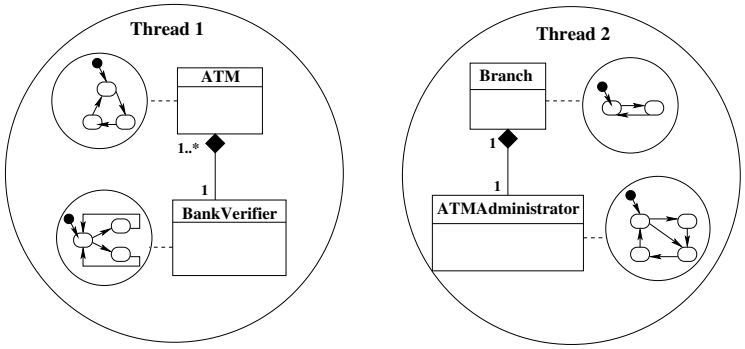

Fig. 11. An example of a multi-threaded behavioural modal of the bank and ATM system.

ure 11, then another instance of the slicing algorithm is executed. For n statecharts in the worst case SV_tL will execute n slicing algorithms in parallel. Langenhove and Hoogewijs [2007] state that the happens-before relation on a single statechart can be easily adapted to apply to a collection of statecharts. This will produce smaller slices because there will be fewer global dependence edges.

Janowska and Janowski [2006] have described a static backward algorithm for slicing timed automata with discrete data. They consider only automata with reducible control flow as defined in [Aho et al. 1986], i.e., those that have two disjoint sets of transitions, one set forms an acyclic graph, while the other consists of transitions whose target dominate their sources. A state a dominates a state b if every path from the start state to b must go through a. The algorithm, first extracts the slicing criterion, which is made of two sets, from a formula ϕ representing a given property to be verified. The first set consists of all enabling conditions and actions defining variables in ϕ. The second set consists of the states in ϕ and their immediate predecessors. Then, the algorithm computes four kinds of dependencies: data, control, clock and time. The transitive closure of the data dependence relation is computed and then the transitive closure of the union of all the other relations on states. Finally, starting from the slicing criterion, the algorithm marks all relevant elements based on the dependencies. The slice consists of marked elements. Any unmarked states, transitions or actions are deleted. We cannot apply this slicing approach to any of the ATM systems given as running examples because they are not timed and transitions are synchronised differently.

Colangelo et al. [2006] have described an approach for slicing Software Architecture (SA) models that are specified as UML state machines. A state in a UML state machine represents an architectural component, while a transition represents the communication channel between two components. Properties are described using the Property Sequence Charts (PSC) language, an extension of UML 2.0 sequence diagrams for specifying Linear-Time Temporal Logic (LTL) properties. The state machines will be translated into PROMELA (input language of the model checker SPIN) and the PSC properties into temporal logic representation for SPIN (Büchi automata) [Holzmann 1997] for model checking. The slicing criterion is a property to be model checked expressed in PSC. The slicing algorithm is based on TeSTOR (a TEst Sequence generaTOR algorithm) [Pelliccione et al. 2005]. TeSTOR is an algorithm that takes as input a state machine and scenarios and produces a set of test sequences that explore the scenarios. The authors extend TeSTOR to implement the slicing algorithm, i.e., instead of returning a set of test sequences, it returns a state machine where the parts of the model that are required to verify the given properties are marked. Their algorithm first

marks every source and target state of a message in the slicing criterion (an arrow in the PSC between two components defines a communication channel and the messages that can occur) in at least one test sequence generated by TeSTOR. Then, for all the variables of transitions that have a marked target state, the algorithm identifies paths from the initial state to all occurrences of variables that are marked. These two steps are iterated. Any unmarked states or transitions that have unmarked source or target states are deleted. The algorithm never deletes states that might break the connectivity of the state machine because any unmarked states on a path starting in the initial state and ending at a marked state are always marked. Therefore, the slice may not be minimal. This slicing approach cannot be applied to the state machines of the ATM system (Figure 6 and Figure 5) because they are not software architecture models with PSC.

Objective: The slice produced by proposition-based slicing is a subset of the state-based model that satisfies a given temporal logic formula.

Applications: Model checking

5.3. Reactive Program Slicing

Ganapathy and Ramesh [2002] devise a new notion of slicing for reactive programs because the traditional notion of slicing for programs [Weiser 1979] is unsuitable. The behaviour of reactive programs maintains an ongoing relationship with its environment, i.e., is a set of I/O sequences and therefore events are of greater interest than variables. Work on program slicing has not considered reactive programs.

Reactive program slicing [Ganapathy and Ramesh 2002] is defined for Argos specifications. Argos is a graphical language based on Boolean Mealy machines with hierarchical states and concurrent state machines used to specify synchronous reactive systems. The slicing criterion $< S, e >$ is given as the name of a state S and a generated event e. Slicing produces a state machine M' by removing zero or more states and transitions from the original machine M and the behaviour of M' up to state S is the same as the behaviour of M up to state S with respect to event e. The slicing algorithm, with respect to $< S, e >$, is a traversal algorithm that works on a graph representing the original state machine M, whose nodes correspond to the states of M and has three types of edges. A *transition edge* exists for every transition in M. A *hierarchy edge* exists between a node A and a node B if the state corresponding to A contains the state corresponding to B as a sub-state. A *trigger edge* occurs between a transition t_1 and t_2, if t_1 generates an output signal that triggers t_2. All of the states and transitions encountered during the traversal are included in the slice. The algorithm starts from S and traverses down the hierarchy edges including the states that preserve the behaviour of M according to e. Then, for the same hierarchy level as S, it traverses backwards up the transition edges and includes all the states encountered. Once all the required states at that level have been traversed, then a similar traversal occurs at the next, higher level, and so on, until it reaches the top-most level. From the top-most level, the algorithm traverses backwards along the trigger edges and includes any state that is concurrent to the states already in the slice.

Similarly to the slicing approaches given in [Chan et al. 1998; Heimdahl and Whalen 1997], transitions that may generate an event of interest are kept in the slice in [Ganapathy and Ramesh 2002]. This ensures that the connectivity of the state machine is not broken during slicing. Also, the slicing algorithm in [Ganapathy and Ramesh 2002] does not fall pray to false dependencies like in [Chan et al. 1998] because transitions in Argos do not have guards.

This slicing algorithm cannot be applied to any of the running examples because Argos programs are composed of Boolean Mealy machines in which the inputs and

outputs are pure signals (or events), i.e., transitions have no guarding conditions and there is no store.

Objective: Given a state S and a generated event e, reactive program slicing produces a state machine M' by removing zero or more states and transitions from the original state machine M, and the behaviour of M' up to state S is the same as the behaviour of M up to state S with respect to event e.

Applications: Model comprehension and model checking

5.4. Dynamic Slicing

Dynamic slicing in programs [Korel and Laski 1988] extracts slices that contain the statements in a program that influence the slicing criterion for a specific execution rather than any execution as in static slicing. It was defined for debugging programs. Figure 1 describes a dynamic slice for an example program.

Guo and Roychoudhury [2008] describe a SBM slicing approach that uses dynamic slicing of programs for debugging model-driven software. Their goal is to take errors found in the Java code and trace the error back to the corresponding part of the model. Rhapsody [Harel and Kugler 2004] or Stateflow [Hamon 2005] statecharts are used to model a system and the authors have defined a tool that automatically generates executable Java code from the models that handles hierarchy, concurrency and event generation. The code generated is tagged with corresponding statechart elements, known as model-code association tags, to ensure traceability. Their tool supports better model-code association tags than Rhapsody and Stateflow as it includes tags for event and transition firings. This leads to more accurate slices because with these tags they can track events which trigger transitions and generated events and can distinguish betweeen which transitions to keep or remove in the slice. Each statechart is translated into a single-threaded Java program. Then, subject to an error being detected, dynamic slicing, using the JSlice [Wang et al. 2008] tool, is applied to the Java code. JSlice is an open source tool that produces backward dynamic slices of sequential Java programs. Guo and Roychoudhury [2008] have previously [Wang and Roychoudhury 2004] modified JSlice to perform online compression during trace collection and they use this version to produce the code slices. The slicing criterion, at the model level, is the last state visited by an object where the error occured. Since slicing is performed at the code level, the slicing criterion is the last state entry point in the code where the error occurred. For dynamic slicing, the inputs that reveal the error are also required as part of the slicing criterion. These are obtained from test cases. In their experiments at least five test cases are chosen for each buggy version of code, and the input for each test case that reveals an error will become a different slicing criterion. The slice produced by JSlice is mapped back to the statechart model using the model-code associations and represented as a model-level bug report. The model-level bug report can then be further processed to reflect the hierarchical and concurrent structure of statecharts.

This approach can be generalised to be used with other types of program slicing and there is around 30 years of work on program slicing and some existing tools, including a commercial tool [Grammatech Inc. 2002]. It avoids developing new slicing algorithms at the model level. However, the main effort of this approach lies in defining mappings between the statechart language and the program language. Defining the mapping of the program slice back into the model is particularly hard in the case of hierarchical and concurrent statecharts if the structure of the original state machine is to be reflected in the slice. Also, the statecharts must be completely specified such that the generated code is executable. Finally, in order to provide confidence in this approach and depending on the application of slicing, the translations between the model and

code level should be verified. Guo and Roychoudhury [2008] have not discussed the correctness of their translations.

Objective: For a given program P, input I, line of code l and set of variables V, dynamic slicing finds what statements or statement instances of P affect the values of the variables V at l in the execution trace corresponding to I.

Applications: Debugging

5.5. Amorphous Slicing

Amorphous slices in programs [Harman and Danicic 1997] are produced by using some program transformations to simplify the program while still preserving its semantics with respect to the slicing criterion. The slices produced are no longer syntax preserving, i.e., subsets of the original programs, and are often much smaller than slices produced using syntax preserving slicing approaches.

Similarly, amorphous slicing for models produces slices that are not strict subsets of the original models and are thus not syntax-preserving. The challenge lies in how to remove the elements not in the slice and re-connect the state machine while preserving the semantics with respect to its slicing criterion. The task of re-connecting the state machine can lead to slices with different semantics than in the original (with respect to the traditional notion of slicing [Weiser 1979]) by possibly introducing additional behaviour (by merging states some transitions become self transitions that means they can be executed more often than in the original machine) and non-determinism. Therefore, for amorphous slicing of SBMs a weaker notion of slicing is defined. For certain applications, such as for model comprehension, these slices are desirable as they are much smaller than slices produced using traditional static slicing algorithms and thus easier to understand and analyse.

The second slicing algorithm described by Korel et al. [2003] produces an amorphous slice for EFSMs as it is not a strict subset of the original EFSM. It constructs a dependence graph by using data and control dependence relations. Then, starting from the node in the dependence graph representing the slicing criterion, which is a transition and its variables, the algorithm marks all backwardly reachable transitions in the dependence graph. The algorithm applies two reduction rules for merging states and deleting unmarked transitions. These rules are not general enough to cover all possible cases, i.e., for differently structured state machines these rules might not be very effective and slices might contain some irrelevant elements (unmarked transitions). Also, by merging states, the slice does not behave in the same way as the original on event sequences that stutter. A stuttering event sequence is a sequence of events whereby not all events trigger transitions. If an event does not trigger a transition, the state machine remains in the same state. Korel et al. [2003] address this problem by defining a new notion of correctness taking into consideration stuttering event sequences.

Consider the ATM shown in Figure 5. The slice obtained using Korel et al.'s second algorithm with the slicing criterion $(sb, T18)$ is shown in Figure 12. The transitions that have been marked from the dependence analysis are: $T1, T4, T8, T17, T18$. However, the slice includes $T10$ as this is required to ensure that $T17$ and $T18$ can be re-executed. For the stuttering event sequence: $T1, T4, T6, T8, T18, T17, T17, T18$, the slice and the original will not behave in the same way according to the traditional notion of correctness. Compared to the slice generated by Korel et al.'s first algorithm with respect to the same slicing criterion $(sb, T18)$ (Figure 7) and also other static slicing algorithms (e.g., see Figure 8), it is much easier to see in the slice (Figure 5) how the transitions $T1, T4, T17$ interact with the slicing criterion.

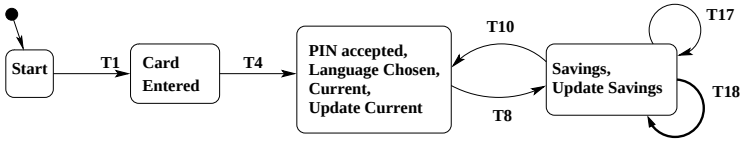

Fig. 12. The slice generated for the ATM system, shown in Figure 5, with respect to $(sb, T18)$ using Korel et al. [2003] second algorithm. The transitions are labelled as described in Table II.

Objective: Amorphous slicing for SBM aims to produce minimal slices (slices containing only elements that are identified to be either control or data dependent) while not preserving the syntax and preserving a weaker notion of correctness with respect to the transition and variables of interest.

Applications: Model comprehension

5.6. Environment-based Slicing

Environment-based slicing (a.k.a. event-based slicing) has only been defined for models [Androutsopoulos et al. 2011; Lano and Kolahdouz-Rahimi 2011], in particular for EFSMs and a restricted subset of UML statecharts. Its purpose is to facilitate model development by specialising models for a specific operating environment. Its applications include specification reuse and property verification.

The slice produced, using environment-based slicing, is a model projection with respect to a set of events I, which is the slicing criterion, that cannot occur in the new environment. Androutsopoulos et al. [2011] define four algorithms, each aiming to reduce the slice further when applied. The first algorithm deletes all transitions whose trigger event corresponds to the events in I. Then it removes all states and transitions that are no longer reachable. The second algorithm further reduces the slice by replacing a constant-value variable by its value on all remaining transitions. This can lead to guards being updated and simplified to False, in which case the corresponding transition (and possibly its target state) can be removed. The final two algorithms merge groups of states that have identical semantics. The first merging algorithm is an extension of an algorithm [Ilie and Yu 2003] for merging R-equivalent finite state automata. States $s1$ and state $s2$ are R-equivalent if, the outgoing transitions from $s1$ and $s2$ are identical in their events, guards and actions and have identical target states after the merge. The second merging algorithm merges a group of states its size is greater that two, all transitions in the group have no actions and the set of triggering events on transitions within the group are disjoint from the set of triggering events of transitions exiting the group. This algorithm results in greater reduction, however, it only preserves the weaker semantic requirement, i.e., behaviour is preserved only for the stutter free event sequences (every event in the sequence triggers a transition in the model) that exclude events in I.

[Lano and Kolahdouz-Rahimi 2011] adopt this approach for slicing reactive systems that are modeled as hierarchical and concurrent state machines, whereby the communication dependencies between the two communicating state machines form an acyclic directed graph. The core algorithm is the same as in [Androutsopoulos et al. 2011], however they apply it to each individual state machines within a hierarchy of communicating state machines, which leads to simplifying both the subordinate and superordinate state machine in the hierarchy. As in [Androutsopoulos et al. 2011] the systems are deterministic.

Consider the ATM system illustrated in Figure 5. Assume that this system is to be reused in an English speaking country only, i.e., the event *Spanish* never occurs. Environment-based slicing [Androutsopoulos et al. 2011] can be used with respect to

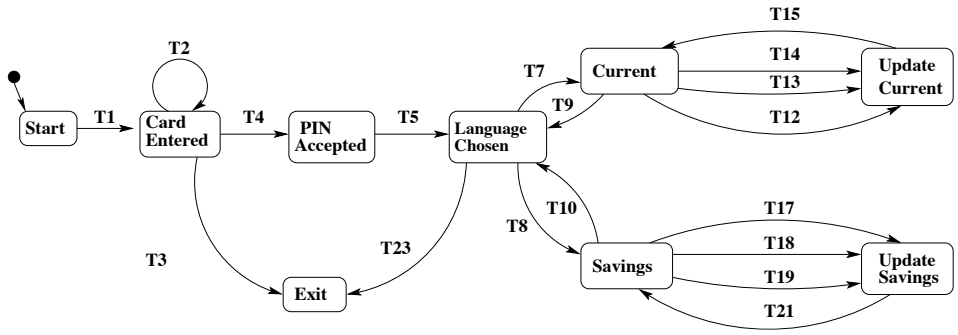

Fig. 13. The slice generated for the ATM system, shown in Figure 5, with respect to {Spanish} using environment-based slicing algorithm [Androutsopoulos et al. 2011] or [Lano and Kolahdouz-Rahimi 2011]. The transitions are labelled as described in Table II.

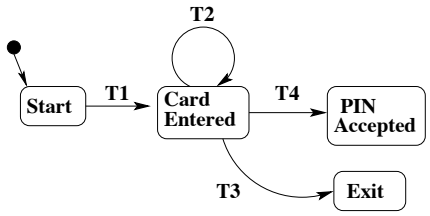

Fig. 14. The slice generated for the ATM system, shown in Figure 5, with respect to {Spanish,English} using environment-based slicing algorithm [Androutsopoulos et al. 2011]. The transitions are labelled as described in Table II.

the slicing criterion $\mathcal{I} = \{\text{Spanish}\}$ to produce an EFSM that is semantically indistinguishable to the original on all event sequences excluding the event *Spanish*. Transition $T6$ is removed first by the first algorithm. This leads to the guard of transition $T11$ always being False, so $T11$ can be deleted too. Similarly, $T16, T20, T22$ are also deleted. Applying the algorithm in [Lano and Kolahdouz-Rahimi 2011] to Figure 5 with respect to the ignore set *Spanish* produces the same slice as [Androutsopoulos et al. 2011], illustrated in Figure 13. However, the algorithm in [Lano and Kolahdouz-Rahimi 2011] cannot be applied to the hierarchical and concurrent statechart of the ATM system illustrated in Figure 6 because the communication between the concurrent state machines *atm* and *bank* is cyclic.

Although environment-based slicing is suited to SBMs as it considers the environment, the algorithm is based on reachability and often produces slices that are not reusable. For example, if the ATM system shown in Figure 5 was to be reused in an environment where the user was not going to be given the choice of language (i.e. the events *English* and *Spanish* could not occur), then applying the environment-based slicing will produce the slice as shown in Figure 14. The user should be able to deposit or withdraw from the savings or checking account, even if he or she was not given a choice of language (i.e. the slice could merge the states PIN Accepted and Language Chosen).

[Lano and Kolahdouz-Rahimi 2011] also define output event-based slicing, which is similar to environment-based slicing, except that the ignore set represents generated events (found in the actions of transitions) that cannot happen rather than input events from the environment. It uses the same algorithm as that for environment-based slicing. The applications of output slicing include refactoring and model compre-

hension (in order to view the state machines modes and effect on one group of output devices). Output event-based slicing cannot be applied to any of the ATM state machines defined as our running examples. This is because the ATM state machine shown in Figure 5 does not have any generated events, while the concurrent state machine in Figure 6 allows for cyclic communication.

Objective: Given, as a slicing criterion, the set of events I that can never occur in the new environment (as inputs or outputs), environment-based slicing finds a reduced EFSM that behaves semantically indistinguishable to the original for all possible sequences of events that exclude the events in I.

Applications: Specification reuse, model checking, refactoring, and model comprehension.

5.7. Conditioned Slicing

Conditioned slicing for programs [Canfora et al. 1998] adds a condition to the traditional static slicing criterion that captures the set of initial program states. There have been two SBM slicing approaches that could be considered as being analogous to condition slicing for programs. The first was defined for RSML specifications [Heimdahl and Whalen 1997; Heimdahl et al. 1998] to aid model comprehension and the second was defined for asynchronous extended automata [Bozga et al. 2000] for improving test case generation. We discuss each in turn.

The SBM slicing approach presented in [Heimdahl and Whalen 1997; Heimdahl et al. 1998] first reduces the RSML specification based on a specific scenario of interest (a domain restriction), which is a form of conditioned slicing. It removes all behaviours that are not possible when the operating conditions defining the reduction scenario are satisfied. A reduction scenario is an AND/OR table and it is used to mark the infeasible columns in each AND/OR table in the specification. An infeasible column is one that contains a truth value that contradicts the scenario. A collection of decision procedures have been implemented for determining whether the predicates over enumerated variables and over states in a column contradict a scenario. After all of the infeasible columns have been marked, they are removed as well as any rows that remain with only "don't care" values. Finally, tables that are left without any columns are removed, as these constitute transitions with unsatisfiable guarding conditions.

Then, in [Heimdahl and Whalen 1997; Heimdahl et al. 1998], static and backward slicing based on data and control dependence is applied to the remaining specification in order to extract the parts effecting selected variables and transitions of interest. Data and control dependence are different but are used together to compute the slice. Data dependence is computed with respect to a transition or variable. It is given as a data flow relation between elements x and y defines that y is required for evaluating x. The algorithm traverses the data dependence graph that is produced using the data flow relation, and marks all elements that directly or indirectly affect the truth value of the guarding transition. Unmarked elements are removed. Control dependence is computed with respect to a transition t with event e. It determines all transitions with event e as an action. The algorithm repeatedly applies the control flow relation for all the transitions that have been added to the slice, until transitions are reached that are triggered by external events. This is similar to the first rule defined for the proposition-based slicing approach in [Chan et al. 1998]. In both of these approaches external events have no dependencies.

This slicing approach cannot be applied to the ATM system state machines (Figure 6 and Figure 5) because they don't have AND/OR tables.

The conditioned slicing approach defined by Bozga et al. [2000] is given with respect to the following slicing criterion: a test purpose and a set of feeds. A *test purpose* de-

scribes a pattern of interaction between the user and the implementation under test (IUT). It is expressed as an acyclic finite state automaton, with inputs and outputs corresponding to inputs and outputs in the implementation. *Feeds* are a set of constrained signal inputs that the tester provides to the IUT during a test. These constrained signal inputs are analogous to adding conditions to the set of initial program states when slicing programs.

The slicing approach [Bozga et al. 2000] consists of three algorithms that are applied iteratively (in any order) until there are no more reductions possible. The first reduces the processes in the extended automata to the sets of states and transitions that can be reached, given the set of feeds, i.e., the algorithm performs reachability analysis. The second algorithm computes the set of relevant variables with respect to test purpose outputs in each state . A variable is relevant at a state if at that state its value could be used to compute the parameter value of some signal output occurring in the test purpose. Variables are used only in external outputs that are referred to in the test purpose or in assignments to relevant variables. The algorithm computes the relevant variables for all processes in a backward manner on the control graphs. The variables that are irrelevant are replaced by the symbol $\top$. Transitions that have definitions assigning irrelevant variants are relabeled as silent transitions. This is similar to anonymising transitions in the static slicing approach in [Androutsopoulos† et al. 2009]. Since the asynchronous extended automata is used as an intermediate program representation (in the IF [Bozga et al. 1999] framework for applying static analysis techniques), removing transitions and re-wiring the graph is not an issue because the automata will be translated, for example, into the input language of a model checker where the variables and transitions will be removed. The third algorithm uses constraints on the feeds and the inputs of the test purpose in order to simplify the specification. These constraints are first added to possible matching inputs and then propagated in the specification via some intra/interproces data flow analysis algorithms. Then, a conservative approximation of the set of possible values is computed for each control state and used to evaluate the guarding conditions of transitions. Any transition guard that can never be triggered is deleted. This slicing approach is not minimal, in that slices could be reduced further. Bozga et al. [2003] define dependence relations for specific specifications, between values of timers, for the Ariane-5 Flight program suggesting that there is possible scope for further reduction.

Objective: By adding a condition on the slicing criterion (results in constraining the input), conditioned slicing finds a reduced SBM that contains all the model elements that influence a set of variables and transitions of interest.

Applications: Model comprehension and testing

5.8. Comparison of SBM Slicing Approaches

We compare the slicing approaches (first column), in Table III, by specifying the following:

— The *Type* (second column) of slicing as described in Section 5 (S = Static, P = Proposition-based, R = Reactive program, D = Dynamic, A = Amorphous, E = Environment-based, C = Conditioned).
— The *Direction* (third column) of traversal to produce the slice , that is, backwards (B) or forwards (F).
— Whether slices are *Executable* (E) or *Closure* (C) (fourth column).
— What *Dependence* (fifth column) relations are supported. Most relations used for SBM slicing are for computing data (D) and control (C) dependence, a few for computing interference (I) dependence for inter-chart slicing. Also, there are other (O) dependence relations defined for specific state machine languages. Janowska and Janowski

Table III. Comparison of SBM slicing approaches.

Approach	**T.**[a]	**Dir.**[b]	**E./C.**[c]	**Dep.**[d]	**Syn. /Sem.**[e]	**Int.**[f]
Korel et al. [2003]	S	B	E	D,C	Y/Y	No
Fox and Luangsodsai [2005]	S	B	E	D	Y/Y	Yes
Ojala [2007]	S	B	E	D,C,I	Y/Y	Yes
Labbé and Gallois [2008]	S	B	C	D,C,I	Y/Y	Yes
Androutsopoulos† et al. [2009]	S	B	C	D,C	Y/Y	No
Lano and Kolahdouz-Rahimi [2011]	S	B	E	D	Y/Y	Yes
Chan et al. [1998]	P	B	E	D	Y/Y	Yes
Wang et al. [2002]	P	B	E	D,C,I	Y/Y	Yes
Langenhove [2006]	P	B	E	D,C,I	Y/Y	Yes
Janowska and Janowski [2006]	P	B	E	D,C,O	Y/Y	Yes
Colangelo et al. [2006]	P	B	E	-	Y/Y	Yes
Ganapathy and Ramesh [2002]	R	B	E	-	Y/Y	Yes
Guo and Roychoudhury [2008]	D	B	E	D,C	Y/Y	Yes
Korel et al. [2003]	A	B	E	D,C	N/Y	No
Androutsopoulos et al. [2011]	E	F	E	-	Y/Y	No
Lano and Kolahdouz-Rahimi [2011]	E	F	E	-	Y/Y	Yes
Heimdahl and Whalen [1997]	C	B	E	D,C	Y/Y	Yes
Bozga et al. [2000]	C	B	E	D	Y/Y	Yes

T.[a] Type of slicing (S = Static, P = Proposition-based, R = Reactive program, D = Dynamic, A= Amorphous, E = Environment-based, C = Conditioned)
Dir.[b] Direction of slicing E./C.[c] Executable or closure slice Dep.[d] Dependencies Syn./Sem.[e] Syntax/Semantics preserving
Int.[f] Interchart slicing

[2006] define clock and time dependence specifically for timed automata. For the approaches that do not compute dependencies by defining dependence relations explicitly, we simply mark the column with a $-$.

— Whether the slices are *Syntax/Semantics preserving* (sixth column). We use Y (yes) or N (no) to indicate whether the slice produced is a projection of the program syntax (that is, the slice is sub-model of the original). We use the same notation to indicate whether the slice is a projection of the original semantics.

— Whether *Inter-chart* slicing (seventh column) is supported by the approach.

6. THE SLICING CRITERION AND APPLICATIONS OF SLICING SBMS

As in program slicing [Harman et al. 1996; Silva 2012], the slicing criterion differs depending on the type of SBM slicing adopted. The SBM language variant also plays a role in the choice of slicing criterion. For example, when slicing EFSMs Korel et al. [2003] and Androutsopoulos† et al. [2009] choose a transition and its variables as a slicing criterion because all of the information is contained on transitions in EFSMs (i.e. trigger events, guards and actions) and none on states. If a specific state was chosen, there could be many transitions that lead to that state, and thus all of these will have to be considered as part of the slicing criterion. Conversely, in EHA actions (variable updates and event generation) occur at the states. Therefore, when slicing EHA, Wang et al. [2002] choose a set of states and transitions as a slicing criterion.

Another key factor that affects the choice of slicing criterion is the application of slicing. There are various applications of SBM slicing and we have broadly categorised these into: (1) model comprehension, (2) model checking, (3) testing (4) debugging, and (5) reuse. For example, when slicing for the purpose of model checking, typically the slicing criterion consists of elements mentioned in the properties to be verified. While, when slicing for the purpose of model comprehension, the slicing criterion is typically

Table IV. Approaches for slicing SBMs and their applications.

Model comprehension	SBM variant	Slicing criterion
Heimdahl and Whalen [1997]	RSML	A transition or variable
Korel et al. [2003]	EFSMs	A transition and its variables
Fox and Luangsodsai [2005]	Statecharts	Collection of states, transitions, actions, variable names
Labbé and Gallois [2008]	IOSTs	Set of transitions
Androutsopoulos† et al. [2009]	EFSMs	A transition and its variables
Ganapathy and Ramesh [2002]	Argos, Lustre	A state and output signal (generated event)
Lano and Kolahdouz-Rahimi [2011]	restricted UML state machines	Tuple of variables
Model checking		
Chan et al. [1998]	RSML	States, events, transitions, or inputs in property
Wang et al. [2002]	EHAs	States and transitions in property
Langenhove [2006]	EHAs	States and transitions in property
Colangelo et al. [2006]	State machines	Property sequence chart (events)
Janowska and Janowski [2006]	Timed automata	A set of variables and states in property
Ojala [2007]	State machines	Set of transitions
Testing		
Bozga et al. [2000]	Extended automata	Test purpose (acyclic finite automata) and a set of feeds (constrained inputs)
Debugging		
Guo and Roychoudhury [2008]	Java (map to statecharts)	Last state visited by an object when error occured
Reuse		
Androutsopoulos et al. [2011]	EFSMs	Set of events to ignore
Lano and Kolahdouz-Rahimi [2011]	Restricted UML state machines	Set of events to ignore

a transition or set of transitions and their variables. Table IV lists the slicing criteria and applications for all SBM slicing approaches. In the following sections we discuss each group of applications in more detail.

6.1. Model Comprehension

Some SBM slicing approaches were developed for helping with model comprehension, analysis or review. Typically, the slicing criterion of such approaches is a transition (or set of transitions) and its variables, and sometimes states, if variables are updated on states rather than transitions. The slice aims to reduce the size of the model to include only transitions (or states) that affect the slicing criterion.

The first application of SBM slicing was for helping manual review of system requirements of large RSML specifications [Heimdahl and Whalen 1997; Heimdahl et al. 1998]. Heimdahl et al. [1998] evaluated the effectiveness of slicing on TCAS II models [Heimdahl et al. 1996], a collection of airborne devices that provide collision avoidance protection for commercial aircraft. It consists of more than 300 states and 650 transitions. Heimdahl et al. [1998] found that slicing reduced the specification, by removing states and transitions, from 68% to 90%.

The slicing algorithms described in [Korel et al. 2003] and [Androutsopoulos† et al. 2009] are used to reduce the size of EFSM specifications in order to enhance model comprehension. Empirical results, given in [Androutsopoulos* et al. 2009], show that the smallest average backward slice size for all possible transitions over 10 EFSM

models, including an industrial model, is 38.42%. This result is comparable to the typical backward slice size of a program, which may be one third of the original program [Binkley and Harman 2003]. Note that a slice according to Androutsopoulos† et al. [2009] consists of marked and unmarked transitions and its size, in terms of number of transitions and number of states, is not reduced. Korel et al. [2003] did not explicitly describe a set of examples and their slices, but claim that experience with their tool showed a reduction of 55%-80% of model size when applying the amorphous slicing algorithm to several EFSM models. Other slicing approaches for enhancing model comprehension are described in [Fox and Luangsodsai 2005; Labbé and Gallois 2008]. The slices produced are sub-model's of the original. Neither provide data about the size of the slices.

Ganapathy and Ramesh [2002] have described an algorithm for slicing Argos specifications that can help with analysis, debugging and verification. They show that for any input sequence, the behaviour of the slice up to state S is the same as the behaviour of the original up to state S as far as the event b is concerned (where $< S, b >$ is the slicing criterion). The algorithm has been run on several example systems, including case studies like the digital watch example, as well as randomly generated Argos programs with large number of states and trigger edges to determine the time complexity. The slicing criteria were chosen randomly. The algorithm was run on each input several times and the average time was taken. For systems of average size (ranging from 100 states to 2000 and trigger edges ranging from 1 to 79), the average system time was negligible (0.01 seconds).

Lano and Kolahdouz-Rahimi [2011] describe an algorithm for slicing a restricted subset of UML state machines. The correctness of their slicing technique has been formally shown. Also, they experimentally evaluate the efficiency of their algorithm by applying it to slice five concurrent state machines, each composed of multiple copies of a component with three states. The first state machine that is sliced has 3 states and 3 transitions, the second has 9 states and 18 transitions, while the last state machine has 243 states and 1215 transitions. The execution time for the smaller state machines were reasonable, e.g., 0 ms for the first state machine and 20 ms for the second. While the execution time for the fourth state machine was 64348 ms and for the final state machine it produced an out-of-memory error.

6.2. Model Checking

Model checking consists of representing a system as a finite model in an appropriate logic and automatically checking whether the model satisfies some desired properties. If the model does not satisfy a property, a counter-example is produced, i.e., a trace that outlines the system behaviour that led to that contradiction. The properties to be verified are expressed as either temporal logic formulae or as automata. The system model is expressed as a transition system. Three types of transition systems are typically used [Muller-Olm et al. 1999]: Kripke structures, whose nodes are annotated with atomic propositions; labelled transition systems (LTS) whose arcs are annotated by actions; and Kripke transition systems that combine Kripke structures and LTS.

Model checking suffers from the state space explosion problem [Clarke et al. 1999]. This is because the state space of a system can be very large, making model checking infeasible because it is impossible to explore the entire state space with limited resources of time and memory. There are several approaches, including slicing, to handle this problem. Slicing can be applied both at the level where the system model is expressed in the input language of a model checker (a model checker is a tool used for model checking), as well as at the state machine specification level (in integrated formal methods), before the specification is translated into the input language of the model checker for verification.

At the level of the input language of the model checker, slicing techniques apply either on the input language itself or on the underlying SBM. Cone of influence [Clarke et al. 1999] is a technique for reducing the size of the underlying SBM by removing variables that do not influence the variables in the specification. This technique only focuses on variables and is similar to slicing after data dependence. Chan et al. [1998] point out that carrying out dependence analysis on the underlying SBM of the model checker, rather than at the state machine specification level, may not be as effective. For example, an event parameter would appear to depend on every event. This false dependency would not occur at the state machine specification level.

Millett and Teitelbaum [1998] and Millett and Teitelbaum [1999] have described an approach for slicing PROMELA, the input language for the SPIN [Holzmann 1997] model checker. PROMELA allows for non-determinism and is concurrent, where communication can be both synchronous or asynchronous. Slicing PROMELA consists of first producing a CFG, which is a directed graph with a set of nodes representing statements and edges representing control flow, and a PDG. Millett and Teitelbaum [1998] extend the features of the CFG and PDG with additional nodes and edges for handling PROMELA's concurrent and non-determinism constructs, while keeping the reachability algorithm as used by CodeSurfer [Grammatech Inc. 2002] and the Wisconsin tool [Horwitz et al. 2000] (both used for program slicing) the same. Since slicing is applied at the CFG of the input language and not on the underlying SBM model, this approach is comparable to program slicing.

We focus on slicing techniques at the state machine specification level. These techniques address the state space explosion problem by extracting a smaller state machine from the original that preserves the behaviour of those parts of the model that affect the truth of a given property. The slicing criterion is typically elements of a property to be model checked, such as states, transitions, events or variables. Ideally, for each slicing approach, the equivalence of the original and sliced state machine with respect to a property, needs to be formally shown.

Chan et al. [1998] have defined an algorithm for slicing RSML models for model checking. They have experimentally evaluated their slicing approach in [Chan et al. 2001] on two models, the TCAS II model [Heimdahl and Leveson 1995] and Boeing EPD (Electrical Power Distribution) case study [Nobe and Bingle 1998]. Results show that applying slicing to TCAS II reduced the Boolean state variables by half for four of the five properties. Chan et al. [1998] encode each RSML variable as a set of Boolean variables, however, typical RSML models have variables of many types, not just Booleans. Only one property required additional optimisations in order for model checking to be feasible. The reduction owing to slicing of the Boeing EPD case study was moderate because the components were more interdependent, i.e., the Boolean state variables were reduced by 30% for three properties and there were no slices for two of its properties because these depended on the entire model.

Wang et al. [2002] and Langenhove [2006] both have presented approaches for slicing Extended Hierarchical Automata (EHA) for reducing the complexity of verifying UML statechart models. A property ϕ to be model checked is given as a Linear-Time Temporal Logic (LTL) [Clarke et al. 1999] formula. Wang et al. [2002] show that slicing with respect to the slicing criterion, which consists of the states and transitions in a property ϕ, extracts a smaller EHA which is ϕ-stuttering equivalent to the original EHA. Stuttering [Lamport 1983] refers to the occurrence of repeated states (with identical labels) along a path in a Kripke structure. According to Lamport a concurrent specification should be invariant to stuttering. ϕ-stuttering equivalence means that on the property ϕ, two Kripke structures have equivalent behaviour and are invariant under stuttering. Langenhove [2006] have shown that a property is satisfied by the sliced model if and only if it is satisfied by the original model.

Ojala [2007] has described a slicing algorithm for UML statecharts for reducing the state space for model checking. A proof of correctness of slicing with respect to a formula to be model checked has not been given nor any experimental results.

Colangelo et al. [2006] have described an approach for slicing SA models, in order to handle the state space explosion problem when model checking. SA models are specified as state machines and the LTL properties to be model checked are expressed using Property Sequence Charts. They have applied their approach to a naval communication environment. The benefits of slicing is that properties that could not be model checked on the original model, because the model checker ran out of memory, could be model checked on the reduced model. No proof of correctness for their slicing approach has been given.

Janowska and Janowski [2006] have presented a slicing approach for a set of timed automata with discrete data for handling the state space explosion problem when model checking. They show that two models (the original and the slice) are equivalent with respect to CTL_{-X^*} [Clarke and Emerson 1982] formulas if there exists a bisimulation between the states of the two structures.

6.3. Testing

Slicing can be used to simplify specifications in order to help with testing. Bozga et al. [2000] have presented a slicing approach for improving automatic test case generation, in particular of conformance test cases for telecommunication protocols. Conformance testing is a black-box testing method that aims to validate that the implementations of systems conform to their specifications. Their testing approach is based on on-the-fly model checking and test cases are generated by exploring a synchronous product of the specification and some test purpose (see Section 5.7 for definition). Both specification and test purposes are described as labelled transition systems. This product can lead to the state space explosion problem arising. Bozga et al. [2000] deal with this problem by representing the specification and test purpose at a higher level, i.e,. as asynchronous extended automata and acyclic finite state automata respectively, and applying slicing before generating test cases. The slicing criterion is a test purpose and a set of feeds (see Section 5.7 for definition).

Bozga et al. [2000] have experimentally evaluated two of the three slicing techniques on a telecommunications protocol that consists of 1075 states, 1291 transitions and 134 variables. The first slicing technique can reduce the specification by removing states and transitions (and actions on transitions) by up to 80% if a suitable set of feeds for each test purpose is chosen. The smallest set of feeds covering the test purpose is not necessarily the most suitable as it is often too restrictive. They start from the smallest and iteratively add other input to the feeds until the model becomes large enough to cover the test purpose behaviour. The second slicing technique reduces the number of variables by up to 40%. They also applied their slicing techniques to a medium access control protocol for wireless ATM as well as the Ariane-5 flight program. For the protocol, they focus on verification rather than testing and found that without slicing they were not able to prove any properties because of memory limitations. The Ariane-5 flight program also benefited from slicing, as processes not involved in the verification or test generation were removed. They do not provide any experimental results for the third slicing technique as it was still under development. Bozga et al. [2000] also define notions of correctness in terms of bisimulation for each of their slicing techniques but do not provide any proofs.

6.4. Debugging

Guo and Roychoudhury [2008] have described a slicing approach used for debugging Statecharts. Their approach translates buggy statecharts into Java programs and ap-

plies dynamic slicing at the program level. The slices are then translated back into statecharts which are used to produce bug reports. They report on experiments using several buggy versions of four Statechart models (a total of nineteen buggy programs). For each buggy version, the slicing criterion is set based on the observable error and the inputs which cause the error obtained using at least five test cases. The average over all the test cases for that buggy version are computed. The results show that the size of the slices at the model level is 27% to 47%, while at the program level is 17% to 30%. For all of the buggy versions of the models, the size of a model level slice is 12% to 25% of the corresponding program level slice. The authors argue that the difference is because a single model element may be implemented by several lines of code.

6.5. Reuse

Androutsopoulos et al. [2011] have defined environment-based slicing, whose purpose is to facilitate model development by specialising models for a specific operating environment. Its applications include specification reuse and property verification. The authors report on experiments that consider the reduction obtained on ten EFSM models, first by considering small number of events as the slicing criterion (a.k.a. small ignore sets) and then large number of events as the slicing criterion (a.k.a. large ignore sets). Also, they report on the performace of the slicer. The results show that for both small and large ignore sets the four algorithms consistently produce smaller slices. Slicing with ignore sets of size four reduces the number of states by 60% and the number of transitions by 70%. In the case of slicing with singleton ignore sets, if there is a large reduction (most of the EFSM) this identifies a key event, for example, a key event in the ATM system shown in Figure 5 is *Card* where slicing with respect to *Card* results in a slice containing only state *Start*. At the other extreme, i.e., considering ignore sets containing all events except one, produces an average slice size of 12.7% states and 1.1% transitions.

Lano and Kolahdouz-Rahimi [2011] describe an algorithm for slicing a restricted subset of UML state machines. The correctness of their slicing technique has been formally shown. Also, they experimentally evaluate the efficiency of their algorithm by applying it to slice five concurrent state machines, each composed of multiple copies of a component with three states. The first state machine that the slice is applied to has 3 states and 3 transitions, the second has 9 states and 18 transitions, while the last state machine has 243 states and 1215 transitions. The execution time for the smaller state machines where reasonable, e.g., 0 ms for the first state machine and 20 ms for the second. While the execution time for the larger state machines where also reasonable, e.g., the fourth state machine was 151 ms and 359 for the final state machine. This algorithm is much more efficient than the static slicing algorithm that they also defined (discussed in Section 5.1).

7. OPEN ISSUES

SBM slicing is still in the early stages and there are still issues to address.

7.1. Correctly Accounting for Control Dependence

Although there has been considerable effort in trying to correctly account for control dependence, there is still much work to be done. For example, some control dependence definitions for models are adaptations of control dependence definitions for programs [Androutsopoulos† et al. 2009]. However, the results of the survey show that work on slicing finite state machines has identified problems that are also present when slicing programs but have never been addressed. For example, slicing nonterminating finite state machines has been addressed as early as in [Heimdahl and

Table V. The SBM elements that slicing approaches remove (indicated by cross) and keep (indicated by tick) in a slice.

Approach	S[a]	T[b]	L[c]	TE[d]	G[e]	A[f]
Labbé and Gallois [2008]	√	√	√	√	√	√
Androutsopoulos[†] et al. [2009]	√	√	×	√	√	√
Heimdahl and Whalen [1997]	×	×	√	√	√	√
Chan et al. [1998]	×	×	√	√	√	√
Korel et al. [2003]	×	×	√	√	√	√
Ganapathy and Ramesh [2002]	×	×	√	√	√	√
Colangelo et al. [2006]	×	×	√	√	√	√
Wang et al. [2002]	×	×	√	√	√	×
Fox and Luangsodsai [2005]	×	×	√	√	√	×
Langenhove [2006]	×	×	√	√	√	×
Janowska and Janowski [2006]	×	×	√	√	√	×
Bozga et al. [2000]	×	×	√	√	√	×
Guo and Roychoudhury [2008]	×	×	√	√	×	×
Lano and Kolahdouz-Rahimi [2011] (static)	×	×	√	√	√	×
Lano and Kolahdouz-Rahimi [2011] (environment-based)	×	×	√	√	√	√
Androutsopoulos et al. [2011] (environment-based)	×	×	√	√	√	√
Ojala [2007]	√	√	√	×	×	×

S[a] States T[b] Transitions L[c] Labels TE[d] Triggering Events
G[e] Guards A[f] Actions

Whalen 1997] while the program slicing community only addressed the problem of slicing non-terminating programs in [Ranganath et al. 2007].

7.2. Improving Precision of Algorithms

7.2.1. State Hierarchy. When slicing hierarchical state machines, the algorithms aim to preserve the state hierarchy in the slices. The algorithms start with the lowest level of states in the hierarchy and consider all states at that level before moving up to the next level. If a state is in the slice, then so is its superstate. However, for many of them, if a state is included in the slice, then all of the sub-states are also included. This leads to larger, less precise slices. Ganapathy and Ramesh [2002] give some suggestions of how to improve precision after slicing, but these have not been implemented. Further work is required for improving algorithms to produce more precise slices of hierarchical state machines.

7.2.2. Concurrency and Communication. All approaches that slice concurrent and communicating state machines are based on extracting the dependencies and then traversing the dependencies. Most approaches handle communication and synchronisation by introducing new dependencies, similar to interference dependence that is defined when slicing concurrent programs. Computing such dependencies is complex and requires that the order of execution be considered to ensure precise slices. Even if the computed dependencies are precise, the slicing algorithm can be imprecise if it just assumes transitivity of the dependencies and traverses the reachable dependencies [Krinke 1998]. Only a few SBM slicing approaches try to compute precise dependencies, such as in [Langenhove 2006], and none actually compute precise slices. Hence there is scope for further work in improving algorithms to produce precise slices for concurrent SBMs.

7.3. Graph Connectivity

The SBM elements that are kept and removed in a slice vary from one slicing approach to another. Table V lists the elements that are removed (indicated by a cross) and kept (indicated by a tick) in a slice that is generated by each slicing approach. For exam-

ple, Labbé and Gallois [2008] do not remove any elements but simply mark those that are in the slice. Heimdahl and Whalen [1997] produce slices by deleting states and transitions. Ojala [2007] only deletes parts of transitions trigger events, guards and actions. What is not shown in Table V is whether removing transitions or states can lead to breaking the connectivity of the model, i.e., some states become unreachable. Most slicing approaches only delete transitions or states that do not cause other states or transitions to become unreachable. This leads to larger, less precise slices. Only Korel et al. [2003] have described an algorithm (their amorphous algorithm described in Section 5.5) for removing transitions and reconnecting the state machine by merging states. For example, Figure 12 shows the slice generated for the ATM state machine shown in Figure 5.

However, there is still much work to be done. First, Korel et al. [2003]'s algorithm applies a couple of rules for merging states and they suggest that more rules can be developed. Thus, this algorithm is not general enough to apply to all possible cases for merging states. Better algorithms could be developed.

Second, depending on the semantics of the state machines, slicing could introduce additional behaviour that is not in the original state machine. Assume the ATM state machine in Figure 5 has skip semantics, i.e., events produced by the environment that do not trigger a transition are consumed and the state machine remains in the same state. If stuttering event sequences are generated by the environment, then according to Weiser's notion of correctness [Weiser 1979] this slicing algorithm is incorrect. The slice obtained using Korel et al.'s amorphous algorithm with the slicing criterion $(sb, T18)$ is shown in Figure 12. An example of a stuttering event sequence is: $T1, T4, T8, T6, T18, T17, T17, T18$, where the slice and the original will not behave in the same way according to the traditional notion of correctness. Korel et al. [2003] have described a new notion of correctness with event sequences (non-stuttering ones) that ensure that the original and the slice produce the same values for the variables of interest. However, this definition of correctness is still in the early stages of development and has not been proved. Further work is required in developing slicing algorithms that improve on these issues.

7.4. Slicing Across Different Levels of Abstraction

Systems can be modelled at different levels of abstraction. For example, a system can be first modelled in a high level of detail and is often non-deterministic because of under-specification. Then it is modelled at a low level, where one state in the high level corresponds to many states in the low level. In some notations this state is modelled as a superstate. Most of the approaches, such as Wang et al. [2002], Korel et al. [2003] and Labbé and Gallois [2008] concern themselves with low-level model representations. There has been no slicing approach that has considered slicing across several models that have varying levels of abstraction.

Furthermore, a transition in a high level model can represent many transitions in a low level model. In this case, the transition may have combined labels of all those transitions, i.e., it may consist of more than one action. This could be a problem when computing data dependence using the existing definitions as dependencies may occur between different actions of a transition. This research problem has not been addressed in the literature.

7.5. Slicing Richer and Larger SBMs

Only some of the features of some SBMs have been considered by slicing approaches, such as hierarchy, concurrency, communication and event generation. Features of richer SBMs, such as UML, have not been considered. These include: the condition and selection circled connectives, delays and timeouts, the entry/exit activity and his-

tories. Moreover, how to slice SBMs with rich action languages for transitions, such as those that allow functions to be defined that interact with other parts of the program, has not been considered in the literature. Slicing approaches could be developed further so that they can be applied to any SBM.

Experimental studies were carried out by various SBM slicing approaches. The largest state machine in terms of states had 2000 states but only 79 transitions [Ganapathy and Ramesh 2002] while the largest state machine in terms of transitions was a telecommunications protocol [Bozga et al. 2000] that had 1075 states, 1291 transitions and 134 variables. Further studies to investigate the performance of the slicing algorithms when applied to large state machine in terms of both states and transitions and number of variables would be of interest.

7.6. How SBM Semantics Affect Slicing

Slicing is defined primarily on the syntax of the SBM. If we fix the syntax of a SBM and change the semantics, how is slicing affected. There are three possibilities:

(1) A different dependence graph is produced, which results in different slices.
(2) The dependence graph produced is the same. There can still be issues with the correctness of the slicing algorithms. Lano and Kolahdouz-Rahimi [2011] state that the choice of semantics can lead to computing different slices. They discuss three alternative semantics that can be used for a SBM that is not completely specified, i.e., for every state there is not a complete set of guards for every possible event occurrence. However, they do not show how these semantics lead to computing different slices.
(3) The choice of syntax/semantics can have problems of itself, which can make the correctness proofs of slicing difficult. Lano and Clark [1999] discuss ways in which problems with the semantics are overcome to achieve global consistency in SBM. For example, if a SBM allows logic on triggers and there are two parallel transitions $t_1 : \neg a/b$ and $t_2 : b/a$, then when there are no input events, $\{t_1, t_2\}$ is constructed in the step as well as the generated events $\{a, b\}$. Thus, a is generated and t_1 is taken even though t_1 is enabled by the absence of a.

Except for [Lano and Kolahdouz-Rahimi 2011], no other existing work in the survey has discussed any of these issues.

8. CONCLUSIONS

This paper is the first to survey existing work on slicing finite state machines. It comprehensively reviewed slicing approaches, classifying them in terms of their type. It also gave an overview of their applications, empirical evaluation, correctness and open problems for future work.

Work on slicing finite state machines is typically seen as extending work on program slicing to the model level. For example, some control dependence definitions for models are adaptations of control dependence definitions for programs. However, the results of the survey show that work on slicing finite state machines has identified problems that are also present when slicing programs but have never been addressed. For example, slicing non-terminating finite state machines has been addressed as early as in [Heimdahl and Whalen 1997] while the program slicing community only addressed the problem of slicing non-terminating programs in [Ranganath et al. 2007]. Moreover, we believe that the problems addressed when slicing finite state machines is similar to those required when slicing interactive programs, which has not been addressed in the program slicing community. This highlights the importance of this survey to both the model and program slicing communities.

ACKNOWLEDGMENTS

This research work is supported by EPSRC Grant EP/F059442/1. The authors also wish to thank Franco Raimondi and Khalid Alzarouni for their insightful comments.

REFERENCES

AGRAWAL, H., DEMILLO, R. A., AND SPAFFORD, E. H. 1991. Dynamic slicing in the presence of unconstrained pointers. In 4^{th} *ACM Symposium on Testing, Analysis, and Verification (TAV4)*. 60–73. Appears as Purdue University Technical Report SERC-TR-93-P.

AGRAWAL, H. AND HORGAN, J. R. 1990. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. White Plains, New York, 246–256.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, (Pearson Education).

ALUR, R. AND DILL, D. L. 1990. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*. Springer-Verlag New York, Inc., New York, NY, USA, 322–335.

ANDROUTSOPOULOS, K., BINKLEY, D., CLARK, D., GOLD, N., HARMAN, M., LANO, K., AND LI, Z. 2011. Model projection: simplifying models in response to restricting the environment. In *Software Engineering (ICSE), 2011 33rd International Conference on*. 291 –300.

ANDROUTSOPOULOS*, K., GOLD, N., HARMAN, M., LI, Z., AND TRATT, L. 2009. A theoretical and empirical study of EFSM dependence. In *Proceedings of the International Conference on Software Maintenance (ICSM)*.

ANDROUTSOPOULOS†, K., CLARK, D., HARMAN, M., LI, Z., AND TRATT, L. 2009. Control dependence for extended finite state machines. In *Fundamental Approaches to Software Engineering (FASE), Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS*. Springer Berlin/Heidelberg, York, UK.

BALL, T. AND HORWITZ, S. 1993. Slicing programs with arbitrary control–flow. In 1^{st} *Conference on Automated Algorithmic Debugging*, P. Fritzson, Ed. Springer, Linköping, Sweden, 206–222.

BINKLEY, D. 1998. The application of program slicing to regression testing. *Information and Software Technology 40,* 11, 583–594.

BINKLEY, D. AND GALLAGHER, K. B. 1996. Program slicing. In *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1–50.

BINKLEY, D. AND HARMAN, M. 2003. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, California, USA, 44–53.

BINKLEY, D. AND HARMAN, M. 2004. A survey of empirical results on program slicing. *Advances in Computers 62*, 105–178.

BINKLEY, D. AND HARMAN, M. 2005. Forward slices are smaller than backward slices. In 5^{th} *IEEE International Workshop on Source Code Analysis and Manipulation* (Budapest, Hungary, September 30th-October 1st 2005). IEEE Computer Society Press, 15–24.

BINKLEY, D., HARMAN, M., AND KRINKE, J. 2007. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems 30,* 1 (Nov.).

BINKLEY, D., HORWITZ, S., AND REPS, T. 1995. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology 4,* 1, 3–35.

BINKLEY, D. W. 2007. Source code analysis: A road map. In *Future of Software Engineering 2007*, L. Briand and A. Wolf, Eds. IEEE Computer Society Press, Los Alamitos, California, USA, 104–119.

BOZGA, M., FERNANDEZ, J.-C., AND GHIRVU, L. 2000. Using static analysis to improve automatic test generation. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer-Verlag, London, UK, 235–250.

BOZGA, M., FERNANDEZ, J.-C., AND GHIRVU, L. 2003. Using static analysis to improve automatic test generation. *International Journal on Software Tools for Technology Transfer (STTT) 4*, 142–152.

BOZGA, M., FERNANDEZ, J.-C., GHIRVU, L., GRAF, S., PIERRE KRIMM, J., MOUNIER, L., AND SIFAKIS, J. 1999. If: An intermediate representation for sdl and its applications. In *Proceedings of SDL-FORUM99*. Elsevier Science, Montreal, Canada, 423–440.

CANFORA, G., CIMITILE, A., AND DE LUCIA, A. 1998. Conditioned program slicing. *Information and Software Technology 40,* 11, 595–607.

CHAN, W., ANDERSON, R. J., BEAME, P., AND NOTKIN, D. 1998. Improving efficiency of symbolic model checking for state-based system requirements. *SIGSOFT Software Engineering Notes 23,* 2, 102–112.

CHAN, W., ANDERSON, R. J., BEAME, P., NOTKIN, D., JONES, D. H., AND WARNER, W. E. 2001. Optimizing symbolic model checking for statecharts. *IEEE Trans. Softw. Eng. 27,* 2, 170–190.

CLARKE, E. M. AND EMERSON, E. A. 1982. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*. Springer, London, UK, 52–71.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. CMIT Press.

COLANGELO, D., COMPARE, D., INVERARDI, P., AND PELLICCIONE, P. 2006. Reducing software architecture models complexity: A slicing and abstraction approach. In *Formal Techniques for Networked and Distributed Systems - FORTE 2006, Lecture Notes in Computer Science*. Springer, Paris, France, 243–258.

CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PASAREANU, C. S., ROBBY, AND ZHENG, H. 2000. Bandera: Extracting finite-state models from Java source code. In 22^{nd} *International Conference on Software Engineering (ICSE'2000)*. IEEE Computer Society Press, Los Alamitos, California, USA, 439–448.

DE LUCIA, A. 2001. Program slicing: Methods and applications. In *International Workshop on Source Code Analysis and Manipulation* (Florence, Italy). IEEE Computer Society Press, Los Alamitos, California, USA, 142–149.

DONG, W., WANG, J., QI, X., AND QI, Z.-C. 2001. Model checking UML statecharts. In *APSEC '01: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*. IEEE Computer Society, Washington, DC, USA, 363.

DUBROVIN, J. 2006. Jumbala — an action language for UML state machines. Research Report HUT-TCS-A101, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland. March.

DWYER, M. B., HATCLIFF, J., HOOSIER, M., RANGANATH, V., AND WALLENTINE, T. 2006. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006*. Springer, 73–89.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9,* 3 (July), 319–349.

FIELD, J., RAMALINGAM, G., AND TIP, F. 1995. Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages*. ACM, San Francisco, CA, 379–392.

FOX, C., DANICIC, S., HARMAN, M., AND HIERONS, R. M. 2004. ConSIT: a fully automated conditioned program slicer. *Software Practice and Experience 34*, 15–46.

FOX, C., HARMAN, M., HIERONS, R. M., AND DANICIC, S. 2001. Backward conditioning: a new program specialisation technique and its application to program comprehension. In 9^{th} *IEEE International Workshop on Program Comprenhesion* (Toronto, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 89–97.

FOX, C. AND LUANGSODSAI, A. 2005. And-or dependence graphs for slicing statecharts. In *Beyond Program Slicing*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

GALLAGHER, K. B. AND LYLE, J. R. 1991. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17,* 8 (Aug.), 751–761.

GANAPATHY, V. AND RAMESH, S. 2002. Slicing synchronous reactive programs. In *Electronic Notes in Theoretical Computer Science, 65(5). 1st Workshop on Synchronous Languages, Applications, and Programming*. Elsevier, Grenoble, France.

GASTON, C., GALL, P. L., RAPIN, N., AND TOUIL, A. 2006. Symbolic execution techniques for test purpose definition. In *Testing of Communicating Systems (TestCom'06), Lecture Notes in Computer Science*. Vol. 3964. Springer, New York, NY, USA,, 1–18.

GRAMMATECH INC. 2002. The CodeSurfer slicing system.

GUO, L. AND ROYCHOUDHURY, A. 2008. Debugging statecharts via model-code traceability. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008*. Springer Berlin/Heidelberg, Port Sani, Greece, 292–306.

GUPTA, R., HARROLD, M. J., AND SOFFA, M. L. 1992. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance* (Orlando, Florida, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 299–308.

GUPTA, R. AND SOFFA, M. L. 1995. Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 29–40.

HAMON, G. 2005. A denotational semantics for stateflow. In *Proceedings of the 5th ACM international conference on Embedded software*. EMSOFT '05. ACM, New York, NY, USA, 164–172.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8,* 3 (June), 231–274.

HAREL, D. AND KUGLER, H. 2004. The RHAPSODY semantics of statecharts (or, on the executable core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering*. Lecture Notes in Computer Science, vol. 3147. Springer, 325–354.

HAREL, D. AND NAAMAD, A. 1996. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol. 5,* 4, 293–333.

HARMAN, M., BINKLEY, D., AND DANICIC, S. 2003. Amorphous program slicing. *Journal of Systems and Software 68,* 1 (Oct.), 45–64.

HARMAN, M. AND DANICIC, S. 1997. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)* (Dearborn, Michigan, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 70–79.

HARMAN, M. AND DANICIC, S. 1998. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance and Evolution 10,* 6, 415–441.

HARMAN, M., DANICIC, S., SIVAGURUNATHAN, Y., AND SIMPSON, D. 1996. The next 700 slicing criteria. In 2^{nd} *UK workshop on program comprehension*, M. Munro, Ed. Durham University, UK.

HARMAN, M. AND HIERONS, R. M. 2001. An overview of program slicing. *Software Focus 2,* 3, 85–92.

HARMAN, M., HIERONS, R. M., DANICIC, S., HOWROYD, J., AND FOX, C. 2001. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)* (Florence, Italy). IEEE Computer Society Press, Los Alamitos, California, USA, 138–147.

HARMAN, M., HU, L., HIERONS, R. M., WEGENER, J., STHAMER, H., BARESEL, A., AND ROPER, M. 2004. Testability transformation. *IEEE Transactions on Software Engineering 30,* 1, 3–16.

HATCLIFF, J., DWYER, M. B., AND ZHENG, H. 2000. Slicing software for model construction. *Higher-Order and Symbolic Computation 13,* 4 (Dec.), 315–353.

HEIMDAHL, M. P. E., LEVESON, N., AND REESE, J. D. 1996. Experiences and lessons from the analysis of TCAS II. *SIGSOFT Softw. Eng. Notes 21,* 3, 79–83.

HEIMDAHL, M. P. E. AND LEVESON, N. G. 1995. Completeness and consistency analysis of state-based requirements. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*. ACM, New York, NY, USA, 3–14.

HEIMDAHL, M. P. E., THOMPSON, J. M., AND WHALEN, M. W. 1998. On the effectiveness of slicing hierarchical state machines: A case study. In *EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO*. IEEE Computer Society, Washington, DC, USA, 10435–10444.

HEIMDAHL, M. P. E. AND WHALEN, M. W. 1997. Reduction and slicing of hierarchical state machines. In *Proc. Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Springer–Verlag, Zurich, Switzerland.

HOLZMANN, G. J. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering 23*, 279–295.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12,* 1, 26–61.

HORWITZ, S., REPS, T., STAFF GENEVIVE ROSAY, ., STUDENTS MANUVIR DAS, . C., HASTI, R., LAMPERT, J., MELSKI, D., SHAPIRO, M., SIFF, M., TURNIDGE, T., STUDENTS, S., VISITORS THOMAS BALL, BINKLEY, D., BARGER, V., BATES, S., BRICKER, T., CAI, J., PAIGE, R., PFEIFFER, P., PRINS, J., YANG, W., RAMALINGAM, G., AND SAGIV, M. 1996-2000. Wisonsin program slicing project. URL `http://www.cs.wisc.edu/wpis/html/`.

HROMKOVIC, J. AND SCHNITGER, G. 2007. Comparing the size of NFAs with and without epsilon-transitions. *Theoretical Computer Science 380,* 1–2, 100–114.

ILIE, L. AND YU, S. 2003. Reducing NFAs by invariant equivalences. *Theoretical Computer Science 306,* 1-3, 373–390.

JANOWSKA, A. AND JANOWSKI, P. 2006. Slicing of timed automata with discrete data. *Fundamenta Informaticae, SPECIAL ISSUE ON CONCURRENCY SPECIFICATION AND PROGRAMMING 72,* 1-3, 181–195.

JHALA, R. AND MAJUMDAR, R. 2005. Path slicing. *SIGPLAN Not. 40,* 6, 38–47.

KNAPP, A. AND MERZ, S. 2002. Model checking and code generation for UML state machines and collaborations. In *Proceding 5th Workshop on Tools for System Design and Verification (FM-Tools)*, D. Haneberg, G. Schellhorn, and W. Reif, Eds. 59–64.

KOMONDOOR, R. AND HORWITZ, S. 2000. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*. ACM Press, N.Y., 155–169.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Information Processing Letters 29,* 3 (Oct.), 155–163.

KOREL, B., SINGH, I., TAHAT, L., AND VAYSBURG, B. 2003. Slicing of state based models. In *IEEE International Conference on Software Maintenance (ICSM'03)* (Amsterdam, Netherlands). IEEE Computer Society Press, Los Alamitos, California, USA, 34–43.

KRINKE, J. 1998. Static slicing of threaded programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*. ACM New York, NY, USA, Montreal, Canada, 35–42.

KUCK, D. J., KUHN, R. H., PADUA, D. A., LEASURE, B., AND WOLFE, M. 1981. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, NY, USA, 207–218.

LABBÉ, S. AND GALLOIS, J.-P. 2008. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing 20,* 6, 563–595.

LABBE, S., GALLOIS, J.-P., AND POUZET, M. 2007. Slicing communicating automata specifications for efficient model reduction. In *Proceedings of ASWEC*. IEEE Computer Society, USA, 191–200.

LAKHOTIA, A. AND SINGH, P. 2003. Challenges in getting formal with viruses. *virus bulletin*, 14–18.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7, 558–565.

LAMPORT, L. 1983. What good is temporal logic? In *Information Processing 83: Proceedings of the IFIP 9th World Congress, R. E. A. Mason, Ed.* North-Holland, Amsterdam, 657–668.

LANGENHOVE, S. V. 2006. Towards the correctness of software behavior in UML a model checking approach based on slicing. Ph.D. thesis, Ghent University.

LANGENHOVE, S. V. AND HOOGEWIJS, A. 2007. SV$_t$L: System verification through logic tool support for verifying sliced hierarchical statecharts. In *Lecture Notes in Computer Science, Recent Trends in Algebraic Development Techniques*. Springer, Berlin / Heidelberg, 142–155.

LANO, K. AND CLARK, D. 1999. Demonstrating preservation of safety properties in reactive control system development.

LANO, K. AND KOLAHDOUZ-RAHIMI, S. 2011. Slicing techniques for uml models. *Journal of Object Technology 10*, 11:1–49.

LEVESON, N., HEIMDAHL, M., HILDRETH, H., AND REESE, J. 1994. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering 20,* 9, 684–706.

MARANINCHI, F. 1991. The Argos language: graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*. IEEE, Kobe, Japan.

MEALY, G. H. 1955. A method to synthesizing sequential circuits. *Bell Systems Technical Journal 34,* 5, 1045–1075.

MILLETT, L. AND TEITELBAUM, T. 1998. Slicing promela and its applications to model checking.

MILLETT, L. I. AND TEITELBAUM, T. 1999. Channel dependence analysis for slicing promela. In *PDSE '99: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*. IEEE Computer Society, Washington, DC, USA, 52.

MOCK, M., ATKINSON, D. C., CHAMBERS, C., AND EGGERS, S. J. 2002. Improving program slicing with dynamic points-to data. In *Proceedings of the* 10^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02)*. ACM Press, New York, 71–80.

MOORE, E. F. 1956. Gedanken experiments on sequential machines. In *Automata Studies*. Princeton U., New Jersey, 129–153.

MULLER-OLM, M., SCHMIDT, D., AND STEFFEN, B. 1999. Model-checking: A tutorial introduction. In *Proceedings of the 6th International Static Analysis Symposium*. Vol. 1694. 331–354.

NOBE, C. AND BINGLE, M. 1998. Model-based development: Five processes used at boeing. In *IEEE International Conference and Workshop: Engineering of Computer-Based Systems*.

OJALA, V. 2007. A slicer for UML state machines. Tech. Rep. HUT-TCS-25, Helsinki University of Technology Laboratory for Theoretical Computer Science, Espoo, Finland.

OMG. 2001. OMG unified modeling language specification 1.4. `http://www.omg.org/cgi-bin/doc?formal/01-09-67`.

OTTENSTEIN, K. J. AND OTTENSTEIN, L. M. 1984. The program dependence graph in software development environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environmt, SIGPLAN Notices 19,* 5, 177–184.

PELLICCIONE, P., MUCCINI, H., BUCCHIARONE, A., AND FACCHINI, F. 2005. Testor: Deriving test sequences from model-based specifications. In *Eighth International SIGSOFT Symposium on Component-based Software Engineering*. LNCS 3489, St. Louis, Missouri (USA), 267–282.

PRAXIS LTD. 2008. The SPADE program analyser.

RANGANATH, V. P., AMTOFT, T., BANERJEE, A., HATCLIFF, J., AND DWYER, M. B. 2007. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems 29,* 5, 27.

SILVA, J. 2012. A vocabulary of program slicing-based techniques. *ACM Computing Surveys 44,* 3 (June), 12:1–12:41.

TIP, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages 3,* 3 (Sept.), 121–189.

VENKATESH, G. A. 1991. The semantic approach to program slicing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. ACM, New York, NY, USA, 107–119.

WANG, J., DONG, W., AND QI, Z.-C. 2002. Slicing hierarchical automata for model checking UML statecharts. In *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM)*. Springer-Verlag, UK, 435–446.

WANG, T. AND ROYCHOUDHURY, A. 2004. Using compressed bytecode traces for slicing java programs. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. 512 – 521.

WANG, T., ROYCHOUDHURY, A., AND GUO, L. 2008. JSlice, version 2.0.

WARD, M. 2003. Slicing the SCAM mug: A case study in semantic slicing. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)* (Amsterdam, Netherlands). IEEE Computer Society Press, Los Alamitos, California, USA, 88–97.

WARD, M. AND ZEDAN, H. 2007. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems 29,* 2, 7.

WEISER, M. 1979. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, University of Michigan, Ann Arbor, MI.

XU, B., QIAN, J., ZHANG, X., WU, Z., AND CHEN, L. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes 30,* 2, 1–36.