UCL DEPARTMENT OF COMPUTER SCIENCE

**Research Note**
RN/10/07

# Survey of Slicing Finite State Machine Models

14 December 2010

***Kelly Androutsopoulos, David Clark, Mark Harman,***

***Jens Krinke, and Laurence Tratt***[1]

## Abstract

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.

1 Middlesex University, The Burroughs, Hendon, London NW4 4BT, United Kingdom.

# Survey of Slicing Finite State Machine Models

K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, L. Tratt

14 December 2010

**Abstract**

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.

## 1 Introduction

Program slicing is a source code analysis and manipulation technique, in which a subprogram is identified using dependence analysis 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 [Wei79].

In recent years authors have begun to consider slicing at the model level. There are many reasons why this migration from code level to model level is interesting for researchers and practitioners concerned with program dependence analyses such as slicing e.g.:

1. Increasingly, software production happens at the design level, particularly during specification and design phases. which makes dependence analysis a worthwhile application domain.

2. As the prevalence of modelling increases and the techniques become more involved, the size and complexity of models is increasing. Therefore slicing, with its ability to focus on a chosen subproblem of interest is increasingly attractive.

3. As this paper will show, the nature of the problem of slicing (state based) modelling notations presents interesting and challenging research problems for the community.

We present a detailed review of existing work on slicing at the level of models, specifically state based models. By state based models we mean program models which are graphical specifications based on finite state machines, often with additional features such as a store, hierarchy, and parallelism. Examples include extended finite state machines, UML statecharts, STATEMATE statecharts and RSML to mention only some. We focus on state based modelling notations because these have received sufficient attention from the slicing community as to necessitate a survey paper. This paper sets out the challenges that state based model slicing presents and how the existing literature has addressed them. The paper also sets out problems that remain open either because of the challenges involved in addressing them or because the community has yet to turn its attention to them.

## 1.1 Why slicing models is interesting and useful

Let us elaborate on why slicing state based models is interesting in its own right.

It is possible, but in our opinion not appropriate, to reduce the problem of slicing models to the problem of slicing programs. This is for two reasons. The first relates to the strong relation between slicing and concrete syntax. State based models are graphs whereas programs are sequences of characters. 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. There is no equivalent level of granularity for a state based model to be sliced 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. Translating state based models into programs may lead to slices that make little sense to a modeller, where the level of granularity is an individual node.

The second relates to the semantics of programming languages and state based modelling languages. The most obvious difference between the two is that the majority of state based modelling languages allow non-determinism (i.e. when a branch in a state based model can validly take more than route); in contrast, programming languages go out of their way to avoid non-determinism. Translating a non-deterministic state based model into a deterministic programming language requires encodings. 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 can lead to bizarre state machines that appear to bear little resemblance to the original.

Translating state based models into programs and slicing at the program level is of limited use, particularly when the resulting slices need to be rein-

terpreted by humans. If, as is likely, the slice bears little obvious relation to the original input, a human is likely to discard the slice as useless. State based slicing needs to be considered as a distinct type of slicing from program slicing.

Another reason slicing state machines is interesting is that it can involve problems for which there is a loose analog of a corresponding program level feature to be found in the state based model level abstraction e.g. hierarchical state machines raise similar issues to those posed by procedural abstraction at the program level. There is also unstructured control flow, interaction with the environment, concurrency, non-termination, and multiple exit points. State based models tend to have a simpler syntax than programming languages which makes dealing with these problems a little simpler and easier. However, if one were to view the task that confronts an approach for slicing a state based model 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 task would denote a highly non–trivial challenge for current program slicing techniques; this combination of characteristics is not addressed by the current literature on slicing [BG96, Bin07, BH04, HH01, Tip95].

Finally, slicing graph based models confronts us with the explicit connectedness of these models, connectedness being an entirely implicit property for programs. This demands a strategy for explicit "re-gluing" after slicing or at least restricting slicing to an outcome which does not actually delete parts of the graph.

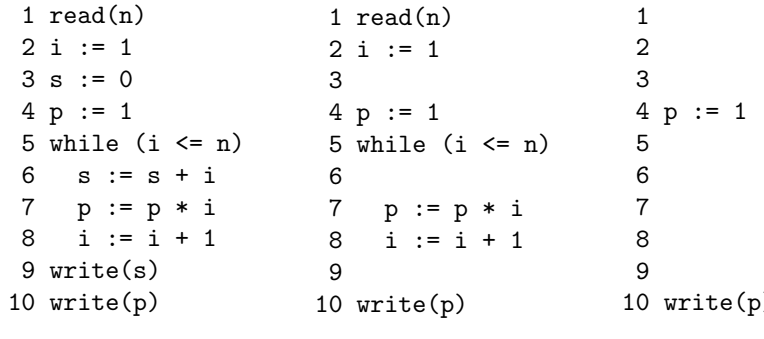
As this paper will demonstrate, there is a considerable body of existing knowledge on state based slicing, spread over many different and sometimes disjoint research communities. It is the goal of this paper to bring this knowledge together into a single survey to provide an integrated account of this knowledge and to highlight as yet uncovered directions for future research on slicing state based models.

### 1.2 Structure of the Paper

Section 2 gives a brief introduction to program slicing. Section 3 gives an overview of the finite state machines languages used for slicing and discusses their key semantic features. Section 4 presents the running example, an ATM. Section 5 categorises and discusses the slicing algorithms. Section 6, 7, 8 and 9 survey the dependence relations used when slicing. Section 10 discusses the factors that influence the choice of slicing criterion and presents the applications of FSM slicing. In Section 11 the open issues are identified and in Section 12 our conclusions are given.

## 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 state based




(a) Original program

(b) Static Slice for (10, p)

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

Figure 1: A program and two slices

slicing work directly or indirectly builds upon it.

In program slicing, the process of slicing can either produce an 'executable slice', an executable subprogram that maintains the behaviour of the original at the slicing criterion point, or it may produce a 'closure slice', which is the set of parts of the program that are relevant to the slicing criterion, but the set does not need to be executable (or even compilable). Because it is able to identify and isolate parts of a program concerned with some chosen aspect, feature or computation of interest, slicing has many applications including program comprehension [HBD03], software maintenance [GL91], testing and debugging [Bin98, HHH+04], virus detection [LS03], integration [BHR95], refactoring [KH00], restructuring, reverse engineering and reuse [CCD98]. 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 [CDH+00, JM05, DHH+06].

Slicing was introduced by [Wei79] as a static dependence analysis that produced executable slices capturing the computation necessary to maintain the effect of the original program on the slicing criterion. Since Weiser's seminal work, program slicing has been developed in many ways to include forward and backward formulations [HRB90, BH05, FHHD01], static, dynamic and hybrids [KL88, AH90, GHS92] and conditioned formulations [CCD98, FRT95, HHD+01, FDHH04] and amorphous formulations [HBD03, War03, WZ07]. Much work has also been conducted on applications of slicing, and algorithmic techniques for handling awkward programming language features [ADS91, BH93, HD98] and for balancing the trade offs of speed and precision in slicing algorithms [GS95, MACE02, BHK07].

Previous work on program slicing has led to several hundred papers on program slicing, techniques and applications, which are referenced in several survey papers [BG96, BH04, De 01, HH01, Tip95, Ven91, XQZ+05].

Weiser defined a slice as any subset of a program, that preserves a specific behavior with respect to a slicing criterion. The slicing criterion is a pair $c =$

$(s, V)$ consisting of a statement $s$ and a subset $V$ of the program's variables. Weiser observed that programmers mentally build abstractions of a program during debugging. He formalised that process and defined slicing. He also presented an approach to compute slices based on iterative data flow analysis [Wei79, Wei84].

Consider the example program in Figure 1 (a), taken from [Tip95], that computes the product `p` and the sum `s` of integer numbers up to a limit `n`. If the slicing criterion is variable `p` in line 10, 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. If we are interested in the statements that have an impact on the criterion for a *specific* execution, we can compute a *dynamic* slice, which eliminates all statements of a program that have no impact on the slicing criterion for specific inputs as determined by a specific execution. In Figure 1 (c) a dynamic slice is shown for the execution where the input to variable `n` is 0. The complete loop has been deleted because the body is never executed, together with the input statement itself.

A popular approach to slicing uses reachability analysis in program dependence graphs [FOW87]. Program dependence graphs (PDGs) mainly consist of nodes representing the statements of a program, and control and data dependence edges. In PDGs, static slicing 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 are *realizable*. This means that, for every path a possible execution of the program exists that executes the statements in the same order. In the presence of procedures, paths are considered realizable only if they obey the calling context (i.e. called procedures always return to the correct call site). [OO84] were the first to suggest the use of PDGs to compute Weiser's slices.

- *Control dependence* between two statement nodes exists if one statement determines whether the other should be executed or not.
- *Data dependence* between two statement nodes exists if a definition of a variable at one statement might reach another statement where the same variable is used.

An example PDG is shown in Figure 2, taken from [Tip95], 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`" and "`s := s+i`" have no direct or transitive influence on the criterion and are not part of the slice.

# 3 Finite State Machines (FSMs)

Finite state machines (FSMs) are used to model the behaviour of a wide variety of systems, such as embedded and non-terminating systems. They consist of

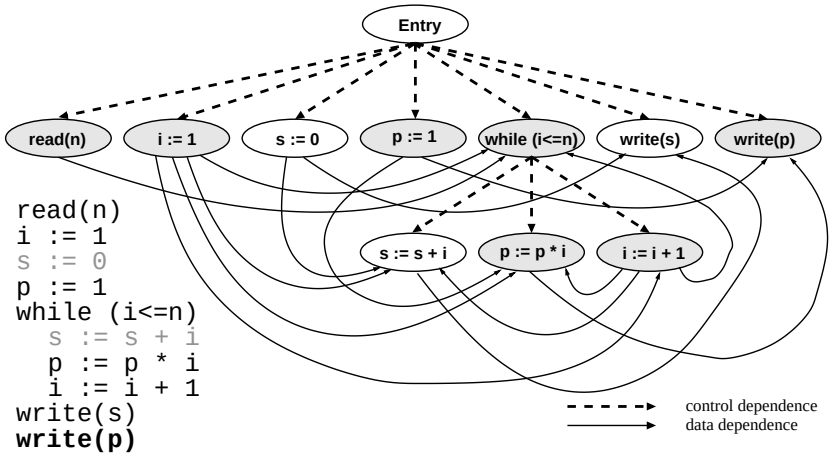



Figure 2: A program dependence graph. The slice for the criterion "`write(p)`" is highlighted in the graph and in the source text.

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 [Moo56] extend state machines with labels on states, while Mealy machines [Mea55] have labels on transitions.

Figure 3 illustrates a simple state machine that has two states, $S1$ and $S2$, and a labelled transition $T1$. $S1$ is a start state indicated by the edge with no source state. Transitions have a source and target state. The source state of $T1$ is $S1$ (we write $source(T1) = S1$) and the target state of $T1$ is $S2$ (we write $target(T1) = S2$). Transition labels are of the form $e[g]/a$, where $e$ is the triggering event, $g$ is the guard (i.e. a boolean expression) that controls whether the transition can be taken when an event $e$ occurs, and $a$ is a sequence of actions. Actions involve variable (store) updates or generation of events. All parts of a label are optional. A transition is executed when its source state is the current state, its trigger event occurs and its guard is true. If executed, the transition's action is performed and its target state become the current state.

FSMs have been extended in order to improve their expressive power or to add structure to handle the exponential blow up of states of complex FSMs. For example, if we flattened (i.e. remove hierarchy and concurrency) the statemachine in Figure 4 would have 6 states (2X2 concurrent + 2 nested) and 10 transitions. We focus on the features that lead to challenges when slicing.

- **Store.** A *store* is represented by a set of variables and can be updated by actions. For example, in Figure 5 the store is $\{sb, cb, p, attempts, l\}$.

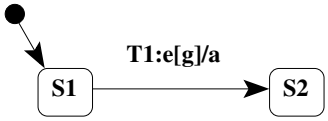



Figure 3: A simple state machine.

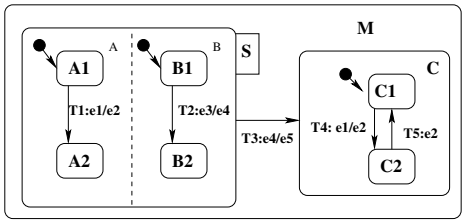



Figure 4: A hierarchical and concurrent state machine.

- **State Hierarchy.** Hierarchical states are an abstraction mechanism for hiding lower-level details. *Composite states* (known in statecharts [Har87] as OR-states) are those which contain other states; while those that do not are termed *basic states*. For example, in Figure 4 $C$ is a composite state, while $C1$ and $C2$ are basic states. A hierarchical state machine can be considered as an internal structure of some higher-level composite state, often referred to as the *root* or *top level* state. For example, in Figure 4 the root state is $M$. *Nested states* refer to the sub-states of composite states. A *superstate* (or 'parent state') of a nested state refers to the composite state one level up. In Figure 4, the superstate of nested states $C1$ and $C2$ is $C$, and the superstate of $C$ is $M$.

- **Concurrency and Communication.** *Concurrency* means that two or more machines or superstates (known as parallel, or AND, states in statecharts [Har87]) can be executed in parallel. For example, in Figure 4, $A$ and $B$ are parallel states (divided by a dashed line). Concurrent machine communication can be synchronous or asynchronous. After sending an event, state machines with synchronous communication must wait until the receiver finishes processing it. After sending an event, state machines with asynchronous communication, can continue without waiting for the receiver to finish processing it.

- **Event generation.** Events can be generated by the state-machine itself by 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*.

- **Parameterised events.** Special types of events can be defined with parameters whose values are determined from the environment. For example, in Figure 5 event *PIN* has the parameter $p$ that represents the pin entered by the user. Parameterised events can be simulated by a set of events without parameters. Each event in this set corresponds to a combination of parameter values.

- **Time.** Some FSM variants add features for modelling time. For example,

timed automata [AD90] model clocks using real-valued variables.

FSMs can also have the following properties:

- **Non-determinism.** FSMs can be *deterministic* or *non-deterministic*. In deterministic FSMs, for each pair of state and event there may be only one possible next state i.e. only one transition is executed. In non-deterministic FSMs, for each pair of state and event there may be several possible next states, i.e. an event may trigger several transitions and if their guarding conditions are all true, then there are several possible transitions to execute.

- **Non-termination.** FSMs can be *terminating* or *non-terminating*. Terminating FSMs have at least one explicit exit or final state. Non-terminating FSMs do not have an exit state and their execution is infinite.

Table 1 gives an overview of all the FSM variants used by the various slicing approaches that we survey and compares them according to whether they are concurrent, hierarchical, and how they communicate. The highlighted rows group FSMs with common features.

All FSM variants allow for a machine to be potentially non-terminating and/or non-deterministic. However, some slicing approaches are restricted to terminating and/or deterministic FSMs. For example, [KSTV03] restrict their approach to apply only to EFSMs with unique exit state and that are deterministic.

# 4 A Running Example

We model the Automatic Teller Machine (ATM) system using state machines in three different ways and use these as running examples. This is because we want to illustrate how the differences between FSM variants affect slicing. The first example models the ATM using a FSM variant that has no concurrency or state hierarchy and is deterministic with a unique exit state. The second example extends the first model by making it non-deterministic and non-terminating (i.e. no exit state). Finally, the third 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 [KSTV03] for EFSMs. 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 2). 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

Table 1: Comparison of a subset of FSM variants (i.e. those used for slicing) according to some features.

| **FSM Variant** | **Slicing Approach** | **C** [a] | **H** [b] | **SC** [c] |
|---|---|---|---|---|
| Extended Finite State Machines (EFSMs) | [KSTV03] [ACH+09] | × | × | × |
| Software Cost Reduction (SCR) [HBGL95] | [HKL+98] [RLHL06] | √ | × | S [d] |
| Synchronous Adaptive System (SAS) [ASS+07] | Schaefer and Poetzsch-Heffter [SPH08] | √ | × | S |
| Unified Modelling Language (UML) Statecharts | [CCIP06] | √ | × | S |
| UML Activity Diagrams | [Esh02] | √ | × | S |
| Timed Automata [AD90] | Janowska and Janowski [JJ06] | √ | × | S |
| Input/Output Symbolic Transition Systems (IOSTs) [GGRT06] | [LG08] | √ | × | S |
| Extended Automata | [BFG00] | √ | × | A [e] |
| UML Statecharts v1.4 | [Oja07] | √ | × | A |
| Statecharts [Har87] | [FL05] Guo and Roychoudhury [GR08] | √ | √ | S |
| Argos [Mar91] | Ganapathy and Ramesh [GR02] | √ | √ | S |
| Requirements State Machine Language (RSML) [LHHR94] | Heimdahl and Whalen [HW97] [CABN98] | √ | √ | S |
| Extended Hierarchical Automata (EHA) | [WDQ02] [Lan06] | √ | √ | S |

[a] C = Concurrency
[b] H = Hierarchy
[c] SC = Synchornisation or Communication
[d] S = Synchronous
[e] A = Asynchronous

$cb$ 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.

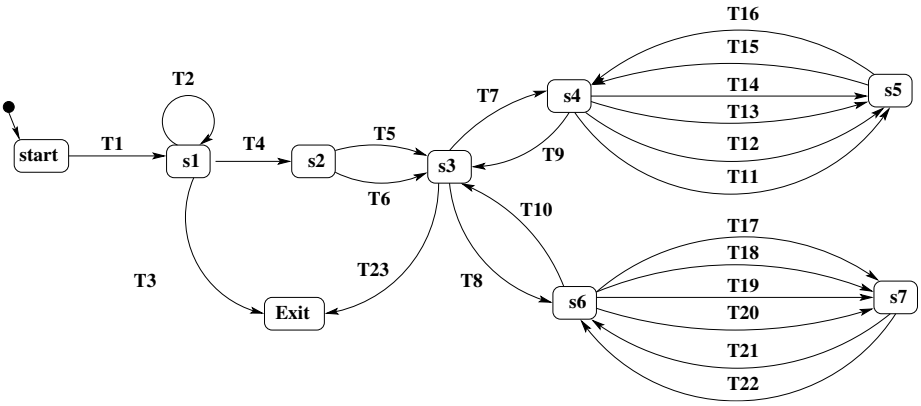



Figure 5: The ATM system as modelled by [KSTV03] for EFSMs with a unique exit state.

Figure 6 is a non-terminating and non-deterministic variant of Figure 5. It is non-terminating because there is no exit state and the target state of transitions $T3$ and $T23$ is *start*, i.e. after $T3$ or $T23$ are executed the ATM system is ready to accept a new customer. It is non-deterministic because transitions $T5$ and $T6$ in Figure 5, which allow the user to choose between English and Spanish, have been removed and replaced in Figure 6 by a single transition that assumes that a language has been chosen, but does not specify which one. This introduces non-determinism e.g. if the machine is in state $S6$ and the $Balance$ event occurs, then two transitions (either $T25$ or $T26$) can be executed .

Figure 7 shows the final ATM variant, that is hierarchical, concurrent and has generated events. 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 7 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 (see Figure **??**). The $bank$ concurrent state models the bank's behaviour as described in [KM02] (see Figure 1 on page 60). 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 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. In Figure 6 only the second verification

Table 2: The transitions of the ATM system as illustrated in Figure 5 and 6.

| 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/Checking”) |
| T6: | Spanish/l=‘s’; print(“Ahorros/Corriente”) |
| T7: | Checking |
| 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/Checking”) |
| 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/Checking”) |
| T22: | Receipt[l=‘s’]/print(“Balanza=”,sb); print(“Ahorros/Corriente”) |
| T23: | Exit/print(“Ejecting card”) |
| T24: | ChooseLanguage |
| T25: | Balance/print(“Balance=”,sb) |
| T26: | Balance/print(“Balanza=”,sb) |
| T27: | Receipt/print(“Balance=”,sb); print(“Savings/Checking”) |
| T28: | Receipt/print(“Balanza=”,sb); print(“Ahorros/Corriente”) |

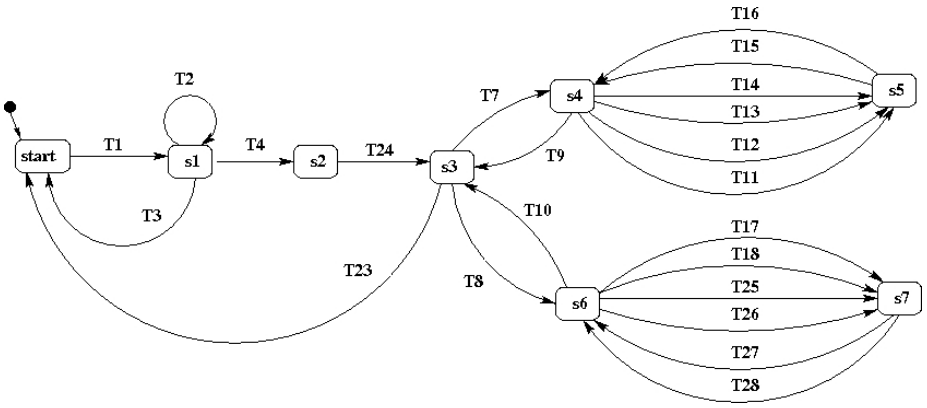



Figure 6: The ATM system modelled for EFSMs that is non-terminating and non-deterministic because of transitions $T25$ , $T26$ as well as $T27$ and $T28$ (see Table 2).

stage is modelled by the self-transition $T2$.

# 5 Algorithms for Slicing FSMs

We classify the algorithms of the FSM slicing approaches into the following types: (1) algorithms that translate FSMs to programs to apply program slicing, (2) dependence graph based algorithms, (3) FSM traversal algorithms, (4) algorithms that iteratively apply rules to a normal form, (5) dataflow analysis based algorithms, (6) hybrid algorithms, and (7) multi-tiered FSM slicing approaches. For each of these, we describe what we mean and then we describe the specific algorithm of that type for each slicing approach. Most of the slicing algorithms involve some dependence analysis. We only briefly mention the dependence relations in this section but discuss them in detail in Sections 6, 7, 8 and 9. Figure 8 illustrates the number of slicing approaches that adopt a particular type of algorithm.

Slicing algorithms can have one or more of the following properties: precision, correctness, executability and optimality (or minimality). A slicing algorithm is *precise* [LG08] if it considers all of the transitive dependencies, for all dependence relations, in the FSM model, and only those. A slicing algorithm is *correct* [LG08] if it consists only of transitions that have influence on the slicing criterion and none of these are sliced away. This means that the original model will be a correct slice of itself, as well as other models that contain transitions that do not influence the criterion. This definition of correctness should be generalised to include states as well as parts of transitions as some FSM slicing approaches can slice these away. A slicing algorithm is *optimal* or *minimal* [LG08] if it contains only the transitions that actually have influence on the

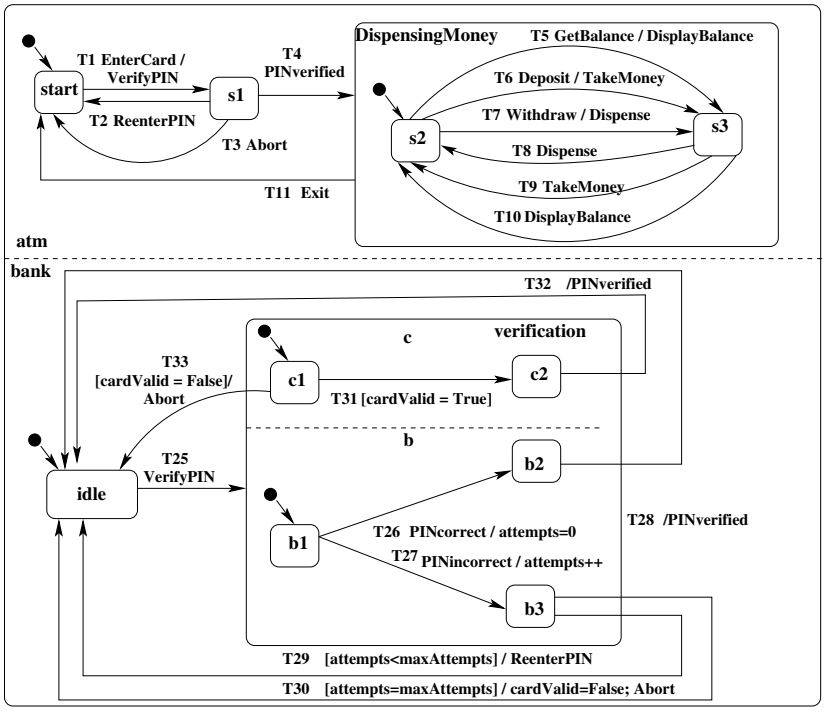



Figure 7: The ATM system modelled by finite state machines with generated events.

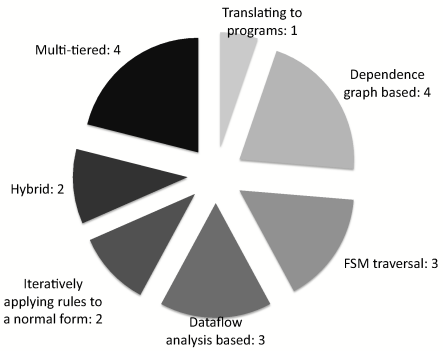



Figure 8: The number of slicing approaches that adopt a particular type of algorithm.

criterion, i.e. its the smallest correct slice. Precise slicing is a notion that lies between correct slicing and optimal slicing. [KSTV03] define the executability property of a slice, if the slice is a sub-model of the original (preserves the functionality of the original) and is executable. A model is executed when given a sequence of events (and event parameters), transitions are taken. A model can only be executed if it is deterministic or non-determinism has been resolved.

## 5.1 Translating to Programs to Apply Program Slicing

A possible approach to slicing a state machine is to translate it into a program, apply slicing at the program level and then map the resulting slice back to the state machine level. Any program slicing approach can be applied, such as static backwards, forwards or dynamic program slicing. This is an attractive approach since it avoids developing new slicing algorithms at the model level and there is around 30 years of work on program slicing and some existing tools, including a commercial tool [Gra02]. However, the main problem with this approach is with the mapping of the program slice back into the model. This is particularly difficult in the case of hierarchical and concurrent state machines if the structure of the original state machine is to be reflected in the slice. Moreover, the state machines should be deterministic as this leads to less challenges when generating code, i.e. it avoids determining how to automatically resolve non-determinism. 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.

In the literature, only [GR08] have described an approach that slices statecharts [Har87] by translating them into code and applying dynamic slicing. In this way, they avoid developing and popularising a tool for slciing statecharts and use existing tools for slicing programs. First Java code is automatically generated from the statecharts while using appropriate tags to store model-code

association information. Then, subject to an error being detected, dynamic slicing, using the JSlice [WRG08] tool, is applied to the Java code. The slicing criterion is the last state entry point in the code where the error occurred. The resulting slice is then mapped back to the statechart model using the association between model entities and the generated code. Also, it is further processed to maintain the hierarchical and concurrent structure of statecharts. No proof of correctness have been given for the mappings between the statecharts and the code. [GR08] point out that since statecharts often contain more information than at the code level, i.e. code is a refinement of a statechart in the model-driven development process, the debugging activity is more focused as it ignores bugs in the model that do not occur at the code level.

## 5.2 Dependence Graph Based Algorithms

Dependence graph based algorithms for slicing FSMs are similar to those for slicing programs (see Figure 2 for an example of a PDG used by some program slicing algorithms), in that they are defined as a reachability problem in a dependence graph representation. Both static backward and forward slices can be generated using these types of algorithms. Dependence graphs for state machines vary from those representing programs and amongst different FSM slicing approaches. However, for all slicing approaches the slicing criterion includes a node or set of nodes in a dependence graph. Also, the dependence relations used to compute the dependence graph can vary from one slicing approach to another.

The algorithms described in [ACH+09, KSTV03] 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 dependence relations. A dependence graph is a directed graph where nodes represent transitions and edges represent data and control dependencies between transitions. Note that the dependence graphs generated by [ACH+09] and [KSTV03] differ because they use different definitions of control dependence (see Section 7). Then the algorithm starts from the node in the dependence graph representing the slicing criterion and nodes (i.e. transitions) that are backward reachable from the slicing criterion in the dependence graph are marked in the slice. Once the transitions in the slice have been marked, [ACH+09] and [KSTV03] have implemented different algorithms for automatically reducing the size of an EFSM slice, and we discuss each respectively.

[KSTV03] describe two slicing algorithms for automatically reducing the size of the EFSM slice. The first slicing algorithm produces slices that are syntax preserving, i.e. they are executable sub-models of the original EFSMs and thus are not much smaller than the original. Consider the ATM example shown in Figure 5. The slice obtained using the first algorithm, as described in [KSTV03], with the slicing criterion $(sb, T18)$ is illustrated in Figure 9. This slice could be produced by just applying a reachability algorithm. The second slicing algorithm is a hybrid approach and is discussed in Section 5.6.

The algorithm described by [ACH+09] anonymises all unmarked transitions

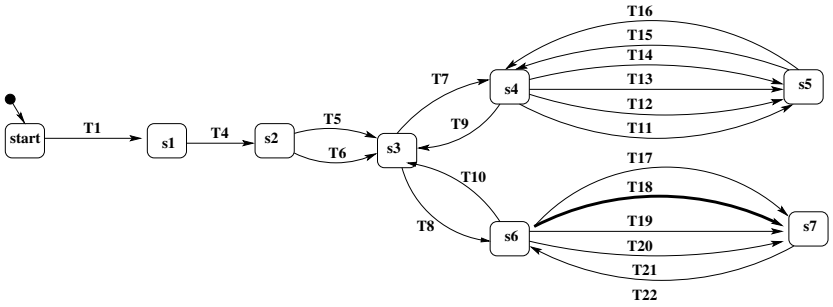



Figure 9: The slice generated for the ATM system, shown in Figure 5, with respect to $(sb, T18)$ (highlighted) using [KSTV03] first algorithm. The transition labels given in Table 4.

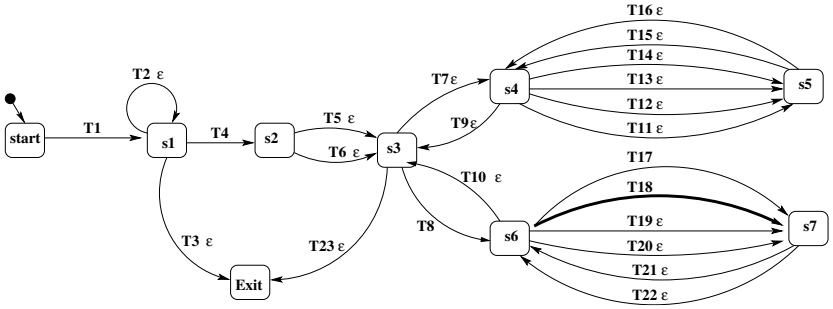



Figure 10: The slice generated for the ATM system, shown in Figure 5, with respect to $(sb, T18)$ using the algorithm by [ACH+09]. The labels of marked transitions are given in Table 4, while unmarked transition have the label $\varepsilon$ indicating an empty label.

i.e. they have empty labels. A slice with unmarked transitions may introduce non-determinism where none previously existed. Consider the ATM example shown in Figure 5. If the slicing criterion is $(sb, T18)$, the slice produced using is shown in Figure 10, where $\varepsilon$ represents unmarked transitions. Non-determinism is introduced at any state where there are more than one outgoing transitions with empty labels 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.

[LG08] have presented an approach for slicing Input/Output Symbolic Transition Systems (IOSTSs). The slicing criterion is a set of transitions. The algorithm is similar to the algorithms in [ACH+09, KSTV03] whereby a dependence graph is constructed and transitions that are backward reachable from the slicing criterion are marked. The slice consists of the transitions that correspond

to the nodes that have been marked i.e. similarly to [ACH+09] the size of the model is not reduced.

[FL05] have defined And-Or dependence graphs that are used to slice statecharts [Har87]. The And-Or dependence graphs are based on dependence graph as in [KKP+81] 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.

## 5.3 FSM Traversal Algorithms

FSM traversal algorithms use dependence relations to determine what is in the slice by searching through the states and transitions of the state machine or a graph representing the state machine and marking the elements in the slice. These approaches tend to lack precision as they tend to include more elements than just the transitive dependencies. We discuss this problem using the ATM example and the algorithm described in [GR02].

[GR02] have presented an algorithm for slicing Argos specifications. Argos is a graphical language based on statecharts with hierarchical states and concurrent state machines used to specify synchronous reactive systems. The slicing criterion $< S, x >$ is given as the name of a state $S$ and an output signal $x$. The slicing algorithm is a traversal algorithm that works on a graph representing a state machine $M$ whose nodes correspond to the states of $M$. The graph consists of three types of edges. A *transition edge* exists for every transition $M$, from a node representing the source state to a node representing the target state. 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 $t_2$.

The algorithm starts from $S$ and traverses down the hierarchy edges including the states that preserve the behaviour of $M$ according to $x$. 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 (i.e. along hierarchy edges first and then transition edges) 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.

Suppose that the ATM example shown in Figure 7 is an Argos specification. Figure 11 shows the resulting slice after applying the slicing algorithm described in [GR02] with respect to the slicing criterion $< bank, Abort >$. The nested states and transitions of *DispensingMoney* are the only elements removed. This algorithm lacks precision as it includes more elements in the slice than is necessary. Moreover, it focuses on removing states rather than transitions or

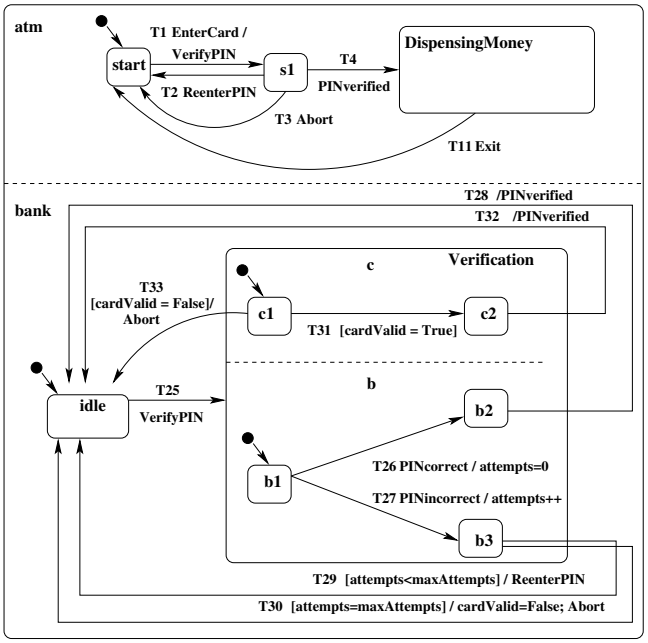



Figure 11: The slice produced by applying the algorithm in [GR02] to the ATM example in Figure 7.

parts of labels of transitions, such as actions or guards. For example, the action of transition $T26$ (i.e. $attempts = 0$) could be removed as it is redefined before it is used again with respect to event *Abort*.

[WDQ02] have presented a slicing algorithm for Extended Hierarchical Automata (EHAs) [DWQQ01]. 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 in a given property to be verified. 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 the 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 on the algorithm described by [GR02] that only deletes states and the transitions associated with that state, but not parts of transitions.

[LH07] have defined two new slicing algorithms, as part of the $SV_tL$ (System Verification through Logic) framework. The first algorithm is an extension of the algorithm defined in [WDQ02] for slicing a single statechart. It removes

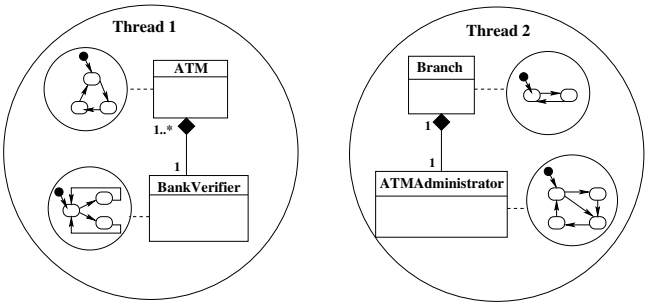



Figure 12: An example of a multi-threaded behavioural modal of the bank and ATM system.

false parallel data dependencies by taking into account the execution chronology and defining a Lamport-like [Lam78] 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 12 shows an example of a bank and ATM system modelled in UML as two threads of control with their classes and collection of statecharts. A collection of statecharts can be reduced to a single statechart, where each statechart will correspond to a separate concurrent region (AND-state). However, they choose not to do this and to slice across all statecharts in order to keep the object-oriented structure of the model. [LH07] 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 [GR02], but draws edges between statecharts rather than statechart elements. Then $\mathrm{SV}_t\mathrm{L}$ starts running an instance of the slicing algorithm for a statechart, e.g. *BankVerifier* in Figure 12. 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 Figure 12, then another instance of slicing algorithm is executed. For $n$ statecharts in the worst case $\mathrm{SV}_t\mathrm{L}$ will execute $n$ slicing algorithms in parallel. [LH07] 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.

## 5.4 Iteratively Applying Rules to a Normal Form

FSM slicing algorithms that iteratively apply a set of rules to a normal form typically define rules for determining whether to keep (or remove) elements in a state machine. Dependence relations are not explicitly defined but are coded in the rules. Ideally the rules should be independent in order to be applied in any order.

[CABN98] have defined an algorithm for slicing RSML (Requirements State Machine Language) specifications. RSML [LHHR94] is a requirements specification language that combines a graphical notation that is based on statecharts [Har87] 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 applies recursively the following rules until a fixpoint 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, 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 ATM example in Figure 7 as RSML can deal with both concurrency and hierarchy. If the slicing criterion is $T33$ and the event is the generated event *Abort* (we assume that this transition and event is in some property to be checked), then the slice produced is as in Figure 11.

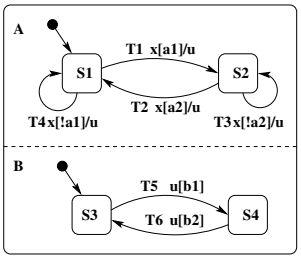



Figure 13: A state machine, as in [CAB+01], that illustrates the false dependency: event $u$ appears to depend on both conditions $a1$ and $a2$, but regardless of the truth values of $a1$ and $a2$, $u$ is generated because of event $x$.

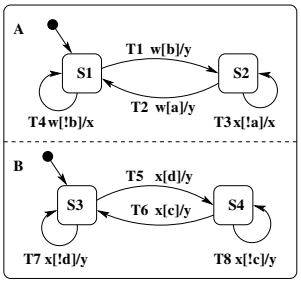



Figure 14: A state machine, as in [CAB+01], that illustrates the false dependency: event $y$ does not depend on any guarding condition.

[CAB+01] point out that it is possible for the algorithm to include false dependencies, i.e. to show that elements are dependent on each other when they should not be. False dependencies may increase the size of the slice and can mislead as to which elements actually affect the slicing criterion. For example,

in the state machine shown in Figure 13, event $u$ appears to depend on the guarding conditions $a1$ and $a2$. However, it does not because regardless of the truth values of $a1$ and $a2$, event $u$ is always generated because of event $x$. This type of false dependency can be automatically detected by checking whether the disjunction of the guarding conditions is a tautology in the case when the triggering event and action is the same. Another false dependency, that is harder to find automatically, is illustrated with the state machine in Figure 14. Event $y$ does not depend on any of the guarding conditions because it is always generated either one or two execution steps after $w$.

[EW04] have presented a slicing algorithm that iteratively applies four rules for reducing the state space of an UML activity hypergraph. UML activity hypergraphs can be considered as an alternative type of state machine. The slicing criterion consists of the variables and events in a property $\phi$ to be model checked. The first two rules apply on the semantics and the last two on the syntax of the activity hypergraphs. The novelty in their approach is that the first two rules restrict the environment based on what is in the slice, i.e. only events that trigger relevant hyperedges can be produced by the environment.

The rules are defined in [EW04] as follows. An external event $e$ is only allowed to occur if it triggers a relevant hyperedge. A hyperedge is a transition relation between nodes and a relevant hyperedge is a hyperedge identified as being in the slice. This rule is applied only if $\phi$ does not refer to any external events and it removes the possibility of any hidden loops. The second rule states that no two named external events can occur at the same time, i.e. they can only be interleaved. This rule is applied only when $\phi$ does not refer to any external events. Also, the activity hypergraph must satisfy some constraints given in [Esh02], such as it does not have duplicate conflicts, to ensure that the slice behaves similarly to the original. The third rule states that a variable $v$ can be removed, as well as every guard that refers to $v$, if: $\phi$ does not refer to $v$, $v$ is not updated by concurrent activities, the only hyperedges that refer to $v$ are those that leave the activity where $v$ is updated, and the disjunction of guard expressions referring to $v$ in decisions is true. The fourth rule describes various cases when nodes can be removed from the activity diagram in a way that ensure that $\phi$ can still be evaluated on the sliced activity hypergraph.

## 5.5 Dataflow Analysis Based Algorithms

In program slicing, [Wei79] introduced an iterative dataflow analysis for computing program slices. This consists of using the control flow graph (CFG) of a program and repeatedly solving a set of dataflow equations for each node in the CFG until a fixpoint is reached. A CFG is a directed graph with a set of nodes representing statements and edges representing control flow. In FSM slicing, dataflow analysis based algorithms adopt a similar approach to those of program slicing, i.e. dataflow equations are evaluated and some approaches use a CFG representation of the state machine.

In [HKL$^+$98, BH99], the slicing technique presented removes variables that are irrelevant with respect to the slicing criterion as well as tables that define

them, and state machines in the case of monitored variables (variables that represent sensor states). The slicing criterion is the set of variable names that occur in the property to be model checked. Let $RF$ be the set of all variable names in a specification. The variables on which a given variable directly depends on in the new state are defined using three dependence relations: $D_{new}$, $D_{old}$ and $D$ on $RF \times RF$. For variables $r_i$ and $r_j$:

- $(r_i, r_j) \in D_{new}$ if $r'_j$ (primed variables are evaluated in the new or next state, while unprimed in the old or current state) is a parameter of the function that defines $r'_i$, i.e. the function maps $r'_i$ to the set of possible values.
- $(r_i, r_j) \in D_{old}$ if $r_j$ is a parameter of the function defining $r'_i$.
- $D = D_{new} \cup D_{old}$.

The variables in $RF$ are partially ordered. No circular dependencies of state variables are allowed in the definitions of state variables. Let $\mathcal{O} \subset RF$ be a set of variables occurring in a property $q$. [BH99] define the set $\mathcal{O}^*$ as the reflexive, transitive closure of $\mathcal{O}$ under the dependence relation $\mathcal{D}$ of an SCR specification for state machine $\Sigma$. The slice $\Sigma_A$ of $\Sigma$ is produced by deleting all of the tables corresponding to variables in set $RF - RF_A$, where $RF_A = \mathcal{O}^*$.

[Oja07] has presented a slicing approach for UML state machines (specifically UML 1.4 [OMG01]). The guards and actions of transitions are expressed in Jumbala [Dub06], 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.

[JJ06] have described a static backward algorithm for slicing timed automata with discrete data. They consider only automata with reducible control flow as defined in [ASU86], i.e. those that have two disjoint sets of transitions, one set forms an acyclic graph, while the other consists of transitions whose source dominate their targets. 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 $\phi$ representing a given property to be verified. The first set consists of all enabling conditions and actions defining

variables in $\phi$. The second set consists of the states in $\phi$ 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.

## 5.6 Hybrid Algorithms

Hybrid algorithms adopt two or more of the other type of algorithms. For example, an algorithm could be both dependence based and iteratively apply a set of rules, such as the second algorithm given by [KSTV03]. Slicing algorithms may apply three types of rules iteratively. First, rules can be defined for determining what elements are in the slice. Second, rules for merging states and deleting transitions, i.e. re-connecting the graph after removing elements not in the slice. Third, rules for applying several slicing techniques until there are no more changes. Algorithms that iteratively apply the last two types of rules, usually also adopt one or more of the other kinds of algorithms discussed. The third type of rules apply to multi-tiered slicing algorithms which we discuss in the next section.

The second slicing algorithm described by [KSTV03] 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 transitions. This approach addresses the challenge of re-connecting the state machine after states and transitions are removed. However, since the solution given is in terms of two rules, these are not general enough to cover all possible cases, i.e. for different structured state machines these rules might not be very effective and might contain some irrelevant elements. 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. [KSTV03] 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 15. 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. Also, for the stuttering event sequence: $T1, T4, T8, T17, T17, T18$, the slice and the original will not behave in the same way.

[CCIP06] have described an approach for slicing Software Architecture (SA) models. The Charmy tool is used for specifying SA models as state machines,

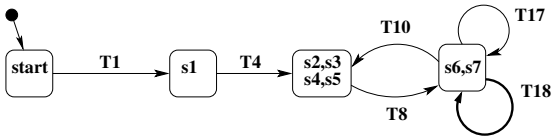



Figure 15: The slice generated for the ATM system, shown in Figure 5, with respect to $(sb, T18)$ using [KSTV03] second algorithm. The transitions are labeled as described in Table 4.

using an extended subset of UML state machines. Properties are specified using the Property Sequence Charts (PSC) language, an extension of UML 2.0 sequence diagrams for specifying Linear-Time Temporal Logic (LTL) properties. The slicing criterion is a property expressed in PSC. The slicing algorithm, which is based on TeSTOR (a TEst Sequence generaTOR algorithm) [PMBF05], marks the states of the state machine that are required and deletes any unmarked states and any transitions that have unmarked source or target states. TeSTOR is based on the idea that scenarios are not completely specified, but important for modelling expected system interactions. State machines are used to complete scenarios by recovering missing information. Thus, the input of the Testor algorithm is state machines and scenarios. The output of the algorithm is a set of sequence diagrams that contain the sequence of messages given by the input sequence diagram and completed by the information contained in the state machines. Two rules are used for marking.

*Rule 1:* Every source and target state of a message in the slicing criterion (i.e. PSC) in at least one test sequence generated by TeSTOR is marked.

*Rule 2:* For all variables of transitions that have a marked target state, find all of their occurrences in the same state machine and if these are contained in transitions that do not have a marked target state $s$, then each path leading from the initial state to $s$ is marked. This rule identifies paths from the initial state to all occurrences of variables that are marked.

## 5.7 Multi-tiered FSM Slicing Approaches

Multi-tiered FSM slicing approaches define two or more slicing algorithms that can be implemented in any of the previously mentioned kinds of algorithms. They differ from hybrid algorithms as they consist of several independent slicing algorithms, while hybrid algorithms refer to a single algorithm whose parts correspond to two or more different kinds of algorithms. The algorithms can be applied independently or combined. However, in some approaches, such as in [SPH08], the order in which the algorithms are applied can affect the performance and complexity of the analysis.

In [HW97, HTW98], a two tiered approach is presented for slicing RSML specifications. First, the RSML specification is automatically reduced based on a specific scenario of interest (a domain restriction), by removing all behaviours

that are not possible when the operating conditions defining the reduction scenario are satisfied. This is a form of conditioned slicing [CCD98], which for programs adds a condition to the traditional static slicing criterion that captures the set of initial program states. 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.

Static and backward slicing based on data flow or control flow is applied to the remaining specification in order to extract the parts effecting selected variables and transitions. For computing data flow slices, the slicing criterion is a transition or variable. The algorithm traverses the data dependence graph, produced using data flow relation, and marks all elements that directly or indirectly affect the truth value of the guarding transition. Unmarked elements are removed. For computing control flow slices, the slicing criterion is a transition. Given a transition $t$ with event $e$, the slice should include 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 environmental inputs.

By using a different tag for each slice, slices can be combined by applying the standard set operators: union, complement and intersection. The slicing algorithms are based on a marking of the abstract syntax tree, similar to Sloane's and Holdworth's approach [SH96]. The RSML parser in [HL95] is modified in order to mark the abstract syntax tree based on different slicing criteria.

[BFG00] have presented three independent types of static slicing techniques for slicing asynchronous extended automata for improving test case generation: relevant control analysis, relevant variable analysis and constraint propagation. These can be applied in any order, and iteratively until no more reductions are possible. The slicing criterion is a test purpose and a set of feeds. A test purpose describes 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 aims to provide to the IUT during a test. Only the first two slicing techniques have been implemented within the IF [BFG$^+$99] framework, an intermediate program representation based on asynchronous communicating timed automata that was developed to support the application of static analysis techniques.

Relevant control analysis 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. external events. Thus, the algorithm for computing this slice performs reachability analysis. It starts with the set of states containing the initial state and an empty set of transitions. A set of rules are defined with respect to feeds

and are applied one at a time, until the least fixed point is reached.

Relevant variable analysis 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. Initially, it has empty sets of variables for each state. Then, for each transition at each step, the set of used variables is recomputed and the relevant variables set for the source state is updated. This process continues until there are no more changes in the relevance sets for any transitions. The variables that are irrelevant are replaced by the symbol $\top$.

Constraint propagation 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/interprocesses 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.

[SPH08] have presented a static and backward slicing approach for Synchronous Adaptive System (SAS) models. SAS [ASS$^+$07] is based on transition systems (transition systems specific formalisations of systems that represent a system as a set of variables and a state consists of an assignment of values to these variables) and was developed as a semantics-based backend for the MARS (Methodologies and Architectures for Runtime Adaptive Systems [TAFJ07]) modelling concepts. Systems are modelled in SAS as a set of synchronous modules, where each module has a set of predetermined behaviour configurations that it can choose to adapt to depending on the state of the environment. Slicing can be applied at three different levels of detail: system slicing, module slicing and adaptive slicing. These can be combined in order to have finer control over the reductions. The choice of level in which slicing is to be applied as well as the order in which the slicing techniques are combined determines what parts of the model are possibly going to be removed and the amount of effort required for the analysis i.e. it is better to remove the largest irrelevant parts first before applying further slicing techniques.

System slicing is performed on the system level. The slicing criterion is the set of variables in a given property $\phi$ to be verified. The algorithm starts from the modules that directly affect the slicing criterion and then iteratively the transitive closure of the modules connected to the inputs of modules is computed until a fixpoint is reached. All modules that do not affect the slicing criterion can be removed.

Module slicing is performed on the intra-module level by analysing which assignments in the transition functions of the module influence the variables in $\phi$ and removing any irrelevant variables. Initially, the slice set consists of the variables in a module $M$ that occur in $\phi$. Then, iteratively the dependencies of the variables in the set are computed, i.e. data, control and adaptive dependence,

and added to the slice set, until a fixpoint is reached. This algorithm can also be applied at the system level, but instead of only considering variables within a module in $\phi$ it considers variables across modules in $\phi$.

Adaptive slicing is applied when the property to be verified refers only to the adaptation behaviour, and since the adaptation behaviour is syntactically independent from the functional behaviour, the functional behaviour of all of the modules can be removed. The adaptation behaviour and the configuration guards are kept.

[RLHL06] use slicing for generating efficient code from Software Cost Reduction (SCR) [Hei02, HBGL95] specifications. SCR is a formal method used to specify the requirements of critical systems as a state machine using a tabular notation. The tables spell out the steps of execution, i.e. the transition relation. The slicing approach consists of applying input slicing, simplification and output slicing to an execution-flow graph and then generating code from it. An execution-flow graph is a specialised control-flow graph defined for representing SCR specifications. It has three types of nodes: header nodes that represent states in which the system is waiting for input, switch nodes that represent the computation of new values of mode classes when transitions occur, and table nodes that represent the computation of new values of dependent variables. Edges in the graph represent the control flow between these nodes.

Input slicing, which is a form of static forward slicing, removes any nodes in the execution-flow graph that update variables whose values cannot change for given inputs. An input usually does not cause an update to every variable, and by removing the nodes where the variables do not change, the amount of processing is reduced. The slicing criterion is an input and a header node. The algorithm computes the set of variables that are update-dependent, directly, or transitively, (defined in Section 9) on a given input and then the table and switch nodes that are reachable from the given header node are recursively copied. A table or switch node is only copied if the variable they depend on is in the update-dependent set, otherwise they are removed and replaced by a copy of the tree reachable from the outgoing edge of the table or switch. This process continues until the header node is reached, whereby new incoming edges are created to the header node.

The simplification technique uses information about modes and inputs encoded in the execution-flow graph, as well as other information given as assertions and assumptions, to reduce the complexity of expressions. It is not a slicing technique, so we focus our discussion only on the input and output slicing techniques.

Output slicing is a form of static backward slicing and is applied only after simplification. It removes any computations that cannot influence a program's output (i.e. controlled variable). The algorithm requires the identification of all prestate and poststate variables that are live at nodes of the execution-flow graph. The current state in which a variable is evaluated is called the prestate, while the next state in which the same variable is evaluated is called the poststate. If the value of a variable can eventually influence the value of a controlled variable, then it is *live* in a state. Variables that are not live can

be removed without affecting the externally visible behaviour of the system. A backwards data-flow analysis is required for computing the set of live variables. First the poststate variables for each node is determined, and then rules for propagating liveness through the execution-flow graph are repeatedly applied until a fixed-point is reached. Input slicing may reduce the effectiveness of output slicing and in order to overcome this, input slicing is suppresed on paths through the execution-flow graph where a mode transition takes place.

Table 3 summarises the multi-tiered slicing approaches and categorises each algorithm according to its type. The type of some algorithms is *N/A*, which means that it does not belong to any of the kinds of algorithms identified for slicing FSMs because either it is not a slicing algorithm or it is trivial. For example, [RLHL06] have described a simplification algorithm, which is not a slicing algorithm. Similarly, the first algorithm presented by [HW97] is not a slicing algorithm. However, the adaptive slicing algorithm presented by [SPH08] is a slicing algorithm but it not categorised according to the kinds of algorithms because it is very simple as it just removes the function behaviour as this is independent from the adaptive behaviour.

# 6 Data Dependence for Slicing FSMs

There have been two general approaches to defining data dependence. The first is based on the idea that an element $x$ is required to evaluate $y$. For example, a variable $y$ is defined in terms of $x$. This is not limited to variables, but can be applied to other elements. For example, to execute a transition, a trigger event, source state and all ancestor states are required. We call this approach the *uses* approach from the uses relation defined by [HTW98] (see Definition 1).

The second is based on definition-clear paths of variables, i.e. a variable $v$ is defined in an element $x$ and used in an element $y$ and there exists a path from $x$ to $y$ where $v$ is not modified. We call this approach *definition-clear paths*. These types of definitions are given at different levels of granularity which could lead to more precise slices. For example, [Oja07] define data dependence between parts of transitions, rather than transitions, and slicing can remove these parts i.e. trigger events, guards or actions. We further divide these definitions according to whether they apply within an automaton or state machine (i.e. intra-automaton) or between parallel automaton or state machines (i.e. inter-automaton).

Table 4 groups the key papers on FSM slicing that define data dependence according to the classification that we have described. In the rest of this section we describe each data dependence definition in turn.

## 6.1 Uses Approach

[HTW98] have defined data dependence for RSML specifications as the set of elements required to determine the value of a particular variable, transition, function and macros (expressions in guards are defined as mathematical func-

Table 3: Key papers on multi-tiered slicing.

| **Slicing Approach** | **Algorithm** | **Slicing Criterion** | **Type of Algorithm** |
|---|---|---|---|
| Heimdahl and Whalen [HW97] for Slicing RSML Specifications | Interpretation Under a Scenario | Reduction Scenario (AND/OR table) | N/A (Checks for Infeasible Columns) |
| | a) Data Flow<br><br>b) Control Flow | Transition or Variable<br>Transition | Dependence Graph Based<br>FSM Traversal |
| [BFG00] for Slicing Extended Automata | Relevant Control Analysis | External Events (Set of Feeds) | Iteratively Applies Rules |
| | Relevant Variable Analysis | Test Purpose Output (Outputs of Implementation) | Dataflow Analysis Based |
| | Constraint Propagation | Constraints on the Feeds and Inputs | Dataflow Analysis Based |
| Schaefer and Poetzsch-Heffter [SPH08] for Slicing SAS Models | System Slicing | Variables Found in Property | Dataflow Analysis Based |
| | Module Slicing | Variables Found in Property | Dataflow Analysis Based |
| | Adaptive Slicing | Property Refers to Adaptation Bahaviour | N/A (Just Removes Functional Behaviour) |
| [RLHL06] for Slicing SCR Specifications | Input Slicing | Input and Header Node (Event) | Dependence Analysis Based |
| | Simplification | N/A | N/A (Reduces the Complexity of Expressions) |
| | Output Slicing | None | Dataflow Analysis Based |

Table 4: A classification of key papers that define data dependence for FSM slicing.

| **Uses** | **Definition-Clear Paths** | |
|---|---|---|
| [HTW98] | **Intra-Automaton** | **Inter-Automaton** |
| [CABN98] | [KSTV03] | [WDQ02] |
| [HKL$^+$98] | [Oja07] | [Lan06] |
| [FL05] | [LG08] | Janowska and |
| [SPH08] | [ACH$^+$09] | Janowski [JJ06] |

tions and macros are defined for frequently used conditions). Data dependence for variables, macros, and functions is simple as it uses the elements that are visible in the definition. While data dependence for transitions requires that transitions are dependent on their guarding conditions, their source state and all ancestors of the source state (because of state hierarchy).

**Definition 1 (Data Dependence for RSML)** *Let $A$ be the union of sets of states, transitions, variables, constants, functions and macros in an RSML specification. [HTW98] define data dependence by the relation uses, which is a mapping $A \mapsto A$ where $uses(x, y)$ means that $y$ is required to evaluate $x$.*

[FL05] and [CABN98] have defined a general notion of dependence that is similar to Definition 1. In addition, [FL05] state that a target state and an action of a transition $t$ depend on its source state, triggering event and guard.

[HKL+98] have defined a dependence relation $D$ that is also similar to Definition 1. It is defined between variables $r_i, r_j$ where $(r_i, r_j) \in D$ if $r_j$ or $r'_j$ is a parameter of the function defining $r'_i$. Primed variables are evaluated in the new state, while unprimed in the old state. In SCR, no circular dependencies in the definitions of state variables are allowed.

According to [SPH08], a variable in an assignment is data dependent on variables in the assigned expression.

## 6.2 Definition-Clear Paths Approach: Intra-Automaton

[KSTV03] have defined data dependence for EFSMs between variables on transitions. In particular, it is defined as a definition-clear path between a variable's definition at a transition $t_1$ and its use at transition $t_2$. This definition is also adopted by [ACH+09].

**Definition 2 (Data Dependence for EFSMs)** *Transition $t_2$ data dependends [KSTV03] on transition $t_2$ with respect to variable $v$ if:*

1. *$v \in D(t_1)$, where $D(t_1)$ is a set of variables defined by transition $t_1$, i.e. variables defined by actions and variables defined by the event of $t_1$ that are not redefined in any action of $t_1$;*
2. *$v \in U(t_2)$, where $U(t_1)$ is a set of variables used in a condition and actions of transition $t_1$;*
3. *there exists a path from $target(t_1)$ to $source(t_2)$ whereby $v$ is not modified.*

[LG08] have introduced a (intra-automata) data dependence definition for communicating automata (IOSTs). Similarly to Definition 2, they extend the definitions of *define* and *use* because variables can be used in transition guards and can be defined or used in actions (variable substitutions). Their extensions must also be able to handle communicating actions, which is not required for Definition 2. Definition 3 shows how $D(t)$ and $U(t)$ in Definition 2 is extended for communicating actions, i.e. what is required in addition.

**Definition 3 (Definition/Use for Communicating Automata)** *[LG08] Let $t$ be a transition and $v$ a variable. Then $v \in D(t)$ if an action $(c?v)$ is performed that causes the system to wait on some channel $c$ for the reception of a value to be assigned to $v$. $v \in U(t)$ if an action $(c!v)$ is performed that causes the system to emit a message having the argument $v$ on channel $c$.*

The data dependence definition defined by [LG08] is similar to Definition 2, with an additional condition (condition three in Definition 4) that is required because of the semantics of IOSTSs as it may be possible that $v$ is redefined at $t_2$ by a input action.

**Definition 4 (Data Dependence for Communicating Automata)** *A transition $t_2$ is data dependent [LG08] on a transition $t_1$ if there exists a variable $v$ such that:*

1. *$v \in D(t_1)$;*
2. *there exists a path $\pi$ from the $target(t_1)$ to the $source(t_2)$ where $v$ is not defined;*
3. *and one of the following is true: a) $v$ is used in the $guard(t_1)$; or b) $v$ is not defined at $action(t_1)$ and $v \in U(t)$ where $t \in \pi$.*

[Oja07] have defined data dependence for UML state machines between nodes in a CFG. The nodes of the CFG represent different parts of UML state machine transitions. Each transition can correspond to several nodes. Table 5 lists the different types of CFG nodes defined and what part of the transitions they represent. Also, the last column shows whether the nodes have a $D$ and $U$ set. $D$ is the set of variables that are defined by the part of the transition. $U$ is the set of variables that are used by the part of the transition. Ojala differentiates between variables that are defined when entering the state and those that are defined when exiting the state. $D$ and $U$ sets are used to define definition-clear paths between CFG nodes. This notion of data dependence is similar to Definition 10 as it applies to parts of transitions and also to concurrent state machines.

**Definition 5 (Data Depedence for UML state machines)** *Paths are defined as a sequence $a_1, .., a_i$ where each $a_j \in \{a_1, .., a_i\}$ is either a SIMPLE, SEND or end node, or b.e where b is a BRANCH node and e its element. Data dependence [Oja07] is defined in terms of definition-clear paths. Definition-clear paths with respect to variable $v$ are paths $p_1, p_2, ..., p_{(n-1)}, p_n$ where $v$ is defined at $p_1$ (i.e. $v \in D(p_1)$), $v$ is used at $p_n$ ($v \in U(p_n)$) and $v$ is not defined in $p_2, .., p_{(n-1)}$. The following definition-clear paths with respect to a variable $v$ are defined:*

- *between a SIMPLE node $s_1$ and a SIMPLE, SEND or end node $s_2$;*
- *between a SIMPLE node $s_1$ and an element $e$ in a BRANCH node $b$;*

| **CFG nodes** | **Part of Transition** | $D/U$ **sets** |
|---|---|---|
| BRANCH | Triggers and guards | $D$ = union of $D$ sets of its elements. $U$ = union of $U$ sets of its elementss. |
| SIMPLE | Actions | $D$, $U$ |
| SEND | Actions | $D$, $U$ |

| **Components of BRANCH node** | **Part of Transition** | $D/U$ **sets** |
|---|---|---|
| Element | Trigger and guard | $D$, $U$ |
| Parameter | Event parameter | $D$ |

Table 5: CFG nodes and what parts of a transition they represent. The sub-structure of the BRANCH node is given in a separate table. Note that the end node is of type BRANCH and has no sub-structure. $D$ is the set of variables that are defined by the part of the transition. $U$ is the set of variables that are used by the part of the transition.

- *between a parameter p (which is in an element and BRANCH) and a SIMPLE, SEND or end node* $s_2$*;*
- *between a parameter p in an element* $e1$ *in a BRANCH* $b1$ *and an ELEMENT* $e2$ *in a BRANCH node* $b2$*;*

*Also, a parameter p is data dependent on a node q if q evaluates an expression whose value gets assigned to p when an event is received. Node q is one of the CFG nodes that are created when a generated event occurs (either in the same UML state machine or in another).*

## 6.3 Definition-Clear Paths Approach: Inter-Automaton

[WDQ02] have defined three data dependence definitions for extended hierarchical automata (EHA): one for sequential automaton (not concurrent) and two for concurrent automaton. Note that variables are updated on states, rather than transitions. Since EHA are concurrent and hierarchical, the definition of $D$ and $U$ differs from Definition 2 and it applies to states rather than transitions. A variable $v$ is used at state $x$ (i.e. $v \in U(x)$) if $v$ is referenced in the actions of state $x$ or it is referenced in the actions of any sub-state or transitions in the sub-EHA of $x$ and can be defined and referenced outside the sub-EHA of $x$. The set of defined variables $D(x)$ at a state $x$ includes all variables that have been defined (have values assigned to them) in the actions in sub-EHA of $x$ and can be defined and referenced outside the sub-EHA of $x$. Internal variables of a state $x$ are local variables that can only be defined and used in sub-EHA of $x$. $U$ and $D$ can also apply to transitions.

**Definition 6 (Sequential Data Dependence for EHA)** *A state or transition y in a sequential automaton A that uses a variable v (i.e.* $v \in U(y)$*) is*

*sequential data dependent [WDQ02] on a state or transition $x$ in $A$ that defines $v$ (i.e. $v \in D(x)$) if there is path in $A$ from $x$ to $y$ where $v$ is not modified.*

Sequential data dependence is defined as a definition-clear path between states or transitions in the same sequential automaton. Parallel data dependence is defined as a definition-clear path between states and transitions in concurrent automata.

**Definition 7 (Parallel Data Dependence (PDD) for EHA)** *Let $A$ and $B$ be two different sequential automata, and $s_A$ is a state in $A$, $t_A$ is a transition in $A$, $s_B$ is a state in $B$ and $t_B$ is a transition in $B$. If $A$ and $B$ are sub-states of $C$ then $s_B$ is parallel data depended [WDQ02] on $s_A$ ($s_A \xrightarrow{PDD} s_B$) iff $U(s_A) \cap D(s_B) \neq \emptyset$. Similarly $s_A \xrightarrow{PDD} t_B$, or $t_A \xrightarrow{PDD} s_B$, or $t_A \xrightarrow{PDD} t_B$ iff $U(s_A) \cap D(t_B) \neq \emptyset$, or $U(t_A) \cap D(s_B) \neq \emptyset$, or $U(t_A) \cap D(t_B) \neq \emptyset$ respectively.*

Refinement data dependence [WDQ02] is defined between states and transitions of concurrent automata. Some state $x2$ is refinement dependent on $x1$, where $x2$ is in a sub-sequential automaton of an element $x1$, if the value of some variable computed at $x1$ is the value that $x2$ will return, or some event generated in $x1$ is used to synchronise with some concurrent state of $x2$. It differs from parallel data dependence as it computes dependencies in sub-states rather than across concurrent states.

**Definition 8 (Refinement Data Dependence (RDD) for EHA)** *If $GE(s)$ is the set of all events that are generated in the actions in sub-states of state $s$ and can be used as the trigger events of transitions outside of $s$. Similarly $GE(t)$, for a transition $t$, is the set of events generated in the action of $t$. If $A$ is a sequential automaton, $s_A$ is a state of $A$, $b$ is a superstate of $s_A$, $s_b$ is a state of $b$, and $t_b$ is a transition of $b$, then:*

- $s_A \xrightarrow{RDD} s_b$ *iff* $D(s_b) \cap D(s_A) \neq \emptyset$, *or* $GE(s_A) \cap GE(s_b) \neq \emptyset$.
- $s_A \xrightarrow{RDD} t_b$ *iff* $D(t_b) \cap D(s_A) \neq \emptyset$, *or* $GE(s_A) \cap GE(t_b) \neq \emptyset$.

[Lan06] adopts the dependence relations defined by [WDQ02] but points out in [LH07] that parallel data dependence can produce a false dependency if the execution chronology is not taken into account. To get rid of these false dependencies, their tool applies the following rule after calculating parallel data dependencies: $x \not\xrightarrow{PDD} y$ iff $x \xrightarrow{SO} y$, where $\xrightarrow{SO}$ is a Lamport-like [Lam78] happens-before relation on state machines. This happens-before relation is defined as using two other relations: the statechart concurrent relation and the statechart sequential order.

[Lan06] have also introduced a new data dependence relation called global data dependence in order for slicing a collection of statechart diagrams. By a collection of statechart models, they mean a statechart that may have many concurrent state machines at the top level. These are not dealt with by [WDQ02],

who only deals with concurrency in sub-states. Global data dependence is defined between states and transitions with respect to global variables. Global variables are variables used by statecharts to communicate and they don't belong to any statechart but can be accessed by all.

**Definition 9 (Global Data Dependence for EHA)** *Let $A$, $B$ be two different statecharts. A state $s_B \in B$ or a transition $t_B \in B$ is global data dependent [Lan06] on a state $s_A \in A$ or transition $t_A \in A$ ( $s_A \xrightarrow{GDD} s_B$, $s_A \xrightarrow{GDD} t_B$, $t_A \xrightarrow{GDD} s_B$, $t_A \xrightarrow{GDD} t_B$) iff:*

- *some global output variables (defined in actions) of a state or transition of $A$ are used in the input of the state or transition of $B$;*
- *or some trigger events of a transition or state of $B$ are generated by a state or transition of $A$.*

[JJ06] have defined data dependence between variables found in boolean expressions of guards and atomic assignments of transitions of timed automata. Compared to Definition 2 and Definition 4, it is of finer granularity, i.e. it applies to parts of transitions rather than transitions leading to slices that can remove parts of transitions. Also, it applies to transitions of a set of timed automata that run in parallel.

**Definition 10 (Data Dependence for Timed Automata)** *Let the atomic assignments of actions $x$ and the boolean expressions of guards $y$ of a transition $t$ be called operations ($opers(t)$), thus $x \in opers(t)$ and $y \in opers(t)$. Let $t_1 \in T_i$, $t_2 \in T_j$, where $i, j = 1, .., n$ and $n$ refers to the number of timed automata that run in parallel. An operation $a_2$ in $t_2$ is data dependent [JJ06] on operation $a_1$ in $t_2$ if there is a variable $v$, which occurs in $a_1$ and $a_2$, if $v \in D(a_1) \cap U(a_2)$ as in Definition 2 and one of the following holds:*

1. *$t_1 = t_2$, $a_2$ follows $a_1$ (actions are sequences so follows is well defined [JJ06]) and $v \notin D(a_3)$ for any $a_3 \in opers(t_1)$ between $a_1$ and $a_2$;*
2. *there exists a path from $target(t_1)$ to the $source(t_2)$ such that $v$ is not re-defined in any operations in transitions contained in the path;*
3. *$i \neq j$ i.e. the automata are different.*

# 7 Control Dependence for Slicing FSMs

One of the challenges facing any attempt to slice a FSM is the problem of how to correctly account for control dependence. It is common for state machines modelling such things as non-terminating systems not to have a final computation point or 'exit state'. Moreover, FSM are interactive, i.e. there is an interplay between the environment and the model, where events are generated by the environment and trigger transitions in the model. Thus it is not only

Table 6: Papers that are categorised according to the type of control dependence definitions.

| **Event-Flow** | **Transition/Guard** | **Program-like** |
|---|---|---|
| Heimdahl and Whalen [HW97] | [WDQ02] | [KSTV03] |
| [CABN98] | [Lan06] | [JJ03] |
| | [SPH08] | [Oja07] |
| | | [LG08] |
| | | [ACH+09] |

conditions that decide whether a transition occurs, but also whether a specific event occurs in the environment.

There have been three general approaches to defining control dependence. The first defines control dependence between events. We classify these definition as *Event-flow*. The second defines control dependence between transitions and guarding conditions and we classify these as *Transition/Guard*. The third defines control dependence similarly to how control dependence is defined by program slicing techniques. We call these definitions *Program-like*. The majority of the literature has focused on traditional definitions. In Table 6 the papers on FSM slicing are categorised according to the type of control dependence definition that they define.

## 7.1 Event-Flow Definitions

[HW97] were the first to define a control dependence-like definition for FSMs, in particular for RSML specifications. It differs from the traditional notion as it defines control flow in terms of dependency between events and generated events rather than as a structural property of the graph. Their definition can be applied to non-terminating systems or systems that have multiple exit nodes. However, it cannot be applied to any finite state machine, such as EFSMs, that do not generate events.

**Definition 11 (Control flow for RSML)** *Let $E$ be the set of all events and $T$ the set of all transitions. The relation $trigger(T \rightarrow E)$ represents the trigger event of a transition. The relation $action(T \rightarrow E^2)$ represents the set of events that make up the action caused by executing a transition. $follows(T \rightarrow T)$ is defined as: $(t_1, t_2) \in follows$ iff $trigger(t_1) \in action(t_2)$.*

[CABN98] use the same control dependence relation as in Definition 11.

## 7.2 Transition/Gaurd Definitions

[WDQ02] have defined two control dependence definitions for EHA that are adopted by [LH07]: transition control dependence and refinement control dependence. Transition control dependence is defined between a state and transi-

tions and can be applied to any FSM that may be non-terminating. They states that if a variable is defined in a state or transition $x$ and used in the guarding condition of a transition $t$ and not redefined on the path from $x$ to $t$, then $t$ is transition control dependent on $x$.

**Definition 12 (Transition Control Dependence (TCD))** *Let $A$ be an EHA automaton, $t$ a transition of $A$ and $CV(t)$ be set of variables reference in the guard of a transition $t$.*

1. *If a variable is defined in a state $v$ or a transition $r$ and used in the guard of $t$ and not redefined on the path from $v$ or $r$ to $t$, then $v \xrightarrow{TCD} t$ or $r \xrightarrow{TCD} t$.*

2. *If $B, C$ be two EHA automaton, $s$ is a state in $C$ and $A, B$ is a sub-state of $s$, $q$ is a state in $B$, $p$ is a transition in $B$, then $t \xrightarrow{TCD} p$ iff $CV(t) \cap D(q) \neq \emptyset$ (or $CV(t) \cap D(p) \neq \emptyset$).*

Refinement control dependence is defined between states of an automaton and its sub-sequential automaton. It can be applied to possibly non-terminating hierarchical FSMs.

**Definition 13 (Refinement Control Dependence)** *If state $v$ is the initial state of a direct sub-sequential automaton of state $u$, then $v$ is refinement control dependent on state $u$.*

These definitions of control dependence resemble data dependence, i.e. if the structure is flattened these correspond to data dependence as described in Definition 2. From the combined dependence relations defined by [WDQ02] (i.e. Definition 6, 7, 8, 12, 13 and 24), the hierarchical layer of an EHA for the sequential automaton of a state or transition can be determined. For example, if the sequential automaton of a state or transition $x$ is found on the $n^{th}$ layer, then $x$ depends on elements that belong to the sequential automata that are local in $(n-1)^{th}$ to $(n+1)^{th}$ layer of the EHA, which makes slicing efficient.

According to [SPH08], a variable assigned in a conditional assignment control depends on variables used in the conditions because these variables determine which branch of the condition is taken. This definition of control dependence also resembles data dependence.

## 7.3 Program-like Definitions

In program slicing, two general types of control dependence definitions are described: *non-termination insensitive* and *non-termination sensitive*. Traditional control dependence, as used in [HRB90] is non-termination *insensitive*, with the consequence that the semantics of a program slice dominates the semantics of the program from which it is; slicing may remove non-termination, but it will never introduce it. A non-termination sensitive formulation was proposed as early as 1993 by [Kam93], but has not been taken up in subsequent slicing

Table 7: Classifying which FSM slicing approaches adopt or are based on NTICD and NTSCD.

| **Non-Termination Sensitive (NTSCD)** | **Non-Termination Insensitive (NTICD)** |
|---|---|
| [JJ03] | [KSTV03] |
| [Oja07] (NTSCD & DOD) | [ACH+09] |
| [LG08] | |

research. Non-termination sensitive slicing tends to produce very large slices, because all iterative constructs that cannot be statically determined to terminate must be retained in the slice, no matter whether they have any effect other than termination on the values computed at the slicing criterion. These 'loop shells' must be retained in order to respect the definition of non-termination sensitivity.

In moving slicing from the program level to the state based model level, the choice of whether FSM slicing should be non-termination sensitive or insensitive needs to be made. Both types of control dependence have been defined in the literature for FSM slicing and we consider each of these. The choice of which to use depends on the application of slicing. For example, non-termination sensitive control dependence (NTSCD) is desired when slicing FSMs for the purpose of model checking, as loops are kept and liveness properties can still be checked. Non-termination insensitive control dependence (NTICD) is preferred when slicing FSM for model comprehension as this will produce smaller slices. Table 7 classifies the papers on FSM slicing according to whether their control dependence definitions are non-termination sensitive or insensitive.

The following definitions of control dependence are given in terms of execution paths. Since a path is commonly presented as a (possibly infinite) sequence of nodes, a node is in a path if it is in the sequence. A transition is in a path if its source state is in the path and its target state is both in the path and immediately follows its source state. A *maximal path* is any path that terminates in an end node or final transition, or is infinite.

[KSTV03] present a definition of control dependence for EFSMs in terms of post dominance that requires execution paths to lead to an exit state. This definition captures the traditional notion of control dependence for static backward slicing. However it can only determine control dependence for state machines with exactly one exit state, failing if there are multiple exit states or if the state machine is possibly non-terminating. For example, it can be applied to the ATM illustrated in Figure 5 but not to the ATM illustrated in Figure 6.

**Definition 14 (Post Dominance [KSTV03])** *Let $Y$ and $Z$ be two states and $T$ be an outgoing transition from $Y$.*

- *State $Z$ post-dominates state $Y$ iff $Z$ is in every path from $Y$ to an exit state.*

- *State $Z$ post-dominates transition $T$ iff $Z$ is on every path from $Y$ to the exit state though $T$. This can be rephrased as $Z$ post-dominates $target(T)$.*

**Definition 15 (Insensitive Control Dependence (ICD))** *Transition $T_k$ is control dependent [KSTV03] on transition $T_i$ if:*

1. *$source(T_k)$ post-dominates transition $T_i$ (or $target(T_i)$), and*
2. *$source(T_k)$ does not post-dominate $source(T_i)$.*

In program slicing, [RAB+05] define control dependence for arbitrary CFGs (with a start node) of non-terminating programs, i.e. that may not have an exit node. They give definition for both *non-termination sensitive* (NTSCD) and *non-termination insensitive* control dependence (NTICD). The difference between these definitions lies in the choice of paths. NTSCD is given in terms of maximal paths, while NTICD is given in terms of control sinks (see Definition 21). This seminal work has inspired control dependence definitions for FSM models hereafter, except for the definition by [JJ03]. We first discuss all the NTSCD definitions and then NTICD definitions.

In [JJ03, JJ06] a NTSCD definition of control dependence is given for potentially non-terminating timed automata. Although they were the first to give such a definition for FSMs which is similar to the definition in [RAB+05] (both in terms of maximal paths), they have not been widely cited by other FSM slicing approaches.

In [JJ06], control dependence is defined in terms of post-dominance between states, where post-dominance is defined in terms of maximal paths and does not require a unique exit state like in Definition 14.

**Definition 16 (Post Dominance [JJ06])** *Let $X_1$ and $X_2$ be two states. State $X_1$ post-dominates state $X_2$ iff every maximal path from $X_1$ goes though $X_2$.*

**Definition 17 (NTSCD-JJ [JJ06])** *A state $S_2$ is control dependent on a state $S_1$ ($S_1 \xrightarrow{NTSCD} S_2$) in the same automaton, if $S_1$ does not post dominate (using Definition 16) $S_2$ and there is a path $\pi$ from $S_1$ to $S_2$ such that every state, except for $S_1$, in $\pi$ post dominates $S_2$.*

The earlier definition of control dependence given in [JJ03], differs from that given in [JJ06] as it includes an additional clause that states that all outgoing transitions from $S_1$ must have non-trivial guarding conditions. Thus the definition in [JJ03] is similar to Definition 20 but for states, rather than transitions.

[Oja07] adopts the NTSCD and decisive order dependence definitions, as in [RAB+05], which is given between nodes in a CFG. NTSCD cannot capture certain dependencies within loops, and hence Ranganath et al. defined decisive order dependence (DOD).

**Definition 18 (NTSCD [RAB+05] )** *In a CFG, a node $n_j$ is non-termination sensitive control dependent on a node $n_i$ ($n_i \xrightarrow{NTSCD} n_j$) iff $n_i$ has at least two successors $n_k$ and $n_l$ such that: for all maximal paths $\pi$ from $n_k$, where $n_j \in \pi$; and there exists a maximal path $\pi_0$ from $n_l$ where $n_j \notin \pi_0$.*

**Definition 19 (Decisive Order Dependence (DOD))** *Two nodes $p_1$ and $p_2$ are decisively order dependent [RAB+05] on $n$ ($n \xrightarrow{DOD} p_1, p_2$) if:*

1. *all maximal paths from $n$ contain both $p_1$ and $p_2$,*
2. *$n$ has a successor from which all maximal paths contain $p_1$ before $p_2$,*
3. *$n$ has a successor from which all maximal paths contain $p_2$ before $p_1$.*

[LG08][1] adapt Ranganath et al.'s NTSCD definition for Input/Output Symbolic Transition Systems (IOSTS). It is given in terms of transitions in an IOSTS model, rather than in terms of nodes in a CFG. The first clause of Definition 20 concerning the non-triviality of guards is introduced in order to avoid a transition being control dependent on transitions that are executed non-deterministically even though they are NTSCD control dependent. This clause prevents this property from being a purely structural property on graphs.

**Definition 20 (NTSCD-LG [LG08])** *A transition $T_j$ is control dependent on a transition $T_i$ if $T_i$ has a sibling transition $T_k$ such that:*

1. *$T_i$ has a non-trivial guard, i.e. a guard whose value is not constant under all variable valuations;*
2. *for all maximal paths $\pi$ from $T_i$, the source of $T_j$ belongs to $\pi$;*
3. *there exists a maximal path $\pi_0$ from $T_k$ such that the source of $T_j$ does not belong to $\pi_0$.*

FSM models differ from CFGs is several ways. They can have multiple start and exit nodes, more than two edges between two states and more than two successors from a state. Moreover, in CFGs, decisions (Boolean conditions) are made at the predicate nodes while in state machines they are made on transitions. Labbé et al. take such differences into account when adapting NTSCD.

[ACH+09] have defined a new control dependence definition by extending Ranganath et al.'s NTICD definition and subsuming Korel et al.'s definition in order to capture a notion of control dependence for EFSMs that has the following properties. First, the definition is general in that it should be applicable to any reasonable FSM variant. Second, it is applicable to non-terminating FSMs and / or those that have multiple exit states. Third, by choosing FSM slicing to be non-termination insensitive (in order to coincide with traditional program slicing) it produces smaller slices than traditional non-termination sensitive slicing.

Following [RAB+05], the paths considered are sink-bounded paths, i.e. those that terminate in a control sink as in Definition 21. A control sink is a region of the graph which, once entered, is never left. These regions are always SCCs, even if only the trivial SCC, i.e. a single node with no successors.

[1] Labbé et al.'s definition of control dependence in [LGP07] differs slightly from Labbé et al. [LG08], so we evaluate the most recent.

Table 8: Comparison of transitive closure of control dependence definitions.

| Definition | Comparison of Transitive Closures |
|---|---|
| ICD [KSTV03] | ICD$^*$ $\subseteq$ NTSCD-JJ$^*$ |
| NTSCD-JJ [JJ06] | UNTICD$^*$ $\subseteq$ NTSCD-JJ$^*$ |
| NTSCD-LG [LG08] | NTSCD-LG$^*$ $\subseteq$ NTSCD-JJ$^*$ |
| UNTICD [ACH$^+$09] | UNTICD$^*$ $\subseteq$ NTICD-JJ$^*$ |

**Definition 21 (Control Sink)** *A control sink, $\mathcal{K}$, is a set of nodes that form a strongly connected component such that, for each node $n$ in $\mathcal{K}$ each successor of $n$ is in $\mathcal{K}$.*

Unlike NTICD, sink-bounded paths are unfair, i.e. we drop the fairness condition in the Ranganath et al.'s definition of sink paths. For non-terminating systems this means that control dependence can be computed within control sinks.

**Definition 22 (Unfair Sink-bounded Paths)** *A maximal path $\pi$ is sink-bounded iff there exists a control sink $\mathcal{K}$ such that $\pi$ contains a transition from $\mathcal{K}$.*

**Definition 23 (Unfair Non-termination Insensitive Control Dependence)** *(UNTICD) A transition $T_j$ is control dependent on a transition $T_i$ iff:*

1. *for all paths $\pi \in UnfairSinkPaths(target(T_i))$, the $source(T_j)$ belongs to $\pi$;*
2. *there exists a path $\pi \in UnfairSinkPaths(source(T_i))$ such that the $source(T_j)$ does not belong to $\pi$.*

UNTICD is in essence a version of NTICD modified to EFSMs (rather than CFGs) and given in terms of unfair sink-bounded paths.

Table 8 compares the transitive closure of control dependence definitions given in terms of transitions. Note that $ICD^*$ denotes the transitive closure of $ICD$. Similarly for $NTSCD-JJ^*$, $NTSCD-LG^*$, and $UNTICD^*$. NTSCD and DOD as given by [Oja07] are not in the table as they are defined between states, rather than transitions.

# 8 Interference Dependence for Slicing FSMs

Most FSM slicing approaches handle communication and synchronisation by introducing new dependencies, such as interference dependence. As with slicing concurrent programs, the computation of interference dependencies can be complex, if the possible orders of execution must be considered to compute *precise* dependencies [Kri03]. 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. As we discuss below, only a few FSM slicing approaches try to compute precise dependencies.

[WDQ02] have defined synchronisation dependence between transitions and states. It states that if the trigger event of some transition in an element (state or transition) $x$ is generated by the action of an element $y$ and the automaton which $x$ and $y$ belong to are concurrent, then $x$ is synchronisation dependent on $y$. However, their slicing algorithm traverses the dependencies (including the various data and control dependencies) and assumes transitivity and thus is imprecise.

**Definition 24 (Synchronisation Dependence (SD))** *A state $s_A$ or transition $t_A$ is synchronisation dependent on a concurrent state $s_B$ or transition $t_B$ ($s_A \xrightarrow{SD} s_B$, or $s_A \xrightarrow{SD} t_B$, or $t_A \xrightarrow{SD} s_B$, or $t_A \xrightarrow{SD} t_B$) iff some events generated by the latter are used as trigger events of the other.*

[LH07] have adopted Definition 24 but also introduce a new dependence relation for a collection of statecharts. Global synchronisation dependence is simililar to Definition 24 except that it is between statecharts and involves global generated events.

**Definition 25 (Global Synchronisation Dependence (GSD))** *Let $A$ and $B$ be two different statecharts. A state $s_A$ or transition $t_A$ is global synchronisation dependent on a state $s_B \in B$ or transition $t_B \in B$ ($s_A \xrightarrow{GSD} s_B$, or $s_A \xrightarrow{GSD} t_B$, or $t_A \xrightarrow{GSD} s_B$, or $t_A \xrightarrow{GSD} t_B$) iff some global events generated by the latter are used as the trigger of the other.*

They use a Lamport-like [Lam78] happens-before relation to ensure that the dependencies exist only between states and transitions where the source state or transition can happen before the target state or transition. Although this increases the precision of the dependence relations, the slicing algorithm is based on the (transitive) traversal of the dependencies and thus is imprecise.

[GR02] have also introduced dependencies to model and slice Argos specifications: *transition*, *hierarchy* and *trigger* edges model the dependencies between states and sub-states. Their slicing algorithm traverses the dependencies with no special handling of intransitivity.

[Oja07] have defined interference dependence for UML state machines between parts of transitions. This definition is similar to the one in [HCD+99] for multi-threaded Java programs. Their slicing algorithm is based on traversal of the dependencies and thus is imprecise.

**Definition 26 (Interference Dependence [Oja07])** *A node $n_j$ is interference dependent on a node or parameter $n_i$ if $v \in U(n_j)$, $w \in D(n_i)$, $v = w$ and the access to $v$ and $w$ are not local to $n_j$ or $n_i$ ($n_i$ and $n_j$ are in different state machines or different instances of the same state machine).*

[LG08] have defined a communication dependence relation for communicating automata (IOSTSs) that identifies dependencies owing to communicating

actions. These dependencies are inter-automata, unlike their data and control dependencies (Definition 4 and Definition 20) that are intra-automata. A communication dependence is defined between two transitions $t_1$ and $t_2$ in two different IOSTSs if there exists a channel that potentially allows a data or control flow to occur between $t_1$ and $t_2$.

**Definition 27 (Communication Dependence [LG08])** *Transitions $t_i$ and $t_j$ are communication dependent iff there exists a channel c such that:*

1. *The action $a_j = c?x$ occurs and the system waits on channel c for the reception of a value to be assigned to the attribute variable x; and b) the action $a_i = c!t$ occurs for the system to emit a message, with t as argument, on the channel c.*

2. *The action $a_j = c?$ occurs and the system waits for a signal to occur on the channel c; and b) The action $a_i = c!$ occurs and the system emits a message on channel c with no arguments.*

[LG08] have presented a slicing algorithm based on traversing the dependencies, however, as communication dependence is not transitive, they accept the reduced precision. [GR08] have compiled statecharts into Java programs which are then sliced dynamically. These types of approaches are different to those discussed in this section as they analyse concrete executions and reduce the machine according to a specific test case similar to dynamic program slicing. Because the synchronization and communication can directly be observed and don't have to be approximated by static analysis, concurrency and communication don't cause problems there.

# 9 Other Dependence Relations for Slicing FSMs

[RLHL06] have defined update dependence for SCR specifications to determine the set of variables that can change during a transition in which a particular monitored variable changes. A monitored variable is used to model sensors in the environment and used to detect changes.

**Definition 28 (Update dependence)** *The set of update dependencies for a variable x is the smallest set of variables such that at least one of those variables' value changes every time x changes. The update dependencies of x are always a subset of the new-state dependencies. Variable a is new-state dependent on a variable b in the new-state if $b'$ is used in the computation of $a'$. Note that a and b are evaluated in the current state, while $a'$ and $b'$ are evaluated in the next state.*

[JJ06] have defined two new dependencies: *clock* and *time* dependence. Their slicing algorithm is based on traversal of the dependencies and thus is imprecise. Clock dependence is defined between two states in the same timed automaton.

**Definition 29 (Clock Dependence)** *A state $s_2$ is clock dependent on a state $s_1$ in the same timed automaton if one of the following holds:*

1. *there exists a clock $x$ in the set of clocks of the automaton and a transition $t$, where $s_1 = source(t)$, such that $x$ is defined in the clock assignment of $t$ and is in set of clock of $s_2$, and $target(t) = s_2$, or*

2. *there exists a path $\pi$ from the $target(t)$ to $s_2$ such that the clock assignment for all transitions $t'$ contained in $\pi$ with respect to $x$ is equal to $x$.*

Time dependence is defined between two states, $s_1$ and $s_2$, of the same process if $s_2$ is reachable from $s_1$ and time has elapsed in $s_2$.

**Definition 30 (Time Dependence)** *For two states $s_1, s_2$ in the same automaton, $s_2$ is time dependent on $s_1$ if $s_2$ is reachable from $s_1$ and:*

1. *all transitions going out of $s_2$ (i.e. $source(t) = s_2$) are urgent and always one transition is enabled. Heuristics are used to check the last condition;*

2. *the state invariant has a constraint of the form $x = 0$, where $x$ is a clock and the clock assignments of all incoming transitions set $x$ to $x_0$.*

*If the given conditions are violated then time in state $s_2$ cannot lapse.*

[SPH08] have defined adaptive dependence, whereby a functional variable is influenced by the adaptive variables occurring in the configuration guards. The adaptive variables in the configuration guards determine whether the functional assignments are executed.

# 10 The Slicing Criterion and Applications of Slicing FSMs

In program slicing, the slicing criterion is a pair $c = (s, V)$ consisting of a statement $s$ and a subset $V$ of the program's variables. In state based model slicing, there is not just one type of slicing criterion. A slicing criterion can be a single or set of transitions, variables, states, events, actions or some combination of these.

One of the factors that affect the choice of slicing criterion is the type of FSM variant. For example, when slicing EFSMs [KSTV03] and [ACH+09] 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. On the contrary, in EHA actions (variable updates and event generation) occur at the states. Therefore, when slicing EHA, [WDQ02] 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 FSM slicing and we have broadly

categorised these into: (1) model comprehension, (2) model checking, (3) testing and (4) generating efficient code. 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 a transition or set of transitions and their variables. Table 9 lists the slicing criteria and applications for all FSM slicing approaches. In the following sections we discuss each group of applications in more detail.

## 10.1 Model Comprehension

Some FSM 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 FSM slicing was for helping manual review of system requirements of large RSML specifications [HW97, HTW98]. [HTW98] evaluated the effectiveness of slicing on TCAS II [HLR96], a collection of airborne devises that provide collision avoidance protection for commercial aircrafts. It consists of more than 300 states and 650 transitions. [HTW98] found that slicing reduced the specification, by removing states and transitions, from 68% to 90%.

The slicing algorithms described in [KSTV03] and [ACH+09] are used to reduce the size of EFSM specifications in order to enhance model comprehension. Empirical results, given in [AGH+09], 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 [BH03]. Note that a slice according to [ACH+09] consists of marked and unmarked transitions and its size, in terms of number of transitions and number of states, is not reduced. [KSTV03] did not provide any experimental results. Other slicing approaches for enhancing model comprehension are described in [FL05, LG08]. The slices produced are sub-model's of the original. Neither describe the correctness of slicing or provide data about the size of the slices. [GR02] have described an algorithm for slicing Argos specifications that can help with analysis, debugging and verification. They did not provide any experimental results but proved the correctness of their algorithm. They formally showed that for any input sequence, the behaviour of the slice up to state $S$ (given as the slicing criterion) is the same as the behaviour of the original up to state $S$ as far as the event $b$ is concerned ($b$ is also part of the slicing criterion).

Table 9: Approaches for slicing FSMs and their applications

| **Slicing Approaches for Model Comprehension** | | |
|---|---|---|
| **Approach** | **FSM Variant** | **Slicing Criterion** |
| [HW97] | RSML | A Transition or Variable |
| [KSTV03] | EFSMs | A Transition and its Variables |
| Fox and Luangsodsai [FL05] | Statecharts | Collection of States, Transitions, Actions, Variable Names |
| [LG08] | IOSTs | Set of Transitions |
| [ACH+09] | EFSMs | A Transition and its Variables |
| Ganapathy and Ramesh [GR02] | Argos, Lustre | A State and Output Signal (Generated Event) |
| **Slicing Approaches for Model Checking** | | |
| **Approach** | **Language** | **Slicing Criterion** |
| [CABN98] | RSML | States, Events, Transitions, or Inputs in Property |
| [HKL+98] | SCR | Set of Variable Names in Property |
| [Esh02] | UML Activity Diagrams | Variables and Events in Property |
| [WDQ02] | EHAs | States and Transitions in Property |
| [Lan06] | EHAs | States and Transitions in Property |
| [CCIP06] | UML Statecharts | Property Sequence Chart (Events) |
| Janowska and Janowski [JJ06] | Timed Automata | A Set of Variables and States in Property |
| [Oja07] | UML Statecharts | Set of Transitions |
| Schaefer and Poetzsch-Heffter [SPH08] | SAS | Variables in Property |
| **Slicing Approaches for Testing** | | |
| **Approach** | **FSM Variant** | **Slicing Criterion** |
| [BFG00] | Extended Automata | Test Purpose (Acyclic Finite Automata) and a Set of Feeds (Constrained Inputs) |
| Guo and Roychoudhury [GR08] | Java (Map to Statecharts) | Last State Visited by an Object when Error Occured |
| **Slicing Approaches for Generating Efficient Code** | | |
| **Approach** | **Language** | **Slicing Criterion** |
| [RLHL06] | SCR | Input Slicing: Input (Event) and Header Node (State). |

## 10.2 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 formulas or as automata. The system model is expressed as a FSM. Three types of FSMs are typically used [MOSS99]: 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 [CGP99]. 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 FSM. Cone of influence [CGP99] is a technique for reducing the size of the underlying FSM 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. [CABN98] point out that carrying out dependence analysis on the underlying FSM 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 level.

[MT98, MT99] have described an approach for slicing PROMELA, the input language for the SPIN [Hol97] model checker. PROMELA allows for nondeterminism 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. [MT98] 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 [Gra02] and the Wisconsin tool [HRsGR$^+$00] (both used for program slicing) the same. Since slicing is applied at the CFG of the input language and not on the underlying FSM 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.

[CABN98] have defined an algorithm for slicing of RSML models in order to reduce the model for model checking. They have experimentally evaluated their slicing approach on two models, the TCAS II model [HL95] and Boeing EDP case study [NB98]. Results show that TCAS II reduced the Boolean state variables by half for four of the five properties. Only one property required additional optimisations in order for model checking to be feasible. The reduction owing to slicing of the Boeing EDP 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. They have not formally shown the correctness of their approach.

In [HKL$^+$98, BH99], two abstraction methods were presented for handling the state space explosion problem when model checking SCR specifications with SPIN. No experimental results or proof of correctness of slicing with respect to properties has been given.

[WDQ02] and [Lan06] both have presented approaches for slicing Extended Hierarchical Automata (EHA) for reducing the complexity of verifying UML statechart models. A property $\phi$ to be model checked is given as a Linear-Time Temporal Logic (LTL) [CGP99] formula. [WDQ02] show that slicing with respect to the slicing criterion, which consists of the states and transitions in a property $\phi$, extracts a smaller EHA which is $\phi$-stuttering equivalent to the original EHA. This is defined by Theorem 10.1.

**Theorem 10.1 ([WDQ02])** *Given an EHA $H$ and a $LTL_{-X}$ formula $\phi$, where $LTL_{-X}$ is the subset of LTL formulas without the next time operator. Let $C_\phi$ be the slicing criterion corresponding to $\phi$, $H_s$ be the result of slicing $H$ with respect to $C_\phi$, and $M$ and $M_s$ be the LTSs of $H$ and $H_s$ respectively. Then $M$ and $M_s$ are $\phi$-stuttering equivalent, i.e. $M \sim_\phi M_s$ holds.*

Stuttering [Lam83] 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. $\phi$-stuttering equivalence means that on the property $\phi$, two Kripke structures have equivalent behaviour and are invariant under stuttering. [Lan06] have shown that a property is satisfied by the sliced model if and only if it is satisfied by the original model.

[EW04] have presented a slicing approach for reducing the state space of an UML activity hypergraph to help with model checking. [Esh02, EW04] have defined two reactive semantics for activity hypergraph: requirement-level semantics, that is based on STATEMATE semantics of statecharts [HN96], and implementation-level semantics, based on the OMG semantics of UML statecharts. [Esh02] has shown that for every reduction rule $r$, a property $\phi$ holds for the FSM of the original activity hypergraph iff it holds for the FSM of the sliced activity hypergraph.

[Oja07] 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.

[CCIP06] 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.

[JJ06] 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 proved, using bisimulation, that a model of a system satisfies a $CTL_{-X^*}$ [CE82] formula $\phi$, if and only if a model of its slice with respect to the set of propositions $P^{\phi}$ satisfies $\phi$.

[SPH08] have implemented a slicing approach for reducing the state space of SAS models for model checking. It consists of three slicing algorithms: system, module and adaptive slicing. It has been shown, using bisimulation, that if a property is true in the original model, then after applying any of the three types of slicing, the property will be true in the reduced system, and vice versa. Also, these algorithms have been evaluated experimentally by applying them to the adaptive vehicle stability control system, which consist of 20 modules. The results show that module slicing performs the best by reducing the system variables by 85%. In general, the size of the model (given as the average number of variables) decreases as a more detailed slicing technique is applied while the computation time increases. In the case where the number of interconnections in a system are high, then system slicing can take longer than module slicing, while in a more loosely coupled system, system slicing can out perform module slicing.

## 10.3 Testing

Slicing can be used to simplify specifications in order to help with testing. [BFG00] 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. Both specification and test purposes are described as labeled transition systems. This product can lead to the state space explosion problem arising. [BFG00] 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.

[BFG00] 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 techniquecan 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 techniquereduces 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 provde any experimental results for the third slicing technique as it was still under development. [BFG00] also define notions of correctness in terms of bisimulation for each of their slicing techniques but do not provide any proofs.

[GR08] have described a slicing approach used for debugging statecharts [Har87]. They have provided experimental results that show that the size of the slices at the model level is 27% to 47%. Their approach translates statecharts into Java programs and applies dynamic slicing at the program level and then translates the slices back into statecharts. Therefore, they also provide the size of the slices at the program level, which is 17% to 30%. They argue that the difference between the program level slices and model level slices is because a single model element may be implemented by several lines of code.

## 10.4 Generating Efficient Code

FSM slicing has also been used for helping with the automatic generation of efficient code from models. A problem with code generators is that they often produce inefficient code that can be slow to run. [RLHL06] have described a slicing approach for generating efficient code from SCR specifications. It consists of applying three techniques, namely, input slicing, simplification and output slicing before generating code. [RLHL06] have experimentally evaluated their approach by automatically generating code for seven benchmark specifications and comparing their running times. The best results were produced when all three techniques were applied. Input slicing produced the largest speedups in performance and output slicing and simplification were also effective in improving performance. None of the techniques lead to increasing the run times of the generated code.

# 11 Open Issues

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

## 11.1 Correctly Accounting for Control Dependence

Although there has been considerate effort in trying to correctly account for control dependence, there is still much work to be done. No slicing approach other than [Oja07] has considered decisive order dependence for irreducible graphs. However, their control dependence definition is as given by [RAB+07] for programs and cannot be applied to all FSMs. Moreover, in program slicing, [Amt07] has introduced a control dependence definition, called Weak Order Dependence (WOD), that applies to irreducible CFGs and is non-termination insensitive. It seems like a good candidate for use for slicing FSMs but has not been considered or adapted as yet. The issue of whether the guarding condition should be trivial should also be investigated.

## 11.2 Improving Precision of Algorithms

### 11.2.1 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. [GR02] 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.

### 11.2.2 Concurrency and Communication

Slicing concurrent and communicating state machines has to solve two problems. First, it has to handle the communication and synchronisation when slicing. Second, the slicing algorithm itself has to handle them. 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 [Kri98]. Only a few FSM slicing approaches try to compute precise dependencies, such as in [Lan06], and none actually compute precise slices. Hence there is scope for

Table 10: The FSM 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] | **EE**[g] |
|---|---|---|---|---|---|---|---|
| [LG08] | √ | √ | √ | √ | √ | √ | √ |
| [ACH+09] | √ | √ | × | √ | √ | √ | √ |
| [HKL+98] | × | √ | √ | √ | √ | √ | √ |
| [HW97] | × | × | √ | √ | √ | √ | √ |
| [CABN98] | × | × | √ | √ | √ | √ | √ |
| [KSTV03] | × | × | √ | √ | √ | √ | √ |
| [GR02] | × | × | √ | √ | √ | √ | √ |
| [CCIP06] | × | × | √ | √ | √ | √ | √ |
| [WDQ02] | × | × | √ | √ | √ | × | √ |
| [FL05] | × | × | √ | √ | √ | × | √ |
| [Lan06] | × | × | √ | √ | √ | × | √ |
| [JJ06] | × | × | √ | √ | √ | × | √ |
| [SPH08] | × | × | √ | √ | √ | × | √ |
| [BFG00] | × | × | √ | √ | √ | × | √ |
| [Esh02] | × | √ | √ | √ | × | × | × |
| [GR08] | × | × | √ | √ | × | × | √ |
| [RLHL06] | × | √ | √ | √ | √ | × | √ |
| [Oja07] | √ | √ | √ | × | × | × | √ |

[a] S = States
[b] T = Transitions
[c] L = Labels
[d] TE = Triggering Events
[e] G = Guards
[f] A = Actions
[g] EE = External/Environmental Events

further work in improving algorithms to produce precise slices for concurrent FSMs.

## 11.3 Graph Connectivity

The FSM elements that are kept and removed in a slice varies from one slicing approach to another. Table 10 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 example, [LG08] do not remove any elements but simply mark those that are in the slice. [HW97] produces slices by deleting states and transitions. [Oja07] only deletes parts of transitions trigger events, guards and actions.

Consider the set of slicing approaches that remove transitions or states. What is not shown in Table 10 is whether removing transitions or states can lead to breaking the connectivity of the model i.e. some states become un-

reachable. For example, consider the state machine in Figure 16 that models a simplified version of the ATM as shown in Figure 5. Let us slice the state machine in Figure 16 using the algorithm given by [KSTV03] and the slicing criterion $< T4, x >$. The first step is to compute the data and control dependencies and mark the transitions that are backwardly reachable in the dependence graph. Since the state machine is non-terminating, Definition 15 cannot be applied. Assume we use Definition 23 instead. There are no UNTICD dependencies for Figure 16. The following transitions in Figure 16 are data dependent: $T1 \xrightarrow{DD} T3$, $T1 \xrightarrow{DD} T4$, $T3 \xrightarrow{DD} T4$ and $T4 \xrightarrow{DD} T3$. Figure 17 illustrates the transitions that are marked (in bold) after computing dependencies. If we choose to simply remove the unmarked transitions, then state $s3$ will become unreachable. Most slicing approaches only delete transitions or states that do not make other states or transitions to become unreachable. This leads to larger, less precise slices. Only [KSTV03] have described an algorithm (their second algorithm described in Section 5.6) for removing transitions and reconnecting the state machine by merging states. For example, Figure 18 shows the slice generated for the state machine shown in Figure 16.

However, there is still much work to be done. First, [KSTV03]'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, the slices could introduce additional behaviour that is not in the original state machine. This could lead to problems with proving the correctness of the slicing algorithm. For example, the semantics of a state machine assumes that events produced by the environment that do not trigger a transition are consumed. This is also called stuttering. Then if the environment of the state machine in Figure 16 generated the following event sequence *enterCard*, *ChooseLanguage*, *withdraw*(10), *deposit*(20), *done*, *withdraw*(30) in Figure 18, then $x = 70$ because event *deposit*(20) occurs while the state machine is in state $S4$ and no transition is trigger so it is consumed i.e. it stutters. If we apply the same sequence to the slice, then $x = 80$ as *deposit*(20) triggers a transition. According to Weiser's notion of correctness [Wei79] this slicing algorithm is incorrect. [KSTV03] 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 problems have not been ironed-out. Further work is required in developing slicing algorithm that improve on just reachability and handle to graph connectivity.

## 11.4 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

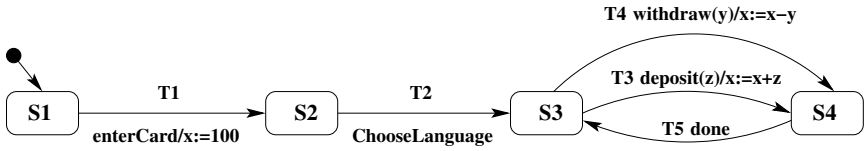



Figure 16: A simplified state machine of the ATM system as illustrated in Figure 5.

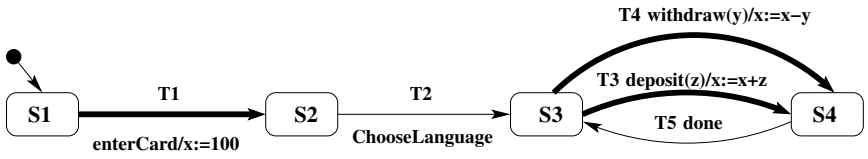



Figure 17: After computing the dependence graph using Definition 2 (data dependence) and Definition 23 (control dependence), transitions that are backwardly reachable from the slicing criterion in the dependence graph are marked. Marked transitions are in bold.

notations this state is modelled as a superstate. Only [SPH08] have defined slicing algorithms that apply at different levels of detail, while most other approaches, such as [WDQ02], [KSTV03] and [LG08] 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 definitions presented in Section 6, as dependencies may occur between different actions of a transition. This research problem has not been addressed in the literature.

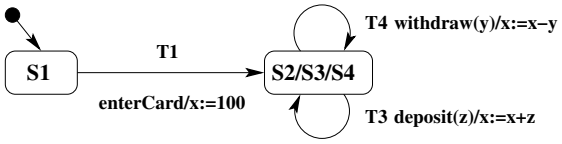



Figure 18: The slice generated using [KSTV03]'s second slicing algorithm.

### 11.5 Slicing Richer FSMs

Only some of the features of some FSMs have been considered by slicing approaches, such as hierarchy, concurrency, communication and event generation. However, some features of richer FSMs, such as UML, have not been considered. These include: the condition and selection circled connectives, delays and timeouts, the entry/exit activity and histories. Slicing approaches could be developed further so that they can be applied to any FSM.

## 12 Conclusions

This paper is the first to survey existing work on slicing finite state machines. It comprehensively reviewed slicing approaches and describes a taxonomy for the types of algorithms and the kinds of dependence used. 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 [HW97] while the program slicing community only addressed the problem of slicing non-terminating programs in [RAB+07]. 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.

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

[ACH+09] Kelly Androutsopoulos†, David Clark, Mark Harman, Zheng Li, and Laurence Tratt. 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*, York, UK, March 2009. Springer Berlin/Heidelberg.

[AD90] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In $4^{th}$ *ACM Symposium on Testing, Analysis, and Verification (TAV4)*, pages 60–73, 1991. Appears as Purdue University Technical Report SERC-TR-93-P.

[AGH+09] Kelly Androutsopoulos*, Nicolas Gold, Mark Harman, Zheng Li, and Laurence Tratt. A theoretical and empirical study of EFSM dependence. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, September 2009.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, New York, June 1990.

[Amt07] Torben Amtoft. Correctness of practical slicing for modern program structures. Technical Report 2007-3, Department of Computing and Information Sciences, Kansas State University, 2007.

[ASS+07] R. Adler, I. Schaefer, T. Schuele, , and E. Vecchi. From model-based design to formal verification of adaptive embedded systems. In *International Conference on Formal Engineering Methods (ICFEM'07)*, Florida, USA, 2007. Springer Berlin/Heidelberg.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, (Pearson Education), 1986.

[BFG+99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean pierre Krimm, Laurent Mounier, and Joseph Sifakis. If: An intermediate representation for sdl and its applications. In *Proceedings of SDL-FORUM99*, pages 423–440, Montreal, Canada, 1999. Elsevier Science.

[BFG00] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. 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*, pages 235–250, London, UK, 2000. Springer-Verlag.

[BG96] David Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control–flow. In Peter Fritzson, editor, $1^{st}$ *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993. Springer.

[BH99] Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Eng.*, 6(1):37–68, 1999.

[BH03] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance*, pages 44–53, Los Alamitos, California, USA, September 2003. IEEE Computer Society Press.

[BH04] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[BH05] David Binkley and Mark Harman. Forward slices are smaller than backward slices. In $5^{th}$ *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 15–24. IEEE Computer Society Press, 2005.

[BHK07] David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems*, 30(1), November 2007.

[BHR95] David Binkley, Susan Horwitz, and Tom Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[Bin98] David Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11):583–594, 1998.

[Bin07] David Wendell Binkley. Source code analysis: A road map. In Lionel Briand and Alexander Wolf, editors, *Future of Software Engineering 2007*, pages 104–119, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[CAB+01] William Chan, Richard J. Anderson, Paul Beame, David Notkin, David H. Jones, and William E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Trans. Softw. Eng.*, 27(2):170–190, 2001.

[CABN98] William Chan, Richard J. Anderson, Paul Beame, and David Notkin. Improving efficiency of symbolic model checking for state-based system requirements. *SIGSOFT Software Engineering Notes*, 23(2):102–112, 1998.

[CCD98] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11):595–607, 1998.

[CCIP06] Daniela Colangelo, Daniele Compare, Paola Inverardi, and Patrizio Pelliccione. 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*, pages 243–258, Paris, France, 2006. Springer Berlin/ Heidelberg.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In $22^{nd}$ *International Conference on Software Engineering (ICSE'2000)*, pages 439–448, Los Alamitos, California, USA, June 2000. IEEE Computer Society Press.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. CMIT Press, 1999.

[De 01] Andrea De Lucia. Program slicing: Methods and applications. In *International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Los Alamitos, California, USA, 2001. IEEE Computer Society Press.

[DHH+06] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Ranganath, and Todd Wallentine. 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*, pages 73–89. Springer, 2006.

[Dub06] Jori Dubrovin. 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 2006.

[DWQQ01] Wei Dong, Ji Wang, Xuan Qi, and Zhi-Chang Qi. Model checking UML statecharts. In *APSEC '01: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, page 363, Washington, DC, USA, 2001. IEEE Computer Society.

[Esh02] Rik Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modeling*. PhD thesis, Centre for telematics and Information Technology, University of Twente, 2002.

[EW04] Rik Eshuis and Roel Wieringa. Tool support for verifying uml activity diagrams. *IEEE Transactions on Software Engineering*, 30:2004, 2004.

[FDHH04] Chris Fox, Sebastian Danicic, Mark Harman, and Robert Mark Hierons. ConSIT: a fully automated conditioned program slicer. *Software Practice and Experience*, 34:15–46, 2004.

[FHHD01] Chris Fox, Mark Harman, Rob Mark Hierons, and Sebastian Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. In $9^{th}$ *IEEE International Workshop on Program Comprenhesion*, pages 89–97, Los Alamitos, California, USA, May 2001. IEEE Computer Society Press.

[FL05] Chris Fox and Arthorn Luangsodsai. And-or dependence graphs for slicing statecharts. In *Beyond Program Slicing*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FRT95] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In $22^{nd}$ *ACM Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, CA, 1995. ACM.

[GGRT06] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *Testing of Communicating Systems (TestCom'06), Lecture Notes in Computer Science*, volume 3964, pages 1–18, New York, NY, USA,, May 16-18 2006. Springer.

[GHS92] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, Los Alamitos, California, USA, 1992. IEEE Computer Society Press.

[GL91] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[GR02] Vinod Ganapathy and S. Ramesh. Slicing synchronous reactive programs. In *Electronic Notes in Theoretical Computer Science, 65(5). 1st Workshop on Synchronous Languages, Applications, and Programming*, Grenoble, France, April 2002. Elsevier.

[GR08] Liang Guo and Abhik Roychoudhury. Debugging statecharts via model-code traceability. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008*, pages 292–306, Port Sani, Greece, 2008. Springer Berlin/Heidelberg.

[Gra02] Grammatech Inc. The CodeSurfer slicing system, 2002.

[GS95] Rajiv Gupta and Mary Lou Soffa. 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*, pages 29–40, October 1995.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HBD03] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, October 2003.

[HBGL95] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance*, pages 109–122, Gaithersburg, MD, June 1995. Springer Berlin/Heidelberg.

[HCD+99] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Symposium on Static Analysis*, pages 1–18, London, UK, 1999. Springer-Verlag.

[HD98] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance and Evolution*, 10(6):415–441, 1998.

[Hei02] C. Heitmeyer. Software cost reduction. In John J. Marciniak, editor, *Encyclopedia of Software Engineering, Two Volumes*, New York, 2002. John Wiley and Sons, Inc.

[HH01] Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[HHD+01] Mark Harman, Rob Mark Hierons, Sebastian Danicic, John Howroyd, and Chris Fox. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 138–147, Los Alamitos, California, USA, November 2001. IEEE Computer Society Press.

[HHH+04] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, Andr Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[HKL+98] Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Softw. Eng.*, 24(11):927–948, 1998.

[HL95] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency analysis of state-based requirements. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 3–14, New York, NY, USA, 1995. ACM.

[HLR96] M. P. E. Heimdahl, N.G. Leveson, and J. D. Reese. Experiences and lessons from the analysis of TCAS II. *SIGSOFT Softw. Eng. Notes*, 21(3):79–83, 1996.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[HRsGR+00] Susan Horwitz, Thomas Reps, Manuvir Das, Rebecca Hasti, Jeff Lampert, Dave Melski, Marc Shapiro, Mike Siff, Todd Turnidge, staff Binkley, Victor Barger, Samuel Bates, Thomas Bricker, Jiazhen Cai, Robert Paige, Phil Pfeiffer, Jan Prins, Wuu Yang, G. Ramalingam, and Mooly Sagiv. Wisonsin program slicing project, 1996-2000. URL `http://www.cs.wisc.edu/wpis/html/`.

[HTW98] Mats P. E. Heimdahl, Jeffrey M. Thompson, and Michael W. Whalen. On the effectiveness of slicing hierarchical state machines: A case study. In *EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO*, pages 10435–10444, Washington, DC, USA, 1998. IEEE Computer Society.

[HW97] Mats P. E. Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *Proc. Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, 1997. Springer–Verlag.

[JJ03] Agata Janowska and PawełJanowski. Slicing timed systems. *Fundam. Inf.*, 60(1-4):187–210, 2003.

[JJ06] Agata Janowska and PawełJanowski. Slicing of timed automata with discrete data. *Fundamenta Informaticae, SPECIAL ISSUE ON CONCURRENCY SPECIFICATION AND PROGRAMMING*, 72(1-3):181–195, 2006.

[JM05] Ranjit Jhala and Rupak Majumdar. Path slicing. *SIGPLAN Not.*, 40(6):38–47, 2005.

[Kam93] Mariam Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing.* PhD thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993.

[KH00] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 155–169, N.Y., January 19–21 2000. ACM Press.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218, NY, USA, 1981. ACM.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[KM02] Alexander Knapp and Stephan Merz. Model checking and code generation for UML state machines and collaborations. In Dominik Haneberg, Gerhard Schellhorn, and Wolfgang Reif, editors, *Proceding 5th Workshop on Tools for System Design and Verification (FM-Tools)*, pages 59–64, 2002.

[Kri98] Jens Krinke. Static slicing of threaded programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM New York, NY, USA.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs.* PhD thesis, Universität Passau, July 2003.

[KSTV03] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state based models. In *IEEE International Conference on Software Maintenance (ICSM'03)*, pages 34–43, Los Alamitos, California, USA, September 2003. IEEE Computer Society Press.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam83] L. Lamport. What good is temporal logic? In *Information Processing 83: Proceedings of the IFIP 9th World Congress, R. E. A. Mason, Ed.*, pages 657–668, North-Holland, Amsterdam, 1983.

[Lan06] Sara Van Langenhove. *Towards the Correctness of Software Behavior in UML A Model Checking Approach based on Slicing*. PhD thesis, Ghent University, May 2006.

[LG08] Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing*, 20(6):563–595, 2008.

[LGP07] Sebastien Labbe, Jean-Pierre Gallois, and Marc Pouzet. Slicing communicating automata specifications for efficient model reduction. In *Proceedings of ASWEC*, pages 191–200, USA, 2007. IEEE Computer Society.

[LH07] Sara Van Langenhove and Albert Hoogewijs. $SV_t$L: System verification through logic tool support for verifying sliced hierarchical statecharts. In *Lecture Notes in Computer Science, Recent Trends in Algebraic Development Techniques*, pages 142–155, Berlin / Heidelberg, 2007. Springer.

[LHHR94] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, 1994.

[LS03] Arun Lakhotia and P. Singh. Challenges in getting formal with viruses. *virus bulletin*, pages 14–18, September 2003.

[MACE02] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. 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)*, pages 71–80, New York, November 2002. ACM Press.

[Mar91] F. Maraninchi. The Argos language: graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Kobe, Japan, 1991. IEEE.

[Mea55] George H. Mealy. A method to synthesizing sequential circuits. *Bell Systems Technical Journal*, 34(5):1045–1075, 1955.

[Moo56] Edward F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton U., New Jersey, 1956.

[MOSS99] Markus Muller-Olm, David Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In *Proceedings of the 6th International Static Analysis Symposium*, volume 1694, pages 331–354, September 1999.

[MT98] L. Millett and T. Teitelbaum. Slicing promela and its applications to model checking, 1998.

[MT99] Lynette I. Millett and Tim Teitelbaum. Channel dependence analysis for slicing promela. In *PDSE '99: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 52, Washington, DC, USA, 1999. IEEE Computer Society.

[NB98] C.R. Nobe and M.G. Bingle. Model-based development: Five processes used at boeing. In *IEEE International Conference and Workshop: Engineering of Computer-Based Systems*, 1998.

[Oja07] Vesa Ojala. A slicer for UML state machines. Technical Report HUT-TCS-25, Helsinki University of Technology Laboratory for Theoretical Computer Science, Espoo, Finland, 2007.

[OMG01] OMG. OMG unified modeling language specification 1.4, September 2001. `http://www.omg.org/cgi-bin/doc?formal/01-09-67`.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. 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, 1984.

[PMBF05] Patrizio Pelliccione, Henry Muccini, Antonio Bucchiarone, and Fabrizio Facchini. Testor: Deriving test sequences from model-based specifications. In *Eighth International SIGSOFT Symposium on Component-based Software Engineering*, pages 267–282, St. Louis, Missouri (USA), May 2005. LNCS 3489.

[RAB+05] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *Programming Languages and Systems, Proceedings of 14th European Symposium on Programming, ESOP*, pages 77–93, Berlin, 2005. Springer-Verlag.

[RAB+07] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems*, 29(5):27, 2007.

[RLHL06] Tom Rothamel, Yanhong A. Liu, Constance L. Heitmeyer, and Elizabeth I. Leonard. Generating optimized code from scr specifications. *SIGPLAN Not.*, 41(7):135–144, 2006.

[SH96] Anthony M. Sloane and Jason Holdsworth. Beyond traditional program slicing. In Steven J. Zeil, editor, *Proceedings of the 1996 International Symposium on Software Testing and analysis*, pages 180–186, New York, January 8–10 1996. ACM Press.

[SPH08] Ina Schaefer and Arnd Poetzsch-Heffter. Slicing for model reduction in adaptive embedded systems development. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 25–32, NY, USA, 2008. ACM.

[TAFJ07] Mario Trapp, Rasmus Adler, Marc Förster, and Janosch Junger. Runtime adaptation in safety-critical automotive systems. In *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*, pages 308–315, Anaheim, CA, USA, 2007. ACTA Press.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119, New York, NY, USA, 1991. ACM.

[War03] Martin Ward. Slicing the SCAM mug: A case study in semantic slicing. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 88–97, Los Alamitos, California, USA, September 2003. IEEE Computer Society Press.

[WDQ02] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing hierarchical automata for model checking UML statecharts. In *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM)*, pages 435–446, UK, 2002. Springer-Verlag.

[Wei79] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[WRG08] T. Wang, A. Roychoudhury, and L. Guo. JSlice, version 2.0, 2008.

[WZ07] Martin Ward and Hussein Zedan. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems*, 29(2):7, 2007.

[XQZ+05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.