# Amorphous Slicing of Extended Finite State Machines

Kelly Androutsopoulos[1], David Clark[1], Mark Harman[1], Robert M. Hierons[2], Zheng Li[3], Laurence Tratt[4]

**Abstract**—Slicing is useful for many Software Engineering applications and has been widely studied for three decades, but there has been comparatively little work on slicing Extended Finite State Machines (EFSMs). This paper introduces a set of dependency based EFSM slicing algorithms and an accompanying tool. We demonstrate that our algorithms are suitable for dependence based slicing. We use our tool to conduct experiments on ten EFSMs, including benchmarks and industrial EFSMs. Ours is the first empirical study of dependence based program slicing for EFSMs. Compared to the only previously published dependence based algorithm, our average slice is smaller 40% of the time and larger only 10% of the time, with an average slice size of 35% for termination insensitive slicing.

**Index Terms**—Slicing, Extended Finite State Machines.

## 1 Introduction

The idea of slicing is simple yet fundamental: software contains many intricately interleaved computations and it is often helpful to extract one of these, leaving the others behind. Furthermore, the process of slicing out a sub-computation can be fully automated, thereby allowing a developer to isolate the portion of software that is of specific interest.

Since Weiser introduced program slicing [59], [60], it has been widely studied, adapted, and applied. There are several surveys on program slicing [10], [13], [56], illustrating the way in which slicing can be applied to many areas within software engineering (e.g. maintenance [22], testing [9], [11], [47], and refactoring [41]).

In recent years, modelling has come to play an important part in the software developer's toolbox, with applications ranging from testing [8], [24], intrusion detection [49], feature interaction [6] and quality of service improvement [17].

Extended Finite State Machines (EFSMs) and their variants (e.g. Harel's [26] and UML state machines) are now widely used to model dynamic behaviour, most often in embedded systems (40 billion of which are expected to be in operation by 2020 [20]). EFSMs are often used to model a system at a higher-level of abstraction than a program; nevertheless, like a program, an EFSM may contain many interleaved computations and so it makes sense to consider ways of slicing out sub-EFSMs. However, EFSM slicing presents new challenges and issues: algorithms and analyses that apply to *program* slicing are not fully applicable to *EFSM* slicing.


- [1]*University College London, CREST, Department of Computer Science, Gower Street London, WC1E 6BT, United Kingdom. E-mail: {k.androutsopoulos, d.clark, m.harman}@cs.ucl.ac.uk*
- [2]*Brunel University, Uxbridge, Middlesex, UB8 3PH, United Kingdom. Email: rob.hierons@brunel.ac.uk*
- [3]*Beijing University of Chemical Technology, 15 BeiSanhuan East Road, ChaoYang District, Beijing, 100029, China. Email: z.li@ieee.org*
- [4]*King's College London, Strand, London, WC2R 2LS, United Kingdom. Email: laurie@tratt.net*


### 1.1 EFSMs

Many readers may have an intuitive notion of Extended Finite State Machines (EFSMs) or one of their variants. In this section we provide a brief introduction and refresher; Section 2 contains a complete, formal definition.

EFSMs are diagrammatic representations of systems that have distinct *states* and *transitions* between those states; at any given time the system is in exactly one state. The external world can generate an event which is received by the system and which *triggers* a transition to another state.

Figure 1 shows an example representing the door control of a lift system [54], waiting for passengers to enter or leave the lift and shutting the door. Initially a timer is set for five time units then run down then the door begins closing. During this period, while it is closing, it can be interrupted. This sets the timer to 3 units and the door begins opening. Once it is opened the timer runs down and the door begins closing again until it is fully closed. Once it receives the open signal it sets the timer to ten units and and begins opening. When fully open the timer is run down and then the door begins closing again. States are represented by circles, with the start state highlighted with a thicker outline (in this case it is also called 'start', though it need not be in general). Transitions are directed arrows between *source* and *target* states (where the source and target may be the same) and specify under what conditions the system will move from one state to another. Transitions can have a *label* of the form `identifier:event[guard]/action`,

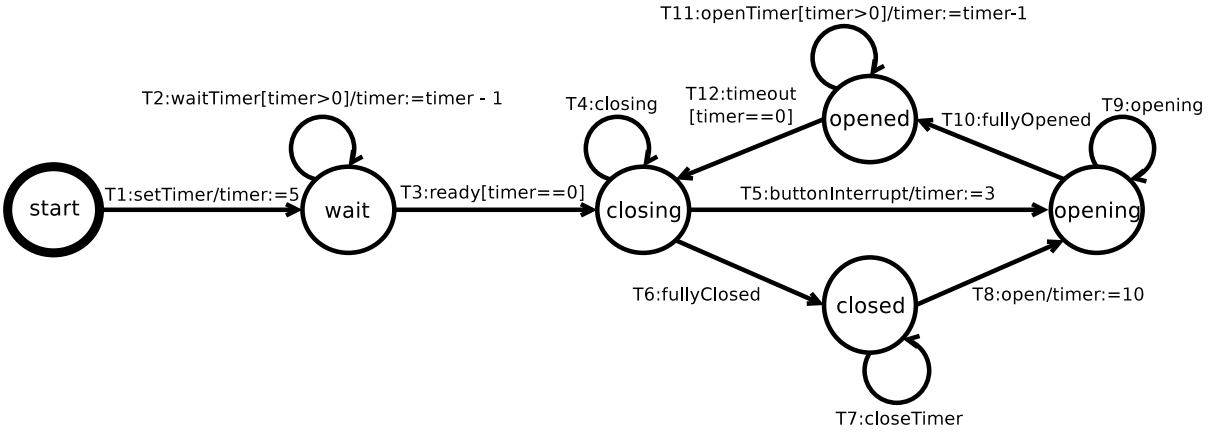


Fig. 1. An EFSM specification for the door controller of the lift system.

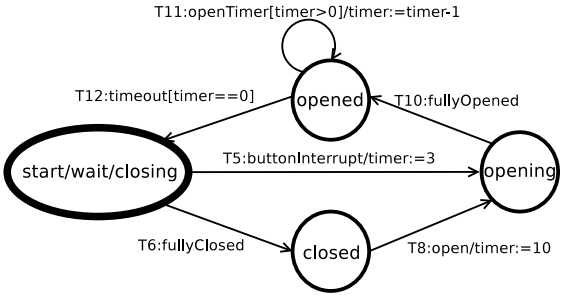


Fig. 2. The slice generated for Figure 1 using UNTICD and the slicing criterion (T11, {timer}).

each part of which is optional. A transition from a given source state is only triggered if an event `event` is received and the conditional `guard` evaluates to true; once triggered, the statements of `action` will be executed. EFSMs have a global store which guards can read and actions can update. In the lift example, a global `timer` variable is used.

EFSMs are useful because they give a clear visual representation of many types of behaviour, and because they are not necessarily expected to represent every detail of execution—in other words, they are frequently used to represent systems at a level of abstraction above a program. The lift example of Figure 1 does not specify any exit states since it is chiefly intended to represent its core behaviour and this in turn is intended to execute indefinitely. However, EFSMs can be used to model completely executable systems, and some embedded systems' behaviour is fully specified using EFSMs.

### 1.2 Slicing EFSMs

Slicing comes in several different flavours: in this paper we consider amorphous slicing [28], that is slicing which in some way rewrites an input program (i.e. changes its appearance). As in program slicing, EFSM slicing takes as input an EFSM and a *slicing criterion* and produces as output a sliced EFSM. The slicing criterion is the point of interest in the EFSM to be used as the basis for slicing: in this paper we use a criterion consisting of a transition and a set of variables. A control dependence formulation then determines which transitions are relevant to the slicing criterion, and a slicing algorithm then rewrites the EFSM to produce a slice. As a first example, Figure 2 shows a slice of the model from Figure 1 using the UNTICD control dependence algorithm [4], the slicing algorithm defined in Section 6.3, and the slicing criterion (T11, {timer}). It represents the part of the model in figure 1 that influences the value of the timer after transition T11 in that figure is executed.

While EFSM slicing has some relation to program slicing[1], there are substantial differences. At a high-level, these can be categorised as follows (see Section 3 for more details):

**Syntactic issues:** Unlike program dependence graphs, computation within EFSMs occurs on transitions, not within nodes. Calculating dependence is therefore subtly different. More significantly, deleting transitions is a more involved process than deleting statements in a program. Where program slicing can simply remove whole lines, EFSM slicing must rewire the EFSM (i.e. change the source and target of transitions). As we show in this paper, a naive rewiring can lead to slices which are larger than the original model.

This then raises the question of what the goal of EFSM slicing is. With program slicing this is uncontroversial: slicing seeks to remove as many statements as it can while respecting the slicing criterion. For EFSMs, there are several possible measures that could be used separately or combined to measure a reduction in size such as: the number of states, number of transitions, or number of uniquely labelled transitions.

1. If EFSM slicing can be thought of as similar to program slicing at all, then it might be thought of as akin to slicing pointer-free, non-textual, non-deterministic, non-terminating, goto programs.

**Semantic issues:** Traditional program slicing makes a fundamental assumption that programs normally terminate: non-termination is considered aberrant, and non-terminating fragments not affecting the slicing criterion are fully removed. While this is rarely an issue for program slicing, it can not be directly translated easily to EFSMs which, because they are often used to abstract away details, are frequently non-terminating (as shown by Figure 1) and sometimes non-deterministic. We previously adapted a definition of control dependence for non-terminating programs [51] to EFSMs [4]. However, the only previously published EFSM slicing algorithm of Korel *et al.* [43] does not deal with non-terminating EFSMs.

### 1.3 Contributions

This paper makes two primary contributions:

1) We introduce a new 'core' slicing algorithm, the first that can slice non-deterministic and / or non-terminating EFSMs. The slicing algorithm is based in part on an adaption of Ilie and Yu's NFA minimisation algorithm [38] which uses a conservative approximation to equivalence to minimise NFAs by merging equivalent states. We use this to minimise the likelihood of slices being larger than their inputs. We prove that our adaption of this algorithm meets the requirements of slicing. We also introduce a new tool based on the algorithm.
   Though there has been work on non-amorphous slicing of EFSMs [21], [31], [43], [45], [46], [58], the only previously published amorphous slicing algorithm is that of Korel *et al.*. Their algorithm is limited to terminating, deterministic EFSMs with a single exit state and uses standard control and data dependence adapted from program slicing. Our algorithm is thus capable of slicing a much larger class of EFSMs.
2) We present the first empirical study of EFSM slicing, involving ten EFSMs, including widely studied benchmarks and real world models. Since our slicing algorithm can be instantiated for different control dependence formulations (leading, effectively, to a set of related slicing algorithms), we are able to test it against a variety of slicing algorithms; we also include the amorphous EFSM slicing approach of Korel *et al.* plus the additional two state merging heuristics they present, to ensure a proper comparison. As well as dealing with a much larger possible class of EFSMs than Korel *et al.*, our algorithm performs at least as well overall, and often better, in comparisons using various slicing metrics.

This paper is structured as follows. We first present a formal definition of EFSMs (Section 2) and an in-depth discussion of the problems facing EFSM slicing (Section 3). We follow this with a sketch giving the flavour of our approach together with an example to provide some intuition (Section 4). The section following this defines the dependence analyses used in the paper (Section 5) while Section 6 is the heart of the paper. It presents and explains the slicing algorithm, offers a running example of its important steps, and provides a case study, illustrating how the algorithm can be used. Section 7 discusses theoretical aspects of the algorithm. There follows a discussion of Korel's algorithm (Section 8) and a case study (Section 9). Section 10 presents our experimental results.

## 2 Extended Finite State Machines

In this section we present a formal definition of EFSMs, which we use throughout this paper.

**Definition 1.** *An Extended Finite State Machine (EFSM) $M$ is a tuple $(S, s_0, T, E, Var, v_0)$ where: $S$ is a set of states; $s_0 \in S$ is the initial state; $T$ is a set of transitions; $E$ is a set of events, where each event is an atomic action or signal, possibly parameterised; $Var$ is a store represented by a set of variables; and $v_0$ is a mapping from the variable names to the initial values of these variables. Transitions have a source state $source(t) \in S$, a target state $target(t) \in S$ and a label $label(t)$. Transition labels are of the form $e[g]/a$ where: $e \in E$; $g$ is a guard (we assume a standard condition language); and $a$ is a sequence of actions (we assume a standard expression language including assignments). All parts of a label are optional.*

States in $S$ are atomic (i.e. we do not consider hierarchical EFSMs). Events can be parameterised—part of their value is set by the environment. The environment, which we do not formally define, produces a stream of input events taken from $E$ and a source of values which can be bound to the parameters of parameterised input events. Actions can involve store updates. A *self-looping transition* is a transition $t$ where $source(t) = target(t)$. Transitions that share the same source state are said to be *siblings*. A transition $t'$ is said to be a *successor* of a transition $t$ if $source(t') = target(t)$. An *exit state* is a state that has no outgoing transitions. A *final transition* is one whose target is an exit state. An $\varepsilon$-transition is one with no trigger event, guard or action, i.e. one with no label. EFSMs may be non-deterministic, i.e. they can contain sibling transitions with the same trigger events and non-disjoint guards.

An EFSM is said to be *complete* or completely specified if, for every state $s$, event $e$ and valuation $v$ for the variables there is a transition with source $s$ and event $e$ whose guard is satisfied. To maintain consistency with Harel's Statecharts [27], we assume that an event $e$ leads to no change in state and no change in the value of the variables in *Var* if $e$ is received when the EFSM cannot process $e$ (i.e. there is no transition with event $e$, or the guard in a transition with that event is not satisfied). For example, if the EFSM of Figure 1 is

in the state $closing$ and receives the event $fullyOpened$, it can't respond to this event as there is no transition whose source state is $closing$ and has $fullyOpened$ as a trigger event. Although we don't present a formal semantics of EFSMs in this paper, this behaviour for an incomplete state machine would need to be made explicit at the semantic level. This is equivalent to the notion of *implicit transitions*; that is, if, in a state $s$ an event $e$ is received but does not enable any outgoing transition, an implicit transition from $s$ to $s$ consumes the event $e$.

In this paper, we make extensive reference to, and use of, Ilie and Yu's Non-deterministic Finite Automaton (NFA) minimisation algorithm. This requires viewing an EFSM as an NFA (or, more generally, Finite State Automaton, or FSA). However EFSMs differ from FSAs in important ways: EFSMs have stores and EFSM transition labels are more complex (they may contain a condition in addition to an event). FSAs are used to define languages and the concept of a final state plays an important role: a sequence (word) is in the language defined by an FSA if and only if it can take the FSA to a final state. If a sequence takes the FSA to a non-final state, but no further, then it is not in the language defined by the FSA. EFSMs have no equivalent notion: any sequence of inputs drawn from its input event alphabet is legal. To reflect this, we nominate every state of an EFSM as "final" in the sense in which the word is used in the FSA context. This is a distinct concept to that of an exit state[2] in an EFSM, i.e. a state at which the computation terminates.

## 3 Background

Program slicing is a well established area of research and practise and there are clearly many similarities between program slicing and EFSM slicing, particularly when viewed at a high-level. However, the low-level differences between the two introduce many challenges. In this section we look at these differences in more detail, providing motivation for considering EFSM slicing as a distinct discipline from program slicing, as well as looking at some of the challenges already identified for EFSM slicing, directly or indirectly, in the literature.

### 3.1 Syntactic issues

Program slicing traditionally relies on the notion of a lexical successor [1] (i.e. that removing one line means that control naturally flows from its predecessor to its successor), where EFSMs have no such notion. In some ways this is a similar, if more extreme, version of the problem that presents itself when slicing unstructured programs (i.e. programs with `gotos`) [1], [7], [29]. In structured programs (i.e. programs without `gotos`), the statements marked by dependence analysis form a *closure slice* [57] which is closely related to the resulting *executable slice* [34] that can be separately compiled and executed. The arbitrary control flow present in unstructured programs can break the relationship between the closure and executable slices. For example, it is impossible to produce an executable slice of an unstructured program that preserves terminating behaviour, introduces no new statements other than those in the closure slice, and no additional `goto` statement not in the original program [30].

Choi and Ferrante [18] introduced two algorithms to balance the compromise that these restrictions necessitate. In the first algorithm, the slice is potentially very large, because it includes as many statements from the original as are needed in order to preserve statement reachability in the slice; however this can lead to slices close in size to the original. To avoid this issue, the second algorithm adds statements to the program that are in neither the closure slice nor in the original program, 'rewiring' the program. Technically this results in an *amorphous slice* [28] i.e. a slice which is not a subset of the input program. Amorphous slices can introduce new, and merge existing, statements, whereas this algorithm introduces new, but does not merge existing, statements. We therefore refer to this as a 'slightly amorphous' slice.

Similar to program slicing, we use the notions of: *EFSM closure slice* (or simply 'closure slice' when the context is unambiguous) to refer to the original EFSM with some of its transitions marked to reflect the dependence analysis; and *EFSM executable slice* (or 'executable slice' when unambiguous) for the sub-EFSM extracted from the original using the slicing process. The unrestricted graph connectivity of EFSMs, where transitions can be thought of as akin to `gotos`, means that EFSMs are similar to unstructured programs: slicing an EFSM means removing states and reconnecting dangling, or creating new, transitions.

When slicing an EFSM we must first perform a dependence analysis that computes an EFSM closure slice. For some authors [45] this is the end point of the slicing process; for others [21], [31], [43], [46], [58] it is a step towards the construction of an EFSM executable slice. Several authors [21], [31] use a reachability approach to take this next step. Like Choi and Ferrante's first algorithm for slicing goto programs, this can yield a very large EFSM executable slice, because otherwise unwanted transitions have to be kept to ensure reachability of the desired transitions. Noting this problem, Korel *et al.* [43] introduced two heuristic rules for state merging that produce 'fully' amorphous slices of EFSMs. The use of these heuristics was illustrated by slicing an ATM EFSM, but there has been no other result in the literature on amorphous slicing of EFSMs. As we will later show, Korel *et al.*'s rules, while often effective, do not work as well as we may wish in some

2. Parts of the literature call these final states, though we will consistently refer to them as exit states.

situations.

The best measure of the effectiveness of slicing algorithms is less obvious than in program slicing, where 'length of text' is the undisputed metric. We may count the number of states, if only for compatibility with conventionally accepted notions of program slicing. However, since computation in EFSMs takes place on transitions, it makes equal sense to count the number of transitions. Furthermore, since an algorithm may duplicate transitions (i.e. changing their source and / or target, but maintaining the label) to 'rewire' the states of an EFSM, the number of unique transitions is also an interesting metric (since non-unique transitions require less effort to comprehend). In this paper we use each of these metrics.

### 3.2 Semantic issues

Dependence is the means by which a closure slice is computed and, by extension, upon which the executable slice also rests. Traditionally, two forms of dependence are considered in slicing: data and control dependence. EFSMs' use of a global store and variable reads and assignments in transitions means that data dependence is much the same in EFSM slicing as in program slicing [43].

Control dependence for EFSMS, on the other hand, is rather more complicated. For example, traditional program control dependence assumes that programs terminate; while this assumption is reasonable for programs, it is not for EFSMs, where non-termination is culturally common. Recently, Ranganath *et al.* [51] defined two notions of control dependence definitions for non-terminating programs: Non-Termination Sensitive Control Dependence (NTSCD) (i.e. slicing retains non-terminating subprograms) and Non-Termination Insensitive Control Dependence (NTICD) definitions (i.e. slicing removes non-terminating subprograms). We have previously adapted Ranganath *et al.*'s work to EFSMs as well as defining a third notion of control dependence for EFSMs called Unfair Non-Termination Insensitive Control Dependence (UNTICD) [4]. UNTICD overcomes NTICD's limitation that control dependence in control sinks is not identified. Section 5 defines all these terms in more detail. From the point of view of this paper, we consider all 3 forms of control dependence.

## 4 Approach overview

We introduce a novel slicing tool for EFSMs, the CREST EFSM slicing tool, that can slice EFSMs using a number of different algorithms: Korel *et al.*'s algorithm, and the set of slicing algorithms we define in Section 6.3. With an appropriate choice of algorithm, our tool can slice non-deterministic and / or non-terminating EFSMs—the first tool to do so. The common parts of the tool are as follows. It takes as input: an EFSM model $M$ (a text file that describes the states and transitions of the EFSM); a slicing criterion $C = (t, V)$, where $t \in T$ is a transition and $V \subseteq Var$ is a subset of the variables; and one of three control dependence definitions (NTSCD, NTICD, UNTICD). It outputs a slice $M'$ of the EFSM $M$. It has two core phases: dependence analysis and slicing.

During dependence analysis, a *dependence graph* is computed, using the given definition of control dependence and data dependence. See Figure 14 for an example of a dependence graph. Nodes in a dependence graph represent EFSM transitions and edges represent control and data dependence, and all transitions backwardly reachable from the slicing criterion $C$ are 'marked'. These marked transitions correspond to the transitive closure of data dependence and control dependence with respect to $C$.

After dependence analysis, the tool then amorphously slices the EFSM, using the dependency graph as a guide. As an example of how this works, our set of slicing algorithms (i.e. not including Korel *et al.*'s algorithm) aim to remove all unmarked transitions and reconnect the EFSM as needed. To do this, unmarked transitions are *anonymised* by replacing their label with $\varepsilon$. We then apply our adaption of Ilie and Yu's NFA minimisation algorithm to remove the $\varepsilon-$transitions and re-wire the graph appropriately.

For example, consider Figure 1 and slicing it using NTICD control dependence with a slicing criterion of $(T11, \{timer\})$. The first step in slicing is the marking of the state machine and the removal of the labels on unmarked transitions (Figure 3). To do this we construct a model dependence graph (MDG) and we mark the transition in the slicing criterion and any other transitions on which that transition depends according to the MDG. Unmarked transitions are actually marked with $\varepsilon$, i.e. the "empty" marking. $\varepsilon$-transitions are then removed, as per Ilie and Yu's algorithm (Figure 8). Unmarked transitions are then deleted (Figure 9) and garbage collection performed (Figure 10). Not all the steps in the algorithm may be used, depending on the EFSM to be sliced. Illustration of the possible minimisation steps can be found in Figures 4 and 5 and in Ilie and Yu's paper [38].

## 5 Dependence analysis

In this section, we present the definitions of NTSCD, NTICD, and UNTICD which differ in the type of paths used: maximal paths, sink-bounded paths, or unfair sink-bounded paths. A path is defined as a sequence of successive transitions. A transition is in the path if it is in the sequence, while a node is in the path if it is either the source or target of one of these transitions. Every path has an initial node but not necessarily a final node. Three types of paths can be used to define different kinds of control dependence: maximal paths, sink-bounded paths that are given in terms of control sinks, and unfair sink-bounded paths.

**Definition 2.** *(Maximal Path) A maximal path is any path that terminates in a final transition, or is infinite.*

**Definition 3.** *(Control Sink) A control sink in an EFSM is a set of transitions $\mathcal{K}$ that forms a strongly connected component (SCC) such that, for each transition $t$ in $\mathcal{K}$ each successor of $t$ is also in $\mathcal{K}$.*

**Definition 4.** *(Sink-bounded Path) A path $\pi$ is a* sink-bounded path *if either $\pi$ contains a final transition or there exists a control sink $\mathcal{K}$ such that $\pi$ contains **every transition** from $\mathcal{K}$ infinitely often.*

The second clause of Definition 4 defines a form of fairness, which prevents control dependence from being calculated in control sinks for non-terminating EFSMs (see e.g. [4], [51]). The following definition removes that clause, allowing such dependence to be calculated. We use the word "unfair" to denote the relaxation of fairness, i.e. not necessarily fair [50].

**Definition 5.** *(Unfair Sink-bounded Path [4]) A path $\pi$ is an* unfair sink-bounded path *if either $\pi$ contains a final transition or there exists a control sink $\mathcal{K}$ such that $\pi$ contains **transitions** from $\mathcal{K}$ infinitely often.*

We define control dependence using the abstract function *PATH* which maps a node to the set of paths (of a given type) that have that node as the source of the first transition on each path. The type of control dependence is parameterised by providing an instantiation of *PATH*: maximal paths define *NTSCD*; sink-bounded paths define *NTICD*; unfair sink-bounded paths define *UNTICD*.

**Definition 6.** *(Control Dependence (CD)) $T_i \xrightarrow{CD} T_j$ means that a transition $T_j$ is control dependent on a transition $T_i$ iff :*

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

We adopt the data dependence definition of [43] for EFSMs: dependence occurs between a variable's definition and use, providing there are no further definitions of that variable in the intervening path.

**Definition 7.** *(Definition/Use) A variable $v$ is defined at transition $T_i$ if:*

1) *$v$ occurs on the the LHS of assignments in the actions of $T_i$, or*
2) *$v$ is a parameter of the event of $T_i$ and $v$ is not redefined in any action of $T_i$.*

*A variable $v$ is used at transition $T_i$ if:*

1) *$v$ occurs in the guard of $T_i$, or*
2) *$v$ occurs in the RHS of assignments in the actions of $T_i$.*

We assume that there are two ways in which a variable can be defined, i.e. given a fresh value: the variable occurs as the parameter in a parameterised input event as in transition T1 of Figure 12, or the variable's value is updated as the result of an action. We assume that input event, condition evaluation and action occur in that order, hence occurring as a parameter of an input event does not define a variable if it is immediately followed by an action which updates it.

| | | |
|---|---|---|
| NTSCD | $T3 \rightarrow T4, T5, T6$ | $T5 \rightarrow T9, T10$ |
| | $T6 \rightarrow T7, T8$ | $T8 \rightarrow T9, T10$ |
| | $T10 \rightarrow T11, T12$ | $T12 \rightarrow T4, T5, T6$ |
| NTICD | No dependences | |
| UNTICD | $T6 \rightarrow T7, T8$ | $T5 \rightarrow T9, T10$ |
| | $T10 \rightarrow T11, T12$ | $T8 \rightarrow T9, T10$ |
| | | $T12 \rightarrow T4, T5, T6$ |
| DD | $T1 \rightarrow T2, T3$ | $T2 \rightarrow T2, T3$ |
| | $T5 \rightarrow T11, T12$ | $T8 \rightarrow T11, T12$ |
| | $T11 \rightarrow T11, T12$ | |

TABLE 1
Dependence for Figure 1.

**Definition 8.** *(Data Dependence (DD)) $T_i \xrightarrow{DD}_v T_k$ means that transition $T_k$ is data dependent on transition $T_i$ with respect to variable $v$ if:*

1) *$v \in D(T_i)$, where $D(T_i)$ is the set of variables defined by transition $T_i$;*
2) *$v \in U(T_k)$, where $U(T_k)$ is the set of variables used by transition $T_k$;*
3) *there exists a path in the EFSM from $target(T_i)$ to $source(T_k)$ on which $v$ is not defined.*

To help make our control and data dependency definitions concrete, dependencies for the door controller EFSM in Figure 1 are given in Table 1. The door controller EFSM has a non-terminating control sink consisting of the transitions T4, T5, T6, T7, T8, T9, T10, T11, and T12. NTSCD has termination sensitive dependences outside of the control sink, namely $T3 \rightarrow T4, T5, T6$, and more within the control sink. NTICD has no dependences, since there are no non-termination insensitive dependences outside the control sink. UNTICD has dependences only within the control sink (which are the same as NTSCD).

## 6 AMORPHOUS SLICING

Graph-based slicing that eliminates unnecessary transitions and nodes produces an output that will not be a sub graph of the original. Although we term this *amorphous slicing* it is amorphous in a lighter sense than amorphous program slicing. This form of slicing arises naturally when slices for an EFSM are constructed based on dependence analysis with respect to a slicing criterion.

**Definition 9.** *(Slicing Criterion) A slicing criterion for an EFSM is a pair $(t, V)$ where transition $t \in T$ and variable*

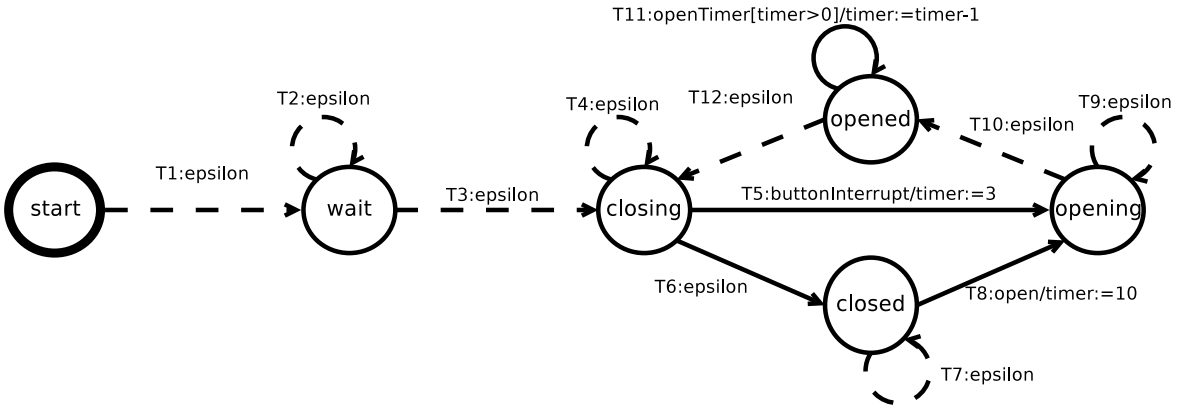


Fig. 3. Figure 1 marked using NTICD and data dependence with the slicing criterion (T11, {timer}).

*set $V \subseteq Var$. It refers to the store value immediately after the execution of the action contained in transition $t$.*

EFSM slicing is comparable to slicing interactive programs. For any possible input event sequence we cannot guarantee that the original and the sliced EFSM model behave similarly with respect to the values of the variables of interest. The following definition of a slice is loosely based on Korel *et al.* and uses non-stuttering event sequences. We view the triggering of an implicit transition as a 'stutter' because nothing happens as a result. No action is taken and the machine remains in the same atomic state. So, by a non-stuttering event sequence we mean an event sequence which never triggers an implicit transition.

**Definition 10.** *(Slice) An EFSM slice $M'$ is a reduced state machine of the original EFSM $M$, if whenever the environment produces a non-stuttering input event sequences $e1$ for $M$, there exists a non-stuttering input event sequence $e2$ for $M'$ such that:*

1) *$e2$ is a sub-sequence of $e1$. A sub-sequence $x$ of a sequence $y$ is a sequence that can be produced from $y$ by removing some elements without changing the order of the remaining elements.*
2) *When executing $e1$ on $M$ the values of the variables $V$ at $t$ are equal to the values of $V$ at $t$ during the execution of $e2$ on $M'$.*

Given an EFSM $M$ and a slicing criterion (t,V), the basic idea of the slicing algorithm, after computing dependence and marking transitions, is to anonymise unmarked transitions and then apply an adaption of Ilie and Yu's NFA reduction algorithm. In the following subsections we expand upon the Ilie and Yu algorithm and discuss how we use it.

### 6.1 $\varepsilon$-elimination

Ilie and Yu [37], [38] describe an algorithm for constructing $\varepsilon$NFAs (an $\varepsilon$NFA is an NFA with silent or $\varepsilon$-transitions) from regular expressions. The algorithm is iterative, and at each step applies three rules to improve the $\varepsilon$NFA further. The $\varepsilon$NFA that is produced is called a *follow* NFA. It then applies the $\varepsilon$-elimination algorithm to the follow NFA and obtains an NFA.

We construct EFSMs with $\varepsilon$-transitions after dependence analysis, by anonymising unmarked transitions, as the example shown in Figure 3. Before we apply the $\varepsilon$-elimination algorithm, we apply the rules, given by Ilie and Yu (part of Algorithm 7 in [38]), for merging states and/or removing $\varepsilon$-transitions. This helps to reduce the risk of increasing the number of transitions of slice, though it cannot eliminate this risk completely. We apply these rules iteratively, until a fixed point is reached. The rules are as follows:

1) for all states $p$ and $q$, if there is a single $\varepsilon$-transition between $p$ and $q$ and no other transition from $p$, then $p$ and $q$ can be merged.
2) any cycle consisting only of $\varepsilon$-transitions between states can be collapsed, by merging states and deleting $\varepsilon$-transitions.
3) copies of transitions with the same source, target and label (and name) are deleted.

Then we apply the $\varepsilon$-elimination algorithm given in [37] (see Algorithm 20), in order to remove $\varepsilon$-transitions. The definition is as follows:

**Definition 11.** *($\varepsilon$-elimination) For every path containing only $\varepsilon$-transitions between two states $p$ and $q$ in an EFSM $M$ and any transition with label $a$ from $q$ to $r$, add a transition with label $a$ from $p$ to $r$ if no such transition already exists. Furthermore, if $q$ is a final state, then $p$ should become a final state. (Note: in EFSMs all states are final states and therefore this step is not required.) Then remove all $\varepsilon$-transitions and unreachable states in $M$.*

For removing unreachable states from the EFSM we use a standard garbage collection process using the start state(s) as the root nodes of the garbage collection (as described in e.g. [40]).

The process of eliminating $\varepsilon$-transitions can make an NFA larger, which is also true of our EFSM slice construction adaption of it. For example, Figure 4

illustrates an EFSM with a single $\varepsilon$-transition and a total of five transitions and four states. After applying the $\varepsilon$-elimination algorithm, the number of states remains the same, while the number of transitions increases by one, as shown in Figure 5. It is known that for an NFA with $n$ states and alphabet size $p$, the process of eliminating $\varepsilon$-transitions can lead to an NFA with $O(n^2p)$ transitions [35]. It is also known that this process cannot do much better: there is a class of regular languages $L_n$ that requires almost quadratically more transitions to describe if $\varepsilon$-transitions are not allowed [35]. There may thus be merit in retaining some of the $\varepsilon$-transitions. One approach aims to minimise the number of non-$\varepsilon$-transitions of an NFA and is guaranteed to be optimal if the NFA is unambiguous [39]: the NFA has at most one accepting computation for each sequence. Unfortunately, in our case all states are final states and so our automata are unambiguous if and only if they are deterministic. As a result, the property of being unambiguous is of no interest. In Section 10 we empirically investigate how often this problem happens in practice.

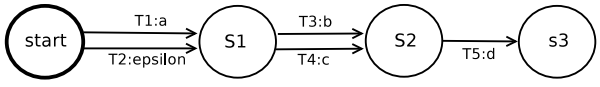


Fig. 4. Example of an EFSM with an $\varepsilon$-transition.

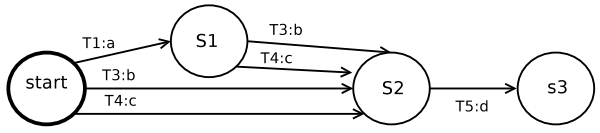


Fig. 5. After applying $\varepsilon$-elimination to Figure 4.

### 6.2 Minimisation

The steps defined previously produce an EFSM $M$ that has no $\varepsilon$-transitions or unreachable states. However, there may be smaller EFSMs that are equivalent to $M$ and so the next step is to try to produce such a smaller EFSM. Unfortunately, while minimisation of DFA can be achieved in low-order polynomial time [32], minimisation of NFAs is known to be PSPACE-complete[3] [23], [33]. In fact, even the problem of deciding whether two states of an NFA are equivalent is PSPACE-complete [23], [33]. Thus, we cannot reasonably expect to minimise $M$.

Since the problem of minimising an NFA is computationally hard, there has been interest in reducing an NFA to a smaller but not necessarily minimal NFA and here we describe such an approach [38]. The essential idea is to merge states that are known to be equivalent but, since deciding equivalence is PSPACE-complete, we use a conservative approximation to

3. All PSPACE-complete problems are NP-hard [23].

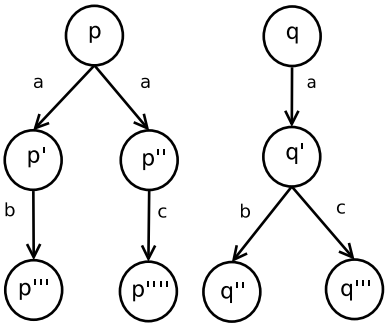


Fig. 6. Finite automata where $p$ and $q$ are not right invariant equivalent.

equivalence. This notion of equivalence is the largest right invariant equivalence:

**Definition 12.** *An equivalence relation $\equiv$ over the states of NFA $M$ is* right invariant *with respect to $M$ if it satisfies the following properties:*

1) *No final state of $M$ is equivalent to a non-final state of $M$ under $\equiv$; and*
2) *For all pairs of states $p, q$ and label $a$, if $p \equiv q$ and there is a transition from $p$ to $p'$ with $a$ then there is a transition from $q$ to some $q'$ with $a$ such that $p' \equiv q'$.*

As we consider EFSMs in this paper, where all states are final, the first property immediately holds. Clearly, if $\equiv$ is a right invariant equivalence with respect to $M$ and $p \equiv q$ then $p$ and $q$ are equivalent[4]. However, the converse is not the case. To see this, consider states $p$ and $q$ defined in Figure 6 in which all states are final states.

It is straightforward to see that $p$ and $q$ are equivalent. However, if $\equiv$ is a right invariant equivalence with respect to $M$ then for us to have that $p \equiv q$ we would require that there is a transition from $p$ with label $a$ to a state $p_0$ such that $p_0 \equiv q'$ but this cannot be the case: if we choose $p_0 = p'$ then we do not have $p_0 \equiv q'$ since there is no transition from $p_0$ with label $c$ and if we choose $p_0 = p''$ then we do not have $p_0 \equiv q'$ since there is no transition from $p_0$ with label $b$.

There can be many alternative right invariant equivalences with respect to $M$ and some may be better than others when considering NFA reduction. A right invariant equivalence is said to be a largest right invariant equivalence if it defines the most pairs of equivalent states. It transpires that the largest right invariant equivalence $\equiv_R$ with respect to $M$ can be found in polynomial time [38]. In addition, if we merge states that are found to be equivalent, $\equiv_R$ leads to a smallest NFA that can be produced from $M$ by merging states that are equivalent under a right invariant equivalence with respect to $M$[5]. Thus, $\equiv_R$ is the best right invariant equivalence to use.

4. This is Lemma 12 in [38].
5. Theorem 17 and Corollary 18 of [38]

It is sometimes possible to further reduce an NFA by merging pairs of states $p, q$ that are *reached* by the same sets of sequences in $M$. It has been observed that we can define the notion of a left-equivalence [38] and that the results and algorithms for right-equivalence can be reused by simply reversing the transitions of $M$: there is a transition from $p$ to $q$ with label $a$ in the reverse $M'$ of $M$ if and only if there is a transition from $q$ to $p$ with label $a$ in $M$. Thus, we can compute the largest left-equivalence $\equiv_L$ with respect to $M$, which is also the largest right-equivalence with respect to $M'$, as well as the largest right-equivalence $\equiv_R$ with respect to $M$. We merge two states $p$ and $q$ if either $p \equiv_L q$ or $p \equiv_R q$.

### 6.3 Formalisation of the slicing algorithm

High-level pseudo code describing our slicing algorithm is given in Figure 7. Lines 2–3 are concerned with dependence analysis. Lines 4–12 are concerned with slicing and adapt the Ilie and Yu algorithm [36].

**Input:** EFSM $M_{pre}$ to be sliced.
**Input:** Slicing criterion $t_{sc}$ and its variables $V_{sc}$.
**Input:** Control Dependence definition $\text{CD}_{def}$.
**Output:** EFSM slice $M_{post}$.
1. $M_{post} \leftarrow M_{pre}$
2. DG $\leftarrow$ compute_dependence_graph($\text{CD}_{def}$,$M_{pre}$)
3. $M_{post} \leftarrow$ traverse_backwards_marking($t_{sc}$,$V_{sc}$,$DG$)
4. $M_{post} \leftarrow$ anonymise_unmarked_transitions($M_{post}$)
5. **while** apply_rule1($M_{post}$) **or** apply_rule2($M_{post}$) **or** apply_rule3($M_{post}$) **do**
6. **end while**
7. apply_epsilon_elimination($M_{post}$)
8. garbage_collect($M_{post}$)
9. $S_{eq} \leftarrow$ right_invariant_equivalence($M_{post}$)
10. merge_states($S_{eq}$,$M_{post}$)
11. $M_{post} \leftarrow$ left_invariant_equivalence($M_{post}$)
12. **return** $M_{post}$

Fig. 7. High-level slicing algorithm.

We now define the functions used in the algorithm of Figure 7.

*compute_dependence_graph($\text{CD}_{def}$,$M_{pre}$)* computes the dependence graph for the EFSM $M_{pre}$ by using the control dependence definition $\text{CD}_{def}$, and data dependence. For data dependence, the algorithm computes, as for program dependence analysis, definition-clear definition-use paths for all variables in each transition and returns all transition pairs in which there exists a definition-clear definition-use path. For control dependence, the algorithm computes control successors (i.e. the transitions that are control dependent on the transition) for each transition $m \in T_{pre}$ using $\text{CD}_{def}$.

*traverse_backwards_marking($t_{sc}$,$V_{sc}$,$DG$)* traverses the dependence graph backwards from the transition of interest $t_{sc}$ and marks all visited nodes. It is as defined in [43].

*anonymise_unmarked_transitions($M_{post}$)* replaces the label of unmarked transitions with an $\varepsilon$.

*apply_rule1($M_{post}$)*, *apply_rule2($M_{post}$)* and *apply_rule3($M_{post}$)* apply the rules as in Algorithm 4 [37] (page 144). Section 6.1 describes these rules for merging states and deleting $\varepsilon$-transitions. The functions return *True* if the rule has been applied, otherwise *False*. Lines 5–6 show that these rules are iteratively applied until they have no effect.

*apply_epsilon_elimination($M_{post}$)* is as defined in Algorithm 20 in [37] (page 153).

*garbage_collect($M_{post}$)* performs a standard mark and sweep garbage collection [40]. Any nodes and edges not reachable from $M_{post}$ are thus deleted.

*right_invariant_equivalence($M_{post}$)* is as defined in Algorithm 14 [36] (page 383). Since all states are final states in EFSMs, we can simplify the algorithm removing the code that checks that final and non-final state must not be equivalent (line 7-8 in Algorithm 14 in [36]). It returns the set of equivalent states.

*merge_states($S_{eq}$,$M_{post}$)* is a function that takes a set of right invariant equivalent states and merges them in $M_{post}$.

*left_invariant_equivalence($M_{post}$)* reverses the direction of transitions of the current EFSM $M_{post}$, resulting in the EFSM $M_{rev}$. Then it applies *right_invariant_equivalence($M_{rev}$)* . If there are any right-equivalent states, these are merged using the function *merge_states* with $M_{rev}$. Then it reverses the direction of the transitions of the EFSM $M_{rev}$ and returns the result as $M_{post}$.

### 6.4 Slicing the door controller EFSM

Suppose that the slicing criterion for the door controller EFSM in Figure 1 is (T11, {timer}). If the NTSCD definition is chosen, then the marked transitions are {T1, T2, T3, T5, T6, T8, T10, T11, T12}. The slice that is generated contains all transitions in Figure 1 except for the self-looping transitions T4, T7 and T9.

If the NTICD definition is chosen, then the marked transitions are {T5, T8, T11}, which are all data dependences owing to the structure of the graph, i.e. there are no NTICD dependences within control sinks and, in this case, there are no control dependences outside of the control sink.

Figure 3 illustrates Figure 1 after marking. Unmarked transitions are indicated by dashed lines and have had their labels replaced with $\varepsilon$, the empty label. Figure 8 shows the resulting state machine after the rules in steps 5–6 have been applied. These remove some $\varepsilon$ transitions and merge some states. In particular, rule 2 is applied that removes the self-transitions and rule 1 merges the states *start*, *wait* and *closing* as well as *opened* and *opening*. Figure 9 shows

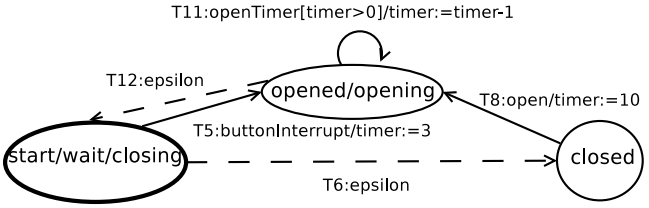


Fig. 8. Figure 3 after applying rules 5-6 of the algorithm.

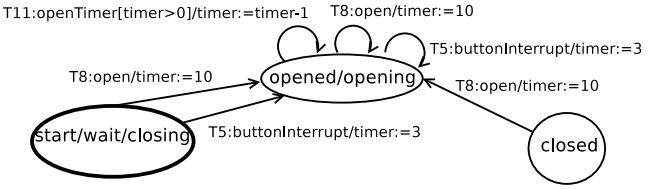


Fig. 9. Figure 8 after epsilon-elimination.

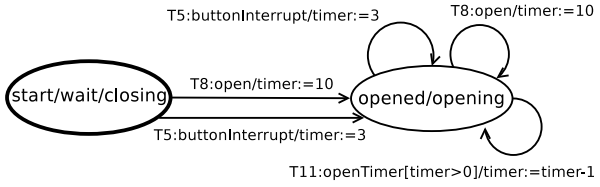


Fig. 10. The slice on (T11,$\{timer\}$) for Figure 1 using NTICD.

the state machine after $\varepsilon$-elimination has been applied and before garbage collection. Finally, Figure 10 illustrates the state machine after garbage collection, i.e. illustrates the slice generated after applying the slicing algorithm. The state *closed* and its transitions are deleted during garbage collection. This slice is no longer a sub-model of the original and thus is not syntax preserving. It is, however, straightforward to demonstrate that it is semantics preserving with respect to the slicing criterion.

If the UNTICD definition is chosen for control dependence, then the marked transitions are {T5, T6, T8, T10, T11, T12}. Figure 2 illustrates the slice generated after applying the slicing algorithm.

## 7 Theoretical Aspects

In this section we discuss some basic properties of the algorithm, choosing between control dependencies, intuitions with regard to correctness issues and present a proof that its complexity is cubic.

### 7.1 Properties of the slicing algorithm

We have identified two properties of the slicing algorithm: first, that the number of states in the slice is never more than that in the original EFSM; second, that the number of unique transitions (transitions with unique labels i.e. that have syntactically different labels) in the slice is never more than the number of unique transitions in the original EFSM. The latter property is chiefly used as a sanity check.

**Proposition 7.1.** *For an EFSM $M$ and its slice $M'$, where $S_M$ is the set of all states in $M$ and $S_{M'}$ is the set of all states in $M'$, then $|S_{M'}| \leq |S_M|$.*

*Proof:* The algorithm can change the number of states of an EFSM in two places: when applying the rules before $\varepsilon$-elimination (lines 5-6), and during the minimisation phase (lines 7-12).

In the first case, there are three rules that are applied. The first rule, if true for two states will merge them, thereby reducing the number of states. The second rule, also merges states where cycles of $\varepsilon$-transitions exist, thus reducing the number of states. The third rule deletes copies of transitions, which does not affect the number of states. If these rules cannot be applied, then the number of states remains the same. The $\varepsilon$-elimination phase deals exclusively with transitions and thus has no effect on the number of states.

In the second case, minimisation merges equivalent states, which reduces the number of states. If there are no equivalent states, then no merging occurs, and the number of states remains the same. Thus, the algorithm cannot increase the number of states. □

**Proposition 7.2.** *For an EFSM $M$ and its slice $M'$, where $U_M$ is the set of all unique transitions in $M$ and $U_{M'}$ is the set of all unique transitions in $M'$, then $|U_{M'}| \leq |U_M|$.*

*Proof:* The $\varepsilon$-elimination algorithm preceded by the rules defined by Ilie and Yu are applied to remove $\varepsilon$-transitions and reconnect the EFSM. The rules merge states and remove $\varepsilon$-transitions. The $\varepsilon$-elimination algorithm copies marked transitions to reconnect the graph, and deletes $\varepsilon$-transitions and any unreachable states and transitions. Therefore, no new unique transitions are introduced, i.e. transitions not in the set of marked transitions. Since, the total number of unique marked transitions is never more than the total number of unique transitions in $M$, $|U_{M'}| \leq |U_M|$. □

### 7.2 Control dependence definitions and the correctness of the algorithm

The effects of using each of the three control dependence definitions in the slicing algorithm are illustrated in section 6.4 and by examining figures 1, 2 and 10. The algorithm performs static, backward slicing so the applications are confined to debugging, code maintenance, comprehension and other static backward slicing applications. The most suitable of the three control algorithms for these slicing applications is NTICD. However, NTICD has a drawback in that it does not find any control dependence information inside control sinks. Since many EFSMs are descriptions of reactive systems and their specification is

either wholly or largely a single control sink, and use of this definition in the slicing algorithm tends to introduce non-determinism in the parts of the SCCs retained in the resulting slice, it is desirable to have a control dependence definition that reduces the non-determinism.

As a result, some research into state machine slicing adopts a definition based on NTSCD as the notion of control dependence [45]. NTSCD is termination sensitive so it includes transitions which make choices between termination and non-termination. This form of control dependence can then distinguish between different possible infinite paths in a control sink, providing an indication as to the most direct path to a slicing criterion but generally producing larger slices. Its most apposite application is as part of an information flow analysis via a dependence graph in order to establish security properties such as non-interference.

UNTICD was created as a halfway house. It behaves like NTICD outside of control sinks and like NTSCD inside control sinks. In this way, like NTICD, it is analogous to the traditional control dependence (which ignores non-termination possibilities) in program slicing, but it has the benefit of reducing non-determinism within control sinks. For detail see Androutsopoulos *et al.* and Ranganath *et al.* where relationships between the three are formally discussed [4], [51]. One result from those papers is that the transitive closure of the dependencies for NTICD is contained within that of UNTICD which is in turn contained within that for NTSCD.

A consequence of the latter is that if a slicing algorithm using NTICD is correct, so are those which replace it with the other two. Note that this replacement assumes that the definition of correctness remains the same, i.e. that the semantics is invariant in some sense with respect to the slicing criterion. NTSCD was conceived as a program language control dependence useful for security analyses in which non-termination is a leakage channel. However it is also commonly used in state based model slicing purely for software engineering applications with a correctness criterion of the type above [3].

Formal consideration of correctness is outside the scope of this paper but we comment that, as indicated in section 6, the interactive nature of EFSMs means that slicing cannot ignore the environment which is interacting with them. In particular, once some transitions have been removed from the slice, the slice is only complete via implicit transitions and may react quite differently from the original machine to some sequences of input events and it may be possible that the slice will not even take a path through the slicing criterion when the original does. This effect can be replicated in interactive programs as well [52]. As a result, the definition of a slice must be weakened so that it applies only to non-stuttering input sequences for the original and projections of these which are non-stuttering in relation to the slice. The definition is further weakened to an existential condition to take account of possible non-determinism. This is the key intuition as to why an algorithm using NTICD, which finds no dependency information inside control sinks, is correct with respect to the definition for EFSMs.

### 7.3 The complexity of the algorithm

This section shows that Algorithm in Figure 7 has polynomial worst case time complexity. Throughout this section we assume that the EFSM $M$ being considered is represented by directed graph $G$ in which state $s_i$ is represented by vertex $n_i$ and there is an edge from vertex $n_i$ to vertex $n_j$ if and only if $M$ has a transition from $s_i$ to $s_j$. We use $n$, $m$ and $k$ to denote the numbers of states, transitions and variables of $M$ respectively and assume that every state is either a source or target of at least one transition[6]. We consider the case where NTSCD is used; the proofs for NTICD and UNTICD are almost identical.

**Lemma 1.** *Data dependence can be computed in $O(m^3k)$ time.*

*Proof:* First consider the problem of determining whether transition $T_j$ is data dependent on transition $T_i$ with respect to variable $v$. In order to decide this we can form a directed graph $G_v$ from $G$ by removing all edges corresponding to transitions in which $v$ is defined and this can be produced in $O(m)$ time. Then $T_j$ is data dependent on $T_i$ with respect to $v$ if and only if $T_i$ defines $v$, $T_j$ uses $v$ and the source state of $T_j$ can be reached from the target state of $T_i$ in $G_v$. Reachability can be decided using a depth-first search in $O(m)$ time [55] and so the result follows from there being $O(m^2)$ pairs of transitions and $k$ variables. □

We now consider control dependence.

**Lemma 2.** *NTSCD control dependence can be computed in $O(m^3)$ time.*

*Proof:* Consider the problem of deciding whether transition $T_j$ is NTSCD control dependent on transition $T_i$. Let us suppose that $T_i$ has source state $s_{i_1}$ and target state $s_{i_2}$ and $T_j$ has source state $s_{j_1}$. We need to determine whether: all maximal paths from $s_{i_2}$ include $s_{j_1}$; and there exists a maximal path from $s_{i_1}$ that does not include $s_{j_1}$.

Let $G_1$ denote the directed graph produced from $G$ by removing vertex $n_{j_1}$ and assume that we know the vertices that are in cycles in $G_1$; these vertices can be found in $O(m)$ time [55]. Then all maximal paths from $s_{i_2}$ include $s_{j_1}$ if and only if either $s_{i_2} = s_{j_1}$ or, in $G_1$, from $n_{i_2}$ it is not possible to reach a cycle or a vertex that corresponds to an exit state. The first part can be decided in constant time and the second can be decided in $O(m)$ time using a depth-first search

6. This allows us to know that $n$ is of $O(m)$.

starting at $n_{i_2}$. Thus, the first condition for $T_j$ being NTSCD control dependent on $T_i$ can be decided in $O(m)$ time. Similarly, there exists a maximal path from $s_{i_1}$ that does not include $s_{j_1}$ if and only if $s_{i_1} \neq s_{j_1}$ and, in $G_1$, from $n_{i_1}$ it is possible to reach a cycle or a vertex that corresponds to an exit state. Again, this can be decided in $O(m)$ time. To summarise, it is possible to decide in $O(m)$ time whether $T_j$ is NTSCD control dependent on $T_i$ and so the result follows from there being $O(m^2)$ pairs of transitions. □

We now consider step 5 of the algorithm, which applies rules 1-3.

**Lemma 3.** *Step 5 of the Algorithm in Figure 7 takes $O(m^2 \log m)$ time.*

*Proof:* We will consider the separate rules, assuming that a list of the transitions is initially sorted on source state, then target state and then label (in $O(m \log m)$ time).

Rule 1 merges states $s$ and $s'$ where there is a single $\epsilon$-transition from $s$ to $s'$ and no other transition from $s$. Since the list of transitions has been sorted, it takes $O(m)$ time to determine whether this rule can be applied and $O(m)$ time to apply the transformation (we replace $s$ by $s'$ in the target states of transitions).

In rule 2, a cycle consisting only of $\epsilon$-transitions is collapsed by merging states and deleting $\epsilon$-transitions. We can determine whether there is such a cycle by considering the directed graph $G_\epsilon$ produced from $G$ by removing all transitions that are not $\epsilon$-transitions. We can produce $G_\epsilon$ in $O(m)$ time and determine whether it has any cycles in $O(m)$ time through applying Tarjan's algorithm, for finding the components of a directed graph [55]. If there is such a cycle then an $O(m)$ depth-first search can be used to find one such cycle. The transformation takes $O(m)$ time (we add a new state $s$ and in the transitions we replace each state of the cycle by $s$). A final $O(m \log m)$ step sorts the list of transitions.

Rule 3 deletes copies of transitions with the same source, target and label (and name). Since the list of transitions has been sorted, it takes $O(m)$ time to determine whether this rule can be applied and constant time to apply the transformation.

The result now follows from the fact that each application of a rule reduces the number of transitions. □

The other steps of the Algorithm are performed using standard algorithms. Garbage collection is simply a depth-first search and so can be computed in $O(m)$ time. The transitive closure of the union of control and data dependence can be computed in $O(n^3)$. For example, one could apply a depth-first search from each vertex in the directed graph $G_D$ in which the vertices represent states of the EFSM and the edges represent control and data dependence: $G_D$ has $O(n^2)$ edges and so one such search can be completed in $O(n^2)$ time and there are at most $n$ such searches. The equivalences used by Ilie and Yu [36] can be computed in $O(nm)$ time [2]. Since Ilie and Yu's $\epsilon$ elimination algorithm takes $O(m^2)$ time [37] we obtain the following result.

**Theorem 7.3.** *Algorithm in Figure 7 has worst case time complexity of $O(m^3 k)$.*

## 8 KOREL *et al.*'S SLICING ALGORITHM

Korel *et al.* [43] present a slicing algorithm, based on dependence analysis, for slicing EFSMs. They assume that the EFSMs are deterministic and executable, i.e. that they contain enough information in order to be executed. Their slicing algorithm is syntax preserving, i.e. the slices are executable sub-models of the original EFSMs (referred to in [43] as the executability property). Slices are computed with respect to the slicing criterion, a transition $t$ and its variables $V$, by using a dependence graph. The dependence graph represents the control and data dependences in the EFSM. Data dependence is defined as in Definition 8. The control dependence definition is given in terms of post dominance that requires execution paths to lead to an exit state. The algorithm deletes all paths that do not reach $t$ and marks all backwardly reachable transitions from $t$ in the dependence graph. Then, in order to preserve the graph structure, it marks any unmarked and non-self-looping transitions. Any unmarked self-looping transitions that are not data dependent on any marked transition are deleted. This is safe because self looping transitions cannot control the taking of subsequent transitions. For more details please see the discussion in an earlier paper [5]. Note that unmarked transitions are not anonymised. Unfortunately, because it is conservative in nature, the algorithm is generally not able to reduce EFSMs in size a great deal. For example, the slice generated for the ATM model (which is used as a running example in [43]) reduces the original (23 transitions and 9 states) only by 3 transitions.

In order to reduce the size of the slices further, Korel *et al.* describe two reduction rules for merging states which are applied after marking transitions. By merging states, slices may become non-deterministic, i.e. there may be numerous possible executions with the same event sequence. The slices produced after applying these rules are no longer syntax preserving, and so are amorphous slices [28]. Moreover, they relax the executability property by requiring that at least one of the executions of the non-deterministic slice, rather than all of the executions, preserves the value of the variables of $t$ for a given input.

We have chosen to compare our slicing algorithm to Korel *et al.*'s slicing algorithm and to an adaptation of Korel *et al.*'s reduction rules with our algorithms because they are the only algorithms in the literature to produce slices that are not sub-models of the original. Thus, similarly to our algorithm, they suffer from the

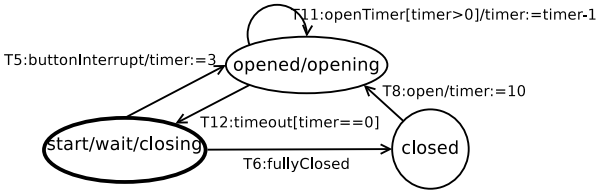


Fig. 11. The slice of the door controller EFSM on (T11,$\{timer\}$), using NTICD and Korel *et al.*'s reduction rules.

problem of re-connecting the graph. However, we take a more general approach to addressing this problem by adapting the Ilie and Yu algorithm [38].

We have implemented Korel *et al.*'s slicing algorithm as described in [43], using their definitions of control dependence and data dependence. Furthermore, we have implemented Korel *et al.*'s reduction rules, which are applied after dependence analysis where transitions are marked. Since Korel *et al.*'s control dependence definition only applies to models with a unique exit state, we also use NTSCD, NITCD and UNTICD during dependence analysis as they apply to a larger set of models, i.e. non-deterministic, non-terminating EFSMs. Then we directly compare the size of the slices. For example, the slice generated for the door controller EFSM using NTICD and Korel *et al.*'s reduction rules is shown in Figure 11. It contains transitions that are not in the set of marked transitions identified during dependence, i.e. T6 and T12. This is because the source state of T6 and T12 have outgoing transitions and thus their reduction rules for state merging cannot be applied. Unlike our slicing algorithm, the set of uniquely labelled marked transitions in the original EFSM $M$ is not equal to uniquely labelled transitions in the slice $M'$.

## 9 Case Study

In this section, a case study is presented to show how slicing can help to debug in EFSMs.

An EFSM model of an ATM system presented by Korel [43] is shown in Figure 12. Two types of accounts (checking account and savings account) and three types of transactions (withdrawal, deposit and check balance) are described in the model. Suppose there is a bug on the Transition T20, where the variable 'cb' should be 'sb'.

To test the ATM system, a test case, a sequence of events with input values associated with these events, is executed. Assume the test case is Card(1234,100,200), PIN(1234), Spanish(), Savings(), Balance(), Receipt(), Done(), Exit(). When the EFSM of the ATM is executed on this input, the screen shows "Balanza=200" but the receipt prints "Balanza=100". Obviously this is not the expected output as the two balances are not equal.

To locate the bug, we first check the sequence of transitions traversed by the test case, which is T1, T2, T4, T6, T8, T20, T22, T10 and T23. As backward slicing is considered, we choose the slicing criterion by taking the transition from the traversed sequence of transitions in reverse order. T23 and T10 do not contain any variables, then T22 is taken as the slicing criterion. Figure 13 gives the slice in terms of marked transitions in the model. It can be observed that transitions T7, T13 and T14, which are related the checking account, are included in the slice. This is obviously wrong, as the slicing criterion T22 is printing receipt for saving account.

To investigate how these three transitions are included in the slicing, Figure 14 shows a sub-dependence graph of the ATM EFSM model with respect to transition T22. It can be seen from the figure, T22 is dependent upon T20, and T20 is dependent on T13 and T14. Therefore, the bug could be on T20. Further inspection on T20 shows that the variable 'cb' should be 'sb'.

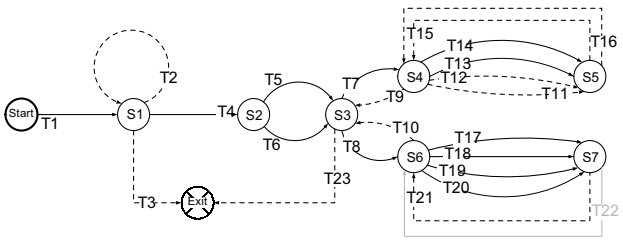


Fig. 13. Marked ATM model with slice criterion of the Transition T22(NTICD+DD), where solid directed edges represent the transitions in the slice.

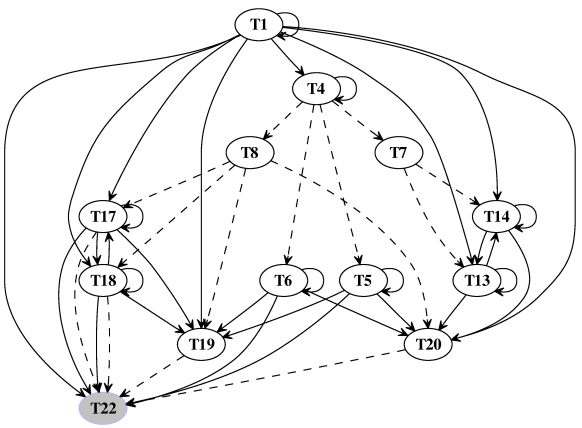


Fig. 14. Sub-dependence graph for ATM model with slice criterion of the Transition T22. The figure was produced by the tool where nodes represent EFSM transitions, solid directed edges represent the data dependence and dotted directed edges are control dependence between transitions.

## 10 Experiments

For EFSM slicing to be useful in practice, it is important to be able to reduce the EFSM by removing

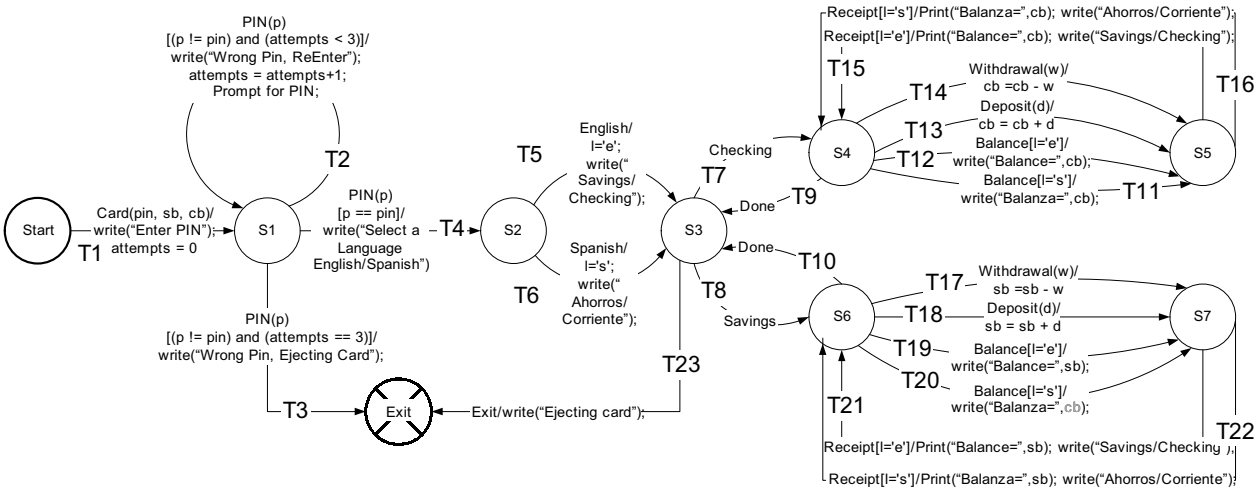


Fig. 12. The EFSM model of ATM presented in [43] with a bug highlighted on Transition T20.

unmarked transitions, merging equivalent states and reconnecting the graph. In this section, we report on the results of experiments designed to answer the following questions.

1) How much does our algorithm **reduce the graph size after slicing**?
2) According to each proposed measure of EFSM reduction, does our algorithm ever create **slices whose size is greater than the original**? If yes, how often do these occur ?
3) Is the **run-time cost** of slicing reasonable ?
4) How does our algorithm **compare to Korel *et al.*'s slicing algorithm and reduction rules** ?

| | EFSM Models | #S | #T | #UT | #Var |
|---|---|---|---|---|---|
| **G1** | ATM [43] | 9 | 23 | 21 | 8 |
| | Cashier | 12 | 21 | 20 | 10 |
| | CruiseControl [42] | 5 | 17 | 15 | 18 |
| | FuelPump [42] | 13 | 25 | 23 | 12 |
| | PrintToken | 11 | 89 | 42 | 5 |
| | VendingMachine | 7 | 28 | 22 | 7 |
| **G2** | INRES protocol [15] | 8 | 18 | 16 | 8 |
| | TCP [62] | 12 | 57 | 46 | 31 |
| | TCSbin(Motorola) | 24 | 65 | 56 | 61 |
| **G3** | DoorController [54] | 6 | 12 | 12 | 1 |
| | **Total** | 107 | 355 | 273 | 161 |

TABLE 2
Experimental Models.

### 10.1 Experimental setup

We have selected ten EFSM models from a variety of sources. The first six models were used by Korel et al. [42], [43] in their research of state-based slicing with traditional dependence analysis. They all contain a start state and every path must end in an exit state. The last four models do not contain exit states. INRES [15] and DoorControl [54] come from previous model-based studies, while TCP [62] and TCSbin are extracted from Specification and Description Language (SDL) specifications [14]. TCSbin is an industry model provided to us by Motorola. Since SDL specifications are richer than EFSMs, these models were transformed to remove 'history' and 'all' state types.

Table 2 provides the model's size in terms of the number of transitions (**#T**), number of states (**#S**) and number of transitions with unique labels (i.e. unique transitions (**#UT**)). Also, it provides the number of variables that appear on the transition labels.

We divide the EFSM models into groups, shown in Table 2, based on common characteristics of their structure. The G1 models are all free of control sinks and contain exit states; G2 models are large control sinks except for a transition from the start state; the single G3 model consists of a large control sink, with three transitions from the start state.

We use three algorithms for the experiments: 1) our slicing algorithm, which is applied to all ten models, using either NTSCD, NTICD or UNTICD, 2) Korel *et al.*'s slicing algorithm (Koreletal1), which can only be applied to the models in $G1$ because of their definition of control dependence, which is limited to EFSMs with an exit state, and 3) Korel *et al.*'s reduction rules (Koreletal2) combined with NTSCD, NTICD, or UNTICD for dependence analysis, which can be applied to all ten models. An algorithm, with its choice of control dependence, is applied to a model by considering each of its transitions and the set of variables referenced by the transition as the slicing criterion. Therefore, the number of slices generated when an algorithm is applied is equal to the number of transitions in the model.

### 10.2 Experimental results and discussion

To answer the first two questions, we examine the size of the slices generated using our algorithm parameterised by each of NTSCD, NTICD or UNTICD. Table 3 presents the results for each of these three forms of control dependence with data dependence. Slice size is measured in terms of number of states, number of transitions and number of unique transitions. In the table, each value is normalised by dividing by the corresponding size measure of the model. For example, $\mathsf{Avg}_S$ is the average number of states of all slices in a model over the total number of states of the model.

The average slice size reported in Table 3 shows that the slice size using NTICD is about one third of the model in terms of number of transitions and unique transitions, which is similar to a typical program slice size [12]. The slices using NTSCD are the largest in all measures, as NTSCD captures dependencies within control sinks while NTICD does not.

The average slice size using UNTICD is between that using NTSCD and NTICD. However, if we consider each model, it can be seen that the average slice size of slices using UNTICD is the same as that of slices using NTICD for the models in $G1$ and that of slices using NTSCD for models in $G2$. This confirms that UNTICD is the same as NTICD outside of control sinks, as all models in $G1$ do not contain control sinks and UNTICD is the same as NTSCD in control sinks, as each model in $G2$ is a large control sink [4].

It can be seen from Table 3 that the slice size using NTICD is smaller than that using NTSCD when averaged over all models. However, inspection of the average slice size for each model reveals an exception, where the $\mathsf{AVG}_T$ of Cashier using NTICD is a little larger than that using NTSCD. Note that the $\mathsf{AVG}_{UT}$ using NTICD is still smaller, so the size increase is caused by $\varepsilon$-elimination that might add transitions. However, this is only one such case (over ten subjects), and the average slice size for Table 3, calculated over all models and control dependency definitions and incorporating the average reductions for transitions, unique transitions and nodes, is 55% of the original model.

The run-time cost of slicing consists of two components: the time to build the dependence graph (which is a one-off cost), and the time to compute the slice for a specific criterion. Table 4 reports $\mathsf{Time}_{Dep}$ for computing dependence graph and the average $\mathsf{AvgTime}_{Slicing}$ for computing a slice for each model using a laptop with Intel(R)Core Duo CPU at 2.4GHz and 4GB memory. $\mathsf{AvgTime}_{Slicing}$ only considers the time taken for the traversal of the dependence graph, $\varepsilon$-elimination and minimisation.

$\mathsf{Time}_{Dep}$ reported in Table 4 reveals that NTSCD and UNTICD are more expensive than NTICD, since both NTSCD and UNTICD capture more dependencies than NTICD [4].

The average execution time $\mathsf{AvgTime}_{Slicing}$ reported in Table 4 shows that for most models, the slice is computed in very reasonable time. The slices generated for TCSbin, using our algorithm with NTSCD, takes the longest time to compute. This is because it is an industry model and it contains the largest number of states and each transition contains many variables and actions.

To answer the final question, we compare the slices generated by our algorithm, Koreletal1 and Koreletal2. Figure 15 shows the size difference between slices using our algorithm and Koreletal1 algorithm over all slices of the models in $G1$, plotted in monotonically increasing order on the x-axis. NTICD, NTSCD and UNTICD are used in the dependence analysis of our algorithm respectively. The x-axis measures the number of slices as a percentage of the total number of slices. The y-axis measures the size difference in number of states, transitions and u. Zero means that the two slices generated using the two algorithms with respect to the same criterion are equal in size. A positive value means our algorithm produces the smaller slice. The three lines represent the three measures in terms of the number of states, number of transitions and number of unique transitions as labelled. The slice size using NTICD and UNTICD is the same for the models in $G1$ so we have produced the graphs for NTICD only..

As discussed in Section 8, the Koreletal1 algorithm may produce relatively large slices. Figure 15 shows that when using NTICD, in more than 40% slices, we produced smaller slices, ranging from 1 to 23 in term of the number of transitions, 1 to 11 in terms of the number of states and 1 to 21 in terms of the number of unique transitions (indicated by the part of three lines which is above 0 on the y-axis in the top graph of Figure 15). In only 10% of the slices did Koreletal1 perform a little better when measuring the size using the number of transitions where the maximum is 4 transitions (indicated by the part of grey lines which is below 0 on y-axis in the top graph of Figure 15). When using NTSCD, our algorithm never performs worse than Koreletal1.

Koreletal2 uses reduction rules that produce relatively small slices. However Figure 16 shows that more than 30% of slices using our algorithm are smaller than those using Koreletal2 in all measures. A further inspection of the data reveals that most of the slices with the greatest reduction occur in TCP and TCSbin, which are large models without exit states. In PrintToken which is also a large model but with fewer states and an exit state, the slice sizes are the same for most transitions. This result suggests that our algorithm can produce significantly smaller slices for larger, non-terminating models.

In the slices generated by the three algorithms when using NTICD, we observe the best and worst case

| | NTICD+DD | | | NTSCD+DD | | | UNTICD+DD | | |
|---|---|---|---|---|---|---|---|---|---|
| Model | $Avg_S$ | $Avg_T$ | $Avg_{UT}$ | $Avg_S$ | $Avg_T$ | $Avg_{UT}$ | $Avg_S$ | $Avg_T$ | $Avg_{UT}$ |
| ATM | 46% | 29% | 24% | 53% | 36% | 31% | 46% | 29% | 24% |
| Cashier | 77% | 76% | 70% | 77% | 68% | 75% | 77% | 77% | 71% |
| CruiseControl | 78% | 70% | 80% | 78% | 70% | 80% | 78% | 70% | 80% |
| FuelPump | 20% | 10% | 10% | 38% | 27% | 29% | 20% | 9% | 10% |
| PrintToken | 91% | 60% | 78% | 91% | 60% | 78% | 91% | 60% | 78% |
| VendingMachine | 56% | 54% | 42% | 84% | 80% | 87% | 56% | 54% | 42% |
| INRES | 26% | 25% | 18% | 73% | 60% | 67% | 73% | 60% | 67% |
| TCP | 16% | 14% | 9% | 45% | 48% | 52% | 45% | 48% | 52% |
| TCSbin | 6% | 9% | 6% | 70% | 81% | 79% | 70% | 81% | 79% |
| DoorController | 24% | 13% | 12% | 85% | 57% | 57% | 60% | 38% | 38% |
| Average | 46% | 35% | 37% | 71% | 61% | 67% | 66% | 57% | 61% |

TABLE 3
Slice size for our slicing algorithm.

| | NTICD+DD | | NTSCD+DD | | UNTICD+DD | |
|---|---|---|---|---|---|---|
| Model | $Time_{Dep}$ | $AvgTime_{Slicing}$ | $Time_{Dep}$ | $AvgTime_{Slicing}$ | $Time_{Dep}$ | $AvgTime_{Slicing}$ |
| ATM | 0.039s | 0.023s | 0.048s | 0.044s | 0.038s | 0.024s |
| Cashier | 0.023s | 0.439s | 0.023s | 0.480s | 0.022s | 0.433s |
| CruiseControl | 0.049s | 0.049s | 0.070s | 0.049s | 0.048s | 0.049s |
| FuelPump | 0.054s | 0.012s | 0.049s | 0.064s | 0.049s | 0.012s |
| PrintToken | 10.164s | 1.764s | 18.408s | 1.732s | 10.261s | 0.559s |
| VendingMachine | 0.067s | 0.041s | 0.117s | 0.179s | 0.069s | 0.042s |
| INRES | 0.020s | 0.007s | 0.023s | 0.070s | 0.023s | 0.071s |
| TCP | 5.222s | 0.024s | 7.560s | 0.558s | 10.911s | 0.559s |
| TCSbin | 45.350s | 0.184s | 48.005s | 15.234s | 58.999s | 14.758s |
| DoorController | 0.007s | 0.002s | 0.007s | 0.035s | 0.006s | 0.009s |
| Average | 6.100s | 0.255s | 7.431s | 1.844s | 8.043s | 1.767s |

TABLE 4
Execution time for our slicing algorithm.

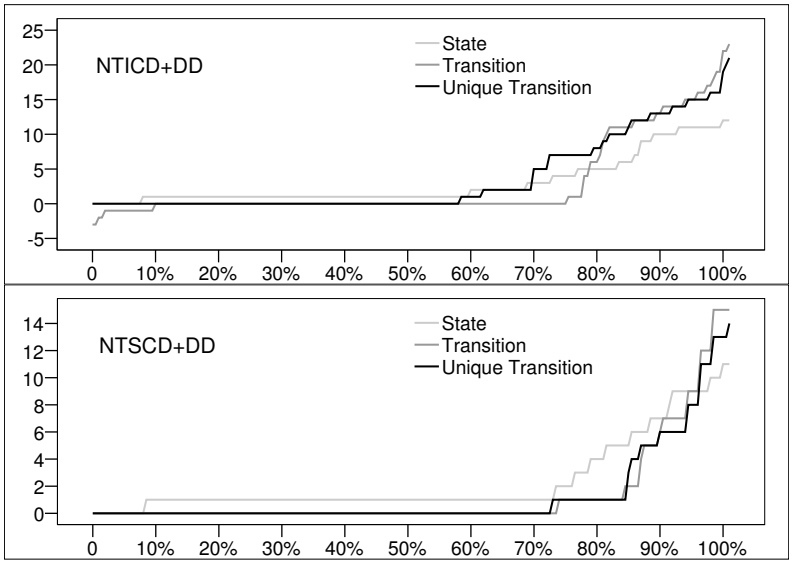


Fig. 15. Comparison between slice sizes for models in G1 using our slicing algorithm and Koreletal1.

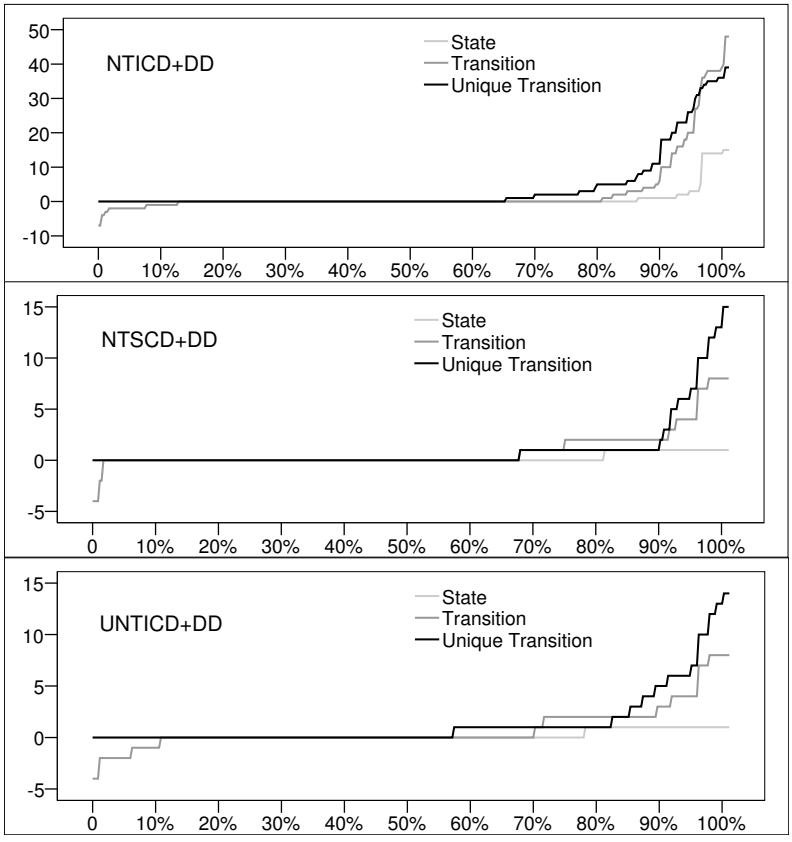


Fig. 16. Comparison between slice sizes for all ten models using our slicing algorithm and Koreletal2.

| | | Our Algorithm | | | Koreletal2 | | |
|---|---|---|---|---|---|---|---|
| Model | Criterion | #S | #T | #UT | #S | #T | #UT |
| ATM | T21 | 6 | 16 | 10 | 6 | 12 | 12 |
| TCSbin | T23 | 1 | 1 | 1 | 15 | 49 | 40 |

TABLE 5
The first row shows the case where Koreletal2 most outperforms ours; the second row where our algorithm most outperforms Koreletal2.

relative to our algorithm using as a measure the number of transitions (i.e. the peak point and bottom point of the dark grey line in Figure 15 and Figure 16). We define the best case to be the slice that has the smallest number of transitions; and the worst case to be the slice that has the greatest number of transitions.

Table 5 shows details for the best case and worst case with respect to slice sizes. The worst case occurs in the ATM when slicing with respect to T21 and its variables. The slice using our algorithm returns 6 states, 16 transitions and 10 unique transitions, while the slice using Koreletal2 returns the same number of states, more unique transitions but fewer transitions overall. This is owing to the $\varepsilon$-elimination phase of our algorithm. Removing unmarked transitions with the algorithm sometimes leads to copying marked transitions in order to guarantee invariant behaviour by the model slice. If these are not minimised later on the number of transitions may increase, but not the number of states or the number of unique transitions. The best case occurs in TCSbin. The slice generated using our slicing algorithm with respect to transition T23 and its variables consists of only one state and one transition. This is because no other transitions in TCSbin are control or data dependent on T23. However, Koreletal2 reduction rules cannot remove these unmarked transitions.

## 11 Related work

Besides Korel *et al.* [43], there have been other approaches to slicing of state-based models. Heimdahl *et al.* [31] present a slicing approach for Requirements State Machine Language (RSML) specifications, that describes hierarchical and concurrent state machines. Two slicing algorithms are defined based on a marking of the abstract syntax tree, similar to Sloane's and Holdworth's approach [53]. First, the specification is reduced based on a specific scenario of interest by removing all behaviours that are not possible when the conditions defining the scenario are satisfied. Then, slicing based on data flow and control flow is subsequently applied, with respect to a transition or variable, to the remaining specification.

Wang *et al.* [58] describe a slicing approach for Extended Hierarchical Automata (EHA) [19]. They define the following dependence relations for handling

hierarchy, concurrency, and communication: sequential, parallel and refinement data dependence, as well as synchronisation, transition and refinement control dependence. The slicing algorithm, given with respect to states or transitions (as slicing criteria), traverses the EHA finding elements to keep that are dependent on the slicing criterion. The slicing algorithm described in [46] extends Wang *et al.*'s slicing approach by describing how to remove false dependencies (elements identified as being dependent on each other when they should not be) and also how to slice a collection of state chart models (expressed as EHA).

Luangsodsai and Fox [21], [48] describe an approach that uses augmented And-Or dependence graphs [44] for slicing concurrent state charts. An And-Or dependence graph is used to represent data, control dependencies as well as interference data dependence, parallel, and interference control dependence. The slicing algorithm is defined as a graph reachability problem on the And-Or dependence graph with respect to the slicing criterion, which is a state, a condition, an event, or an action.

Except for Korel *et al.*'s reduction rules [43], the slicing algorithms described above [21], [31], [46], [48], [58] are not amorphous. They all produce slices that are syntactic sub-models of the original i.e. they do not remove elements that break the connectivity of the model. Thus, they do not suffer from the problem of re-connecting the graph. However, slices produced can be large and may contain model elements in addition to those marked when considering the transitive dependencies for all dependence relations defined. Our algorithm aims to improve on these by focusing on producing smaller and more precise slices.

Labbé *et al.* [45] have developed a tool for slicing communicating automata, in particular Input/Output Symbolic Transition Systems (IOSTSs) [25]. The slicing criterion is a set of transitions. The slicing algorithm first constructs a dependence graph based on their definitions of data, control, and interference dependence (that identifies dependences between concurrent parts of the model). All nodes backwardly reachable from the slicing criterion are marked. This approach does not take a state machine and produce a reduced state machine. Rather, it marks states and transitions that can influence the slicing criterion. As this paper has demonstrated, the process of turning a dependence analysis into a sliced EFSM is far from trivial; it is related to the problem of minimisation in automata theory.

In general the transition minimisation problem is PSPACE-complete [39]. Another approach concentrates on Thompson NFA, which are NFA that can result from one of the standard approaches to converting a regular expression into an NFA (with $\varepsilon$ transitions) [61]. An NFA is a Thompson NFA if for every state $s$ and label $a$ there are at most two transitions starting at $s$ that have label $a$. A linear time algorithm is given that minimises a Thompson NFA, where the notion of minimisation used is that as many states are eliminated as possible while retaining the Thompson property [61]. However, while our NFA may be Thompson NFAs, we are not interested in retaining this property as the approach limits our freedom in minimisation and potentially may lead to an overly large slice. In summary, it appears that currently there are relatively few approaches that minimise an NFA while potentially retaining $\varepsilon$-transitions and they have been defined for forms of state machines that are not well suited for slicing. As a result it is hard to determine how well they will perform using EFSMs produced in practice and the implementation and evaluation of such approaches is a topic for future work.

Calude *et al.* [16] take an entirely different approach to defining minimality, considering the effort required to simulate an NFA. This corresponds to the effort required to determine the set of possible states of an NFA after a sequence. Interestingly, under this notion of minimality, for any NFA there is a minimal NFA that is unique up to isomorphism [16]. Currently, however, there appear to be no reported algorithms or complexity results for this notion of minimality.

## 12 Conclusions

This paper introduced a set of related slicing algorithms based on a set of different EFSM control dependence definitions and a tool for EFSM slicing. All of our algorithms are capable of slicing non-deterministic, non-terminating EFSMs and each is adapted from the Ilie and Yu NFA minimisation algorithm parameterised by one of three different notions of control dependence. The paper proved the suitability of properties of the algorithm for slicing and presented a detailed empirical study on ten EFSM models, using standard benchmarks and an industrial EFSM from production systems. The study compared our three algorithms to Korel *et al.*'s slicing algorithm. It also compared our adapted NFA minimisation algorithm with an adaptation of Korel *et al.*'s reduction rules.

The results of the empirical study indicate that our algorithm can significantly reduce the size of EFSM slices. The slices that use control dependence definitions that are non-termination insensitive produce the smallest average slices size of 35%. Compared to Korel *et al.*'s slicing algorithm, our average slice is smaller 40% of the time and larger only 10% of the time. Compared to Korel *et al.*'s reduction rules, our algorithm produced smaller slices in more than 30% of slices. Because our algorithm is an adaptation of the Ilie and Yu NFA minimisation algorithm, in the worst cases the number of transitions may increase to be larger than in the original model. However, the empirical study showed that this did not occur for the ten EFSM models used in the study.

## ACKNOWLEDGEMENTS

This research work is supported in part by EPSRC Grant EP/F059442/1 and National Natural Science Foundation of China under Grant No.60903002 and No.61170082. Author order is alphabetical.

## REFERENCES

[1] H. Agrawal. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida, June 20–24 1994. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[2] M. H. Albert and S. Linton. A practical algorithm for reducing non-deterministic finite state automata. Technical Report OUCS-2004-11, University of Otago, 2004.

[3] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, Z. Li, and L. Tratt. State-based model slicing: A survey. *ACM Computing Surveys*, to appear.

[4] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt. Control dependence for extended finite state machines (best theory paper award winner). In *Fundamental Approaches to Software Engineering (FASE '09)*, volume 5503, pages 216–230, York, UK, Mar. 2009. Springer LNCS.

[5] K. Androutsopoulos, N. Gold, M. Harman, Z. Li, and L. Tratt. A theoretical and empirical study of EFSM dependence. In $25^{th}$ *IEEE International Conference on Software Maintenance (ICSM 2009)*, Edmonton, Alberta, Canada, 23rd–26th September 2009.

[6] P. K. Au and J. M. Atlee. Evaluation of a state-based model of feature interactions. In *Feature Interactions in Telecommunications Networks IV (FIW 97)*, pages 153–167, Montréal, Canada, 1997. IOS Press.

[7] T. Ball and S. Horwitz. Slicing programs with arbitrary control–flow. In P. Fritzson, editor, $1^{st}$ *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993. Springer. Also available as University of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[8] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini. Model-based generation of testbeds for web services. In $8^{th}$ *International Workshop on Formal Approaches to Testing Software (FATES '08)*, volume 5047 of *Lecture Notes in Computer Science*, pages 266–282. Springer, 2008.

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

[10] D. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[11] D. Binkley and M. Harman. An empirical study of predicate dependence levels and trends. In $25^{th}$ *IEEE International Conference and Software Engineering (ICSE 2003)*, pages 330–339, Los Alamitos, California, USA, May 2003. IEEE Computer Society Press.

[12] D. Binkley and M. 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, California, USA, Sept. 2003. IEEE Computer Society Press.

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

[14] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Specification and description language (SDL),WebPro Forum Tutorial, Int. Eng. Consortium.

[15] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols. In *IWTCS'97*, pages 75–90, 1997.

[16] C. Calude, E. Calude, and B. Khoussainov. Finite nondeterministic automata: Simulation and minimality. *Theoretical Computer Science*, 242(1–2):219–235, 2000.

[17] M. Caporuscio, A. D. Marco, and P. Inverardi. Model-based system reconfiguration for dynamic performance management. *Journal of Systems and Software*, 80(4):455–473, 2007.

[18] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[19] W. Dong, J. Wang, X. Qi, and Z.-C. 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.

[20] European Union. ARTEMIS programme embedded computing systems call for proposals, 2009. Available online at https://www.artemis-ju.eu/.

[21] C. Fox and A. Luangsodsai. And-or dependence graphs for slicing statecharts. In *Beyond Program Slicing*, Dagstuhl, Germany, 2005.

[22] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[24] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed systems based on UML models. In *International Conference on Software Engineering (ICSE)*, pages 391–400, 2006.

[25] C. Gaston, P. L. Gall, N. Rapin, and A. Touil. Symbolic execution techniques for test purpose definition. In *Proceedings of Testing of Communicating Systems: 18th IFIP TC 6/WG 6.1 International Conference, TestCom*, pages 1–18, New York, NY, USA, May 16-18 2006. Springer.

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

[27] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering Methodology*, 5(4):293–333, 1996.

[28] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

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

[30] M. Harman, A. Lakhotia, and D. Binkley. A framework for static slicers of unstructured programs. *Information and Software Technology*. To appear.

[31] M. P. E. Heimdahl and M. W. Whalen. Reduction and slicing of hierarchical state machines. In *Proc. Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Switzerland, 1997.

[32] J. E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The theory of Machines and Computation*, pages 189–196. Academic Press, 1971.

[33] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, Reading, MA, 1969.

[34] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[35] J. Hromkovic and G. Schnitger. Comparing the size of NFAs with and without epsilon-transitions. *Theoretical Computer Science*, 380(1–2):100–114, 2007.

[36] L. Ilie, G. Navarro, and S. Yu. On NFA reductions. In *Theory Is Forever*, pages 112–124, 2004.

[37] L. Ilie and S. Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.

[38] L. Ilie and S. Yu. Reducing NFAs by invariant equivalences. *Theoretical Computer Science*, 306(1-3):373–390, 2003.

[39] S. John. Minimal unambiguous eNFA. In *Implementation and Application of Automata, 9th International Conference, CIAA 2004, Canada, 2004, Revised Selected Papers*, volume 3317 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 2004.

[40] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1999.

[41] R. Komondoor and S. 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., Jan. 19–21 2000. ACM Press.

[42] B. Korel, G. Koutsogiannakis, and L. H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 34–43, USA, 2007. ACM.

[43] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state based models. In *IEEE International Conference on Software Maintenance (ICSM'03)*, pages 34–43, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[44] 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, New York, NY, USA, 1981. ACM.

[45] S. Labbé and J.-P. Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing*, 20(6):563–595, 2008.

[46] S. V. Langenhove and A. Hoogewijs. $SV_tL$: 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.

[47] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In $22^{nd}$ *IEEE/ACM international conference on Automated Software Engineering (ASE 07)*, pages 417–420, New York, NY, USA, 2007. ACM.

[48] A. Luangsodsai and C. Fox. Concurrent Statechart Slicing. In *Computer Science and Eelectronic Eengineering Cinference (CEEC)*, pages 1–7, September 2010.

[49] F. Massicotte, M. Couture, L. C. Briand, and Y. Labiche. Model-driven, network-context sensitive intrusion detection. In *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2007.

[50] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *European Symposium on Programming*, pages 77–93, 2005.

[51] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. 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.

[52] Y. Sivagurunathan, M. Harman, and S. Danicic. Slicing, I/O and the implicit state. In M. Kamkar, editor, $3^{rd}$ *International Workshop on Automated Debugging ( AADEBUG'97 )*, volume 2 of *Linköping Electronic Articles in Computer and Information Science*, pages 59–65, Linköping, Sweden, May 1997.

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

[54] F. Strobl and A. Wisspeintner. Specification of an elevator control system – an autofocus case study. Technical Report TUM-I9906, Technische Univerität München, 1999.

[55] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

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

[57] 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.

[58] J. Wang, W. Dong, and Z.-C. 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.

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

[60] M. Weiser. Program slicing. In $5^{th}$ *International Conference on Software Engineering*, pages 439–449, San Diego, CA, Mar. 1981.

[61] G. Xing. Minimized thompson NFA. *Int. J. Comput. Math.*, 81(9):1097–1106, 2004.

[62] R. Y. Zaghal and J. I. Khan. EFSM/SDL modeling of the original TCP standard (RFC793) and the congestion control mechanism of TCP Reno. Technical Report TR2005-07-22, Internetworking and Media Communications Research Laboratories, Department of Computer Science, Kent State University, 2005.


**Kelly Androutsopoulos** received a MEng in computer Science at Imperial College and a PhD in computer Science from King's College London. She is currently a research associate on the SLIM (SLIcing state based Models) project funded by EPSRC at the University College London. Her research interests include modelling with state-based languages, static and dynamic analysis, specification and verification of reactive systems.

**David Clark** David Clark is a Senior Lecturer in the Department of Computer Science at University College London. He researches semantics based program analysis and is well known for his work on quantifying information flow using information theory. He also has a track record of research into state based models and their semantics.

**Mark Harman** is professor of Software Engineering in the Department of Computer Science at University College London where he directs the CREST centre. He is widely known for work on source code analysis and testing and as a founder of the field of Search Based Software Engineering, an area of Software Engineering research on which he has given 16 keynote talks in the past five years.

**Robert Hierons** received a BA in Mathematics (Trinity College, Cambridge), and a PhD in Computer Science (Brunel University). He then joined the Department of Mathematical and Computing Sciences at Goldsmiths College, University of London, before returning to Brunel University in 2000. He was promoted to full Professor in 2003.

**Zheng Li** is a professor at Beijing University of Chemical Technology. He obtained his PhD from King's College London, CREST centre in 2009. He has worked as a research associate at King's College London and University College London. He has worked on program regression testing, dependence analysis and source code analysis and manipulation. More recently he is interested in search-based software engineering and slicing state-based models.

**Laurence Tratt** is a Lecturer in the Department of Informatics at Kings College London in the UK. He is an Associate Editor in Chief of IEEE Software and sits on the Editorial Board of The Journal of Object Technology.