Correctly Slicing Extended Finite State Machines

Torben Amtoft[1], Kelly Androutsopoulos[2], and David Clark[3]

[1] Kansas State University, Manhattan KS 66506, USA
tamtoft@ksu.edu
[2] Middlesex University, London, NW4 4BT, United Kingdom
K.Androutsopoulos@mdx.ac.uk
[3] University College London, WC1E 6BT, United Kingdom
david.clark@ucl.ac.uk

Abstract. We consider slicing extended finite state machines. Extended finite state machines (EFSMs) combine a finite state machine with a store and can model a range of computational phenomena, from high-level software to cyber-physical systems. EFSMs are essentially interactive, possibly non-terminating or with multiple exit states and may be nondeterministic, so standard techniques for slicing, developed for control flow graphs of programs with a functional semantics, are not immediately applicable.
This paper addresses the various aspects of correctness for slicing of EFSMs, and provides syntactic criteria that we prove are sufficient for our proposed notions of semantic correctness. The syntactic criteria are based on the "weak commitment" and "strong commitment" properties highlighted by Danicic et alia. We provide polynomial-time algorithms to compute the least sets satisfying each of these two properties. We have conducted experiments using widely-studied benchmark and industrial EFSMs that compare our slicing algorithms with those using existing definitions of control dependence. We found that our algorithms produce the smallest average slices sizes, 21% of the original EFSMs when "weak commitment" is sufficient and 58% when "strong commitment" is needed (to preserve termination properties).

Keywords: Extended Finite State Machines · Slicing

1 Introduction

Program slicing was introduced in 1979 by Weiser [34, 33] to remove the parts of a program that are irrelevant in a given context. Given a slicing criterion <p, V>, where p is a program point and V is a subset of program variables, the slice consists of all the statements that may directly or indirectly affect the slicing criterion (determined by computing dependences on control flow graphs). Weiser's notion of slicing is executable (the slice is not just a closure of statements but can be compiled and run), static (all possible executions are considered) and backward (considers dependences backwards from the slicing criterion). Since then,

there have been hundreds of papers on program slicing, and several surveys [31, 35, 28] that illustrate its wide application in many areas including compiler optimizations, debugging and reverse engineering.

Researchers have moved from considering only program slicing to model slicing [3, 19, 23, 13], because a significant amount of software production is done with models, in particular specification and design. Models describe many types of information better than programs but can become large and unmanageable more quickly. Applications of slicing at the model level include model checking, comprehension, testing and reuse.

We consider slicing extended finite state machines. Extended finite state machines (EFSMs) and their variants (e.g. Harel's [15] and UML state machines) are now widely used to model dynamic behaviour of cyber physical systems [10], embedded systems [24] and product lines [23, 11]. EFSMs combine a finite state machine with a store, i.e. a set of program variables that can be updated via commands attached to transitions between the atomic states. EFSMs can model a range of computational phenomena, from high-level software to cyber-physical systems. They are essentially interactive and may be nondeterministic and possibly without exit points (or having multiple such). For EFSMs it is more natural to let slices be sets of transitions rather than sets of nodes (atomic states) [20].

One might imagine, given the long history of both slicing and finite state machines, that the correct slicing of finite state machines is a closed question or, if not closed, a relatively uninteresting one. However, neither consideration is true when aiming to produce correct, executable slices, that is to say reduced "programs" for which any nodes not relevant to the restricted semantics have been removed and the "program" has been "rewired" to maintain executability (the slice can be compiled and run). What makes EFSMs particularly interesting from a theoretical perspective is that their interactive, event driven semantics offers a simplified laboratory in which to analyse the correctness of slicing in the presence of interaction with an environment.

Work on program slicing has largely focused on programs with a functional semantics, and notions of correctness assume interaction only at the initial input. But in the context of interaction with a dynamic environment, traditional slicing algorithms produce incorrect slices. To see this, consider the example program illustrated in Figure 1. If the environment generates the input sequence 1,2,3,4,..., the value of z in program P is 4 (as x=1, y=2, z=3). If P is sliced with respect to $<5, \{z\}>$, line 2 will be removed as line 4 is not (data) dependent on it. For the same input sequence the value of z in the slice will be 3 (as x=1, z=2), which does not satisfy Weiser's correctness property.

In the context of model slicing, any work has to address interaction with environmental events. A critical line of research leading to this paper has been contributions on how to slice correctly when the input sequence is part of the implicit state and not described by any program variable.

Sivagurunathan et al. [29] address this issue by sketching a program transformation technique to be applied before the application of traditional slicing algorithms in order to make the implicit state explicit by modeling it with addi-

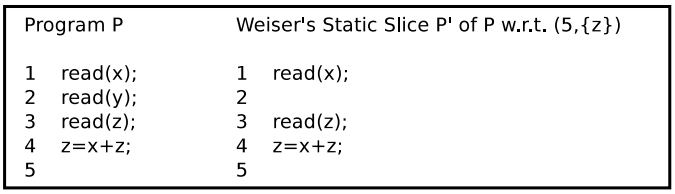

Fig. 1. A simple interactive program and its slice.

tional variables. The limitations in that work are that a suitably general set of transformation rules is not given and the programs to which it can be applied assume a single exit on which the definition of control dependence rests.

Ganapathy et al. [12] defined correctness for slicing reactive programs, where events (in particular generated events) are part of the slicing criterion. However, not all interactive systems generate events, and the drawback of their work is the large size of their slices: they include all states and transitions that reach the event of interest. Labbé et al. [21] presented polynomial algorithms for slicing Input/Output Symbolic Transition Systems (IOSTSs). However, they do not state correctness properties, and consider only a control dependence [26] that is sensitive to non-termination.

Korel et al. [20] considered slicing of EFSMs and proposed new correctness criteria, in particular they made a key observation: in order for the sliced EFSM to have behavior similar to the original EFSM, certain events must be allowed to be "idle" (the term "stuttering" is also used), that is, not trigger transitions. For example, in Fig. 1 we would have 3 kinds of events: `read-x`, `read-y`, and `read-z`; the sliced program would be idle on the event `read-y` and hence an input sequence `read-x(1), read-y(2), read-z(3)` would result in 1 and 3 being read, but 2 ignored. We build on the work of Korel et al. [20] and our approach can be viewed as a formalisation and validation of their correctness notion, but extended to handle even EFSMs with no (or with multiple) exit states.

Like EFSMs, modern programs may have no exit points (or multiple such) so as to program indefinitely non-terminating, reactive systems. This has required the development of new definitions of control dependence for programming languages and models [26, 27, 1, 4]. Definitions of control dependence began to proliferate until Danicic and others extracted order out of chaos by focusing on the desirable *properties* of the resulting dependence relations rather than the control dependence definitions themselves [9]:

- *Weak Commitment Closure* (WCC) means that each node has *at most one* "next observable", where an observable is a node relevant to the slicing criterion.
- *Strong Commitment Closure* (SCC) in addition demands that from a node that has a "next observable", there is no way to infinitely avoid that observable.

EFSMs, being models rather than programs, also allow non-determinism. An early attempt to handle non-determinism by Hatcliff et alia considers a multi-threaded language but does not prove any correctness results [16].

Any correct algorithm for slicing, in formalisms based on finite state models, has to consider all of the above issues: interaction with the environment; non-determinism; and a suitable notion of control dependence. This paper is the first that successfully and generally deals with all these issues to develop algorithms that we can prove produce (the least) slices for EFSMs that satisfy certain correctness properties. In the process, we establish that the commitment closure approach to control dependence of Danicic et alia [9] is flexible enough to provide correctness for interactive programming languages with modern control structures and non-determinism.

The main contributions of this paper are:

1. A formalization (Section 3) of new definitions of correctness for slicing an EFSM (whose syntax and semantics are defined in Section 2) in the presence of interaction, non-determinism and non-termination. We require (Completeness) that the slice machine simulates the original machine, with some events now perhaps idle. In the other direction (Soundness), we cannot hope for a perfect simulation, but demand that for each "observable" step by the slice machine, either the original machine simulates it, perhaps inserting some events, or the original machine either *(i)* gets stuck, or *(ii)* loops (a possibility excluded by "strong soundness").
2. Syntactic criteria (Section 4) that allow us to prove (in Appendix A) the correctness properties from 1. It suffices to demand that the set of transitions in the slice satisfies two conditions: it must be closed under the well-known notion of data dependence, and it must have the abovementioned WCC property, first highlighted by Danicic et al but in this work adapted (in a non-trivial way) to EFSMs; to get strong soundness, we need the SCC property (likewise adapted to EFSMs). Observe that, unlike the classical notions of control dependence [6, 17, 25], we can handle control flow graphs without an exit point (or with multiple such).
3. Two novel polynomial-time algorithms for computing least slices for EFSMs (Section 5); one does it wrt. WCC and another wrt. SCC. (We discovered them in 2013 as part of a very early version[4] of this work, whereas Danicic et al presented in their groundbreaking work [9] only a very abstract algorithm — though in subsequent work they have presented algorithms that, while for different settings, are somewhat similar to our algorithms.)
4. A demonstration of the practical usefulness of the two algorithms over a suite of thirteen EFSMs, taken from textbook benchmark examples and real world production EFSM models, and experimental comparisons with existing control dependence definitions (Sections 6 and 7).

[4] Published as Research Note RN/13/22 from University College London.

2 Extended Finite State Machines (EFSMs)

We shall now formally describe the kind of Extended Finite State Machines (EFSMs) we shall consider in this paper. An EFSM M is a finite state automaton where transitions have guards, may manipulate a common store, and may be triggered by external events. In Section 2.1 we shall formally define their syntax and in Section 2.3 their semantics, which is designed so as to also allow reasoning about a slice as defined in Section 2.2. In Section 3 we shall discuss what it means for a slice to be semantically correct, and propose suitable definitions of correctness.

2.1 Syntax of EFSMs

An EFSM is a tuple $(\hat{S}, \hat{T}, \hat{E}, \hat{V})$ where $\hat{S}$ is a finite set of states (one state may be designated as the initial), $\hat{T}$ is a finite set of transitions, $\hat{E}$ is a finite set of events (some of which may take a parameter), and $\hat{V}$ is a finite set of variables; variables and parameters are assumed to be eventually bound to scalar values such as integers or strings. We shall employ naming conventions and in particular use n or m to range over states, and use t or u to range over transitions.

A transition t (or u) $\in \hat{T}$ has a source state $\mathsf{S}(t) \in \hat{S}$, a target state $\mathsf{T}(t) \in \hat{S}$, and a label of the form $\tilde{e}[g]/a$, each component of which we shall now describe:

- $\tilde{e} = \mathsf{E}(t)$ is either ε (the transition is "spontaneous"), of the form e where e is an event in $\hat{E}$ that does not take a parameter, or of the form $e(v_b)$ where e is an event in $\hat{E}$ that expects a parameter which will be bound to v_b (not occurring in $\hat{V}$).
- $g = \mathsf{G}(t)$ is a guard (which must be true in order for t to be chosen), and a is an action which we may assume is either of the form $v := A$ with $v \in \hat{V}$ or `skip`; here g and A may refer to the variables in $\hat{V}$ and also to v_b if it exists.

We define $\mathsf{D}(t)$, the variables defined by t, as $\{v\}$ if a is $v := A$ and $\emptyset$ if a is `skip`; we define $\mathsf{A}(t)$, the (arithmetic) expression mentioned in t, as A if a is $v := A$ and 0 (arbitrarily) if a is `skip`. We define $\mathsf{U}(t)$, the variables used by t, as (with fv/bv denoting free/bound variables) $(fv(\mathsf{G}(t)) \cup fv(\mathsf{A}(t))) \setminus bv(\tilde{e})$ where $bv(e(v_b)) = \{v_b\}$ and $bv(e) = bv(\varepsilon) = \emptyset$. When depicting an EFSM, all parts of a label are optional: one may omit $\tilde{e}$ if it is ε, omit g if it is `true`, and omit a if it is `skip`. For examples of labels on transitions, see Figure 3. A transition where all parts of its label can be omitted is called an ε-transition. Note that we do not allow for transitions that *produce* events.

Figure 2 depicts a non-trivial EFSM (with the initial state `S1`) that shall serve as our main running example. Since a main challenge of this paper is to provide a better understanding of control aspects, while employing a standard notion of control dependence (which is already well understood and rather non-controversial), we ignore the labels of the transitions.

We say that t is *self-looping* if $\mathsf{S}(t) = \mathsf{T}(t)$, and that u is a *successor* of t if $\mathsf{S}(u) = \mathsf{T}(t)$. In Figure 2, `t11` is self-looping, and `t5` is a successor of `t1`.

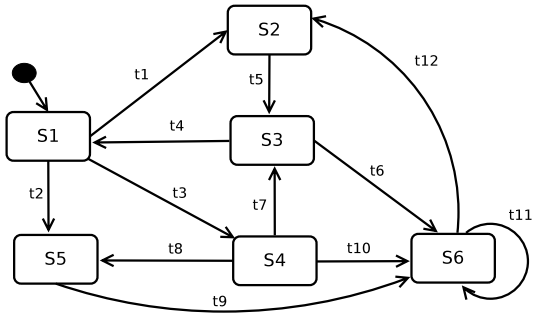

Fig. 2. Our running example EFSM.

We say that n is an *exit* state if no other node can be reached from n, that is, there is no transition t with $\mathsf{S}(t) = n$ and $\mathsf{T}(t) \neq n$. In Figure 2, there is no exit state. An exit state n may, or may not, be the source and target of a self-looping transition.

We say that $[t_1..t_k]$ ($k \geq 0$) is a path (of length k) if for all $j \in 1 \ldots k-1$, t_{j+1} is a successor of t_j. If $k \geq 1$, we say that $[t_1..t_k]$ is a path from $\mathsf{S}(t_1)$ to $\mathsf{T}(t_k)$ (if $k = 0$, then $[t_1..t_k]$ is a path from n to n for all n). We say that n occurs in $[t_1..t_k]$ if there exists $j \in 1 \ldots k$ such that $n = \mathsf{S}(t_j)$ or $n = \mathsf{T}(t_j)$. With L as set of transitions, we say that a path $[t_1..t_k]$ is outside L iff $t_j \notin L$ for all $j \in 1 \ldots k$. In Figure 2, [`t5,t4,t3`] is a path from `S2` to `S4` where the states `S2`, `S3`, `S1` and `S4` occur but which is outside say {`t1,t6`}. We say that $[t_1..t_k..]$ is an infinite path from $\mathsf{S}(t_1)$ if t_{j+1} is a successor of t_j for all $j \geq 1$; we say that the path avoids n if $n \neq \mathsf{S}(t_j)$ for all $j \geq 1$. In Figure 2, the path [`t4,t3,t7,t4,t3,t7,...`] is an infinite path from `S3` that avoids all of `S2`, `S5`, and `S6`.

2.2 Slicing EFSMs

In general, a slice is a subpart of the original program. In our setting, this raises the question: should we consider a slice to be a subset of the *states* (together with the transitions between these states), or to be a subset of the *transitions* (together with the states involved in those transitions). We shall (as Korel et al. [20]) choose the latter option, as the former is a special case (where a slice contains a transition iff it contains its source and also its target), thus:

Definition 1 (Slice Set) *A slice set for an EFSM* $(\hat{S}, \hat{T}, \hat{E}, \hat{V})$ *is a subset of* $\hat{T}$.

We must demand that a slice set contains the given *slicing criterion*, that is, those transitions that are required to be preserved by the slice. But in order for also the context of such transitions to be "invariant" under the slice, and thus ensure certain kinds of semantic correctness (as defined in Section 3), we must

in addition demand that the slice set includes the transitions that may "impact" the slicing criterion; this can be formalized as a demand that the slice set satisfies certain conditions (given in Section 4). We shall even aim for the slice set to be the least such set (in Section 5 we shall show how to achieve that).

Our definitions will often be implicitly parameterized with respect to a given EFSM, and with respect to a fixed slice set which is often called $\mathcal{L}$ while its members are called "observables". As will be formalized in Section 2.3, slicing amounts to keeping the transitions in $\mathcal{L}$ as they are, whereas transitions not in $\mathcal{L}$ are replaced by ε-transitions (all parts of the label are removed).

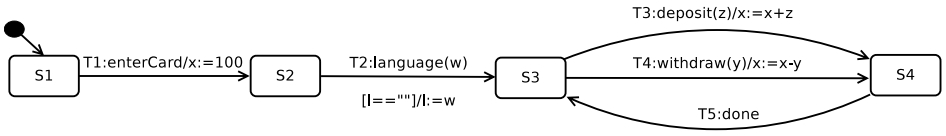

Fig. 3. A simplified EFSM for ATM operations.

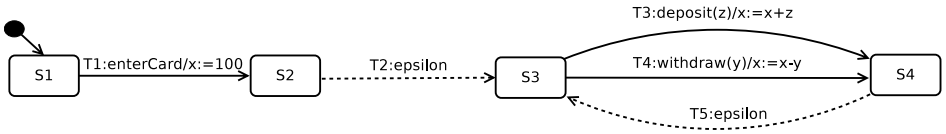

Fig. 4. The result of slicing the EFSM from Figure 3 with respect to `T1` and `T3` and `T4`.

Example 1 *Consider the EFSM in Figure 3 which models a simple ATM system, and assume that the slicing criterion consists of* `T4`*. Then the slice set* $\mathcal{L}$ *must also contain* `T1` *and* `T3` *(as these transitions provide definitions of the* `x` *which is used by* `T4`*). The result of slicing wrt.* $\mathcal{L}$ *is the EFSM depicted in Figure 4.*

The presence of ε-transitions may enable the further optimization of slices [2]. For example, in Figure 4, since `S3` is reachable by an ε-transition from `S2` and from `S4`, these three states may be merged. We shall not further address this optimization.

We use $\hat{E}_{\mathcal{L}}$ to denote the events relevant for the sliced EFSM:

$$\hat{E}_{\mathcal{L}} = \{e \in \hat{E} \mid \exists t \in \mathcal{L} : \mathsf{E}(t) = e \text{ or } \mathsf{E}(t) = e(v_b) \text{ for some } v_b\}.$$

For Example 1, we have $\hat{E}_{\mathcal{L}} = \{\texttt{enterCard}, \texttt{deposit}, \texttt{withdraw}\}$.

2.3 Semantics of EFSMs

We shall present a semantics that facilitates reasoning about slicing. Our development is relative to a given EFSM $(\hat{S}, \hat{T}, \hat{E}, \hat{V})$ and a fixed slice set $\mathcal{L}$. We shall use Example 1, where $\mathcal{L}$ contains `T1` and `T3` and `T4`, to illustrate the definitions.

A *configuration* C is a pair (n, s) where $n \in \hat{S}$ is a state and s is a store which maps variables $v \in \hat{V}$ to values. The domain of values is unspecified but we assume an expression language with $[\![A]\!]s$ denoting the value resulting from evaluating expression A in store s; similarly we assume a guard language with $[\![B]\!]s \in \{true, false\}$ denoting the value of the boolean expression B wrt. store s.

In our definitions, we shall use the subscript 1 to refer to the original EFSM, and subscript 2 to refer to the sliced EFSM. Thus $\mathsf{E}_1(t) = \mathsf{E}(t)$ and $\mathsf{G}_1(t) = \mathsf{G}(t)$, *etc.* If $t \in \mathcal{L}$ then $\mathsf{E}_2(t) = \mathsf{E}(t)$ and $\mathsf{G}_2(t) = \mathsf{G}(t)$, *etc.*, but otherwise (as then t is replaced by a ε-transition) we have $\mathsf{E}_2(t) = \varepsilon$ and $\mathsf{G}_2(t) = \texttt{true}$ and $\mathsf{D}_2(t) = \emptyset$ and $\mathsf{A}_2(t) = 0$. For example, in Example 1, we have $\mathsf{D}_1(\texttt{T2}) = \{\texttt{1}\}$ but $\mathsf{D}_2(\texttt{T2}) = \emptyset$.

An *event sequence* E is a sequence where each element is either of the form e where e is an unparameterized event in $\hat{E}$, or of the form $e(c)$ where e is a parameterized event in $\hat{E}$ and c is a value; the empty event sequence is denoted ε. To disregard events that are "irrelevant" for the sliced program, we use the (idempotent) function filter:

Definition 2 $\mathsf{filter}(E)$ *returns a subsequence of* E *which includes* e *or* $e(c)$ *iff* $e \in \hat{E}_{\mathcal{L}}$.

For Example 1, we have $\mathsf{filter}(\texttt{deposit(20)}, \texttt{done}) = \texttt{deposit(20)}$.

We are now ready to define the semantics. First a one-step move:

Definition 3 *We write*

$$i \vdash t : (n, s) \xrightarrow{E} (n', s')$$

to denote that (n, s) *in the* i*-semantics* ($i = 1, 2$) *through transition* t *moves to* (n', s') *while consuming the event sequence* E *(which will be either empty or a singleton). This happens when* t *is such that all the conditions listed below hold:*

- $\mathsf{S}(t) = n$ *and* $\mathsf{T}(t) = n'$
- *either* $\mathsf{E}_i(t) = \varepsilon = E$ *or there exists* $e \in \hat{E}$ *such that either* $\mathsf{E}_i(t) = e = E$ *or (for some* c *and* v_b*)* $\mathsf{E}_i(t) = e(v_b)$ *and* $E = e(c)$
- $[\![g]\!]s = true$
- *if* $\mathsf{D}_i(t) = \emptyset$ *then* $s' = s$ *but if* $\mathsf{D}_i(t)$ *is a singleton* $\{v\}$ *then* $s' = s[v \mapsto [\![A]\!]s]$

where g *and* A *are defined as follows: if* $E = e(c)$ *and* $\mathsf{E}_i(t) = e(v_b)$ *then* $g = \mathsf{G}_i(t)[c/v_b]$ *and* $A = \mathsf{A}_i(t)[c/v_b]$ *but otherwise* $g = \mathsf{G}_i(t)$ *and* $A = \mathsf{A}_i(t)$.

In Example 1, when $E = \texttt{deposit(20)}$ we have (for $i = 1, 2$ and for all stores s):

$$i \vdash \texttt{T3} : (\texttt{S3}, s) \xrightarrow{E} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20]).$$

A configuration is *stuck* (as is $(\texttt{S2}, s)$ in Example 1 if $s(\texttt{1})$ is non-empty) if no transitions apply:

Definition 4 *We say that (n, s) is i-stuck if for all t with $\mathsf{S}(t) = n$: $[\![\mathsf{G}_i(t)]\!]s = \mathit{false}$, or if $\mathsf{E}_i(t)$ is of the form $e(v_b)$: $[\![\mathsf{G}_i(t)[c/v_b]]\!]s = \mathit{false}$ for all c.*

As discussed in Section 1, we shall allow "idle" events: an EFSM will ignore an event for which there is no transition in the system, rather than block on it. This is formalized in the rule for multi-step moves:

Definition 5 *We write $i \vdash \pi : C \xrightarrow{E} C'$ iff with $\pi = [t_1..t_{k-1}]$ ($k \geq 1$) there exists $C_1, \ldots, C_k$ with $C = C_1$ and $C' = C_k$, and $E_1, \ldots, E_{k-1}$ where $E_1 \ldots E_{k-1}$ is a subsequence of E, such that $i \vdash t_j : C_j \xrightarrow{E_j} C_{j+1}$ for all $j \in 1 \ldots k-1$.*

If the moves are all non-idle, i.e., $E = E_1 \ldots E_{k-1}$, we write $i \vdash_{ni} \pi : C \xrightarrow{E} C'$.

Example 2 *In Example 1, with $E_1 =$* $\texttt{deposit(30)}, \texttt{done}, \texttt{withdraw(10)}$ *and with*
$E_2 =$ $\texttt{deposit(30)}, \texttt{withdraw(10)}$*, for all stores s we have*

$$1 \vdash_{ni} [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{E_1} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20])$$

$$2 \vdash_{ni} [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{E_2} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20])$$

but also, since E_2 is a subsequence of E_1:

$$2 \vdash [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{E_1} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20]).$$

Since the path taken to get from one configuration to another is invisible to the environment, but the event sequence is visible, we often only depict the latter:

Definition 6 *We write $i \vdash C \xrightarrow{E} C'$ iff $i \vdash \pi : C \xrightarrow{E} C'$ for some path π.*

We write $i \vdash_{ni} C \xrightarrow{E} C'$ iff $i \vdash_{ni} \pi : C \xrightarrow{E} C'$ for some path π.

We are often interested in moves that are unobservable (wrt. a given $\mathcal{L}$):

Definition 7 *We write $i \vdash C \xRightarrow{E} C'$ iff $i \vdash \pi : C \xrightarrow{E} C'$ for some π outside $\mathcal{L}$.*

We write $i \vdash_{ni} C \xRightarrow{E} C'$ iff $i \vdash_{ni} \pi : C \xrightarrow{E} C'$ for some π outside $\mathcal{L}$.

Even if $1 \vdash_{\mathrm{ni}} t \overset{(n,s)}{\Longrightarrow} E(n', s')$ it may happen that $s' \neq s$ and $E \neq \varepsilon$. But we have

Fact 8 *If $2 \vdash t : (n, s) \xrightarrow{E} (n', s')$ with $t \notin \mathcal{L}$ then $s' = s$ and $E = \varepsilon$.*

If $2 \vdash (n, s) \xRightarrow{E} (n', s')$ then $s' = s$, and if $2 \vdash_{ni} (n, s) \xRightarrow{E} (n', s')$ also $E = \varepsilon$.

Of special interest is moves where all but the last step are unobservable:

Definition 9 *We write $i \vdash^t_{ni} C \xRightarrow{E} C'$ if $t \in \mathcal{L}$ and there exists C_0, and E_0, E' with $E = E_0E'$, such that $i \vdash_{ni} C \xRightarrow{E_0} C_0$ and $i \vdash t : C_0 \xrightarrow{E'} C'$.*

Example 3 *In Example 1 we have (for all s with $s(\mathtt{l})$ the empty string)*

$$1 \vdash^{\mathtt{T3}}_{ni} (\mathtt{S2}, s) \stackrel{E_1}{\Rightarrow\!\!\rightarrow} (\mathtt{S4}, s[\mathtt{x} \mapsto s(\mathtt{x}) + 20, l \mapsto \mathtt{English}])$$

$$2 \vdash^{\mathtt{T3}}_{ni} (\mathtt{S2}, s) \stackrel{E_2}{\Rightarrow\!\!\rightarrow} (\mathtt{S4}, s[\mathtt{x} \mapsto s(\mathtt{x}) + 20])$$

where $E_1 = \mathtt{language}(\mathtt{English}), \mathtt{deposit}(20)$ *and* $E_2 = \mathtt{deposit}(20)$.

It will often be the case that a slice set is *uniform* in that it contains either all the transitions that mention a given event, or none of them:

Definition 10 (Uniformity) *We say that a slice set $\mathcal{L}$ is* uniform *iff the following property holds for all $e \in \hat{E}$: if $t_1, t_2 \in \hat{T}$ are such that $\mathsf{E}(t_1)$ equals e or is of the form $e(v_b)$, and $\mathsf{E}(t_2)$ equals e or is of the form $e(v_b)$, then either $t_1, t_2 \in \mathcal{L}$ or $t_1, t_2 \notin \mathcal{L}$.*

We shall see in the next section that a uniform slice set allows for stronger correctness properties, since we have:

Fact 11 *Assume that $\mathcal{L}$ is uniform. If $i \vdash_{ni} C \stackrel{E}{\Longrightarrow} C'$ then $\mathsf{filter}(E) = \varepsilon$.*

3 Slicing an EFSM: What does Correctness Mean?

In this section we shall develop definitions of what it means for slicing, with respect to a given slice set $\mathcal{L}$, to be semantically correct. We would like to capture two (yet rather informal) notions:

Completeness moves by the original EFSM can also be done by the sliced EFSM.

Soundness moves by the sliced EFSM can also be done by the original EFSM.

This suggests (assuming non-idle moves) that we aim for the following goals:

Completeness Attempt 1 *If $1 \vdash_{ni} C \stackrel{E}{\longrightarrow} C'$ then $2 \vdash_{ni} C \stackrel{E}{\longrightarrow} C'$.*

Soundness Attempt 1 *If $2 \vdash_{ni} C \stackrel{E}{\longrightarrow} C'$ then $1 \vdash_{ni} C \stackrel{E}{\longrightarrow} C'$.*

But these goals are too ambitious, for several reasons which will be detailed in the next subsections where we shall rewrite the goals so as to motivate the final versions, given in Section 3.3 (completeness) and Section 3.4 (soundness). We believe that the resulting goals capture what it should mean for slicing of EFSMs to be correct.

3.1 Event Sequences

Looking at Completeness Attempt 1 and Soundness Attempt 1, one may wonder that the *same* event sequence is used in the antecedent as in the consequent. And in fact, for soundness this demand is too strong, as we shall now show using Example 1. With $E_2 = \texttt{deposit}(30), \texttt{withdraw}(10)$, we have (cf. Example 2)

$$2 \vdash_{\text{ni}} [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{E_2} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20]).$$

Yet $1 \vdash_{\text{ni}} (\texttt{S3}, s) \xrightarrow{E_2} (\texttt{S4}, s')$ does *not* hold for any s'; in order for the original EFSM to execute E_2 we need to *pad* a `done` event: with $E_1 = \texttt{deposit}(30), \texttt{done}, \texttt{withdraw}(10)$, we have $\mathsf{filter}(E_1) = E_2$ and (cf.Example 2)

$$1 \vdash_{\text{ni}} [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{E_1} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20]).$$

This motivates the following modified goal:

Soundness Attempt 2 *If* $2 \vdash_{ni} C \xrightarrow{E_2} C'$ *then* E_2 *is a subsequence of an* E_1 *with* $1 \vdash_{ni} C \xrightarrow{E_1} C'$.

Moreover, if $\mathcal{L}$ *is uniform then* $\mathsf{filter}(E_1) = E_2$.

We shall now address completeness, and again look at Example 1 where with $E_1 = \texttt{deposit}(30), \texttt{done}, \texttt{withdraw}(10)$ we have (cf. Example 2)

$$1 \vdash_{\text{ni}} [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{E_1} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20])$$

but we clearly do not have

$$2 \vdash_{\text{ni}} [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{E_1} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20])$$

since $\texttt{done} \notin \hat{E}_{\mathcal{L}}$. On the other hand, `done` can be considered idle so (by Definition 5) we do have (cf. Example 2)

$$2 \vdash [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{E_1} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20])$$

and (with $\mathsf{filter}(E_1) = \texttt{deposit}(30), \texttt{withdraw}(10)$)

$$2 \vdash_{\text{ni}} [\texttt{T3}, \texttt{T5}, \texttt{T4}] : (\texttt{S3}, s) \xrightarrow{\mathsf{filter}(E_1)} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20]).$$

This motivates the following modified goal:

Completeness Attempt 2 *If* $1 \vdash_{ni} C \xrightarrow{E} C'$ *then* $2 \vdash C \xrightarrow{E} C'$.

Moreover, if $\mathcal{L}$ *is uniform then* $2 \vdash_{ni} C \xrightarrow{\mathsf{filter}(E)} C'$.

We shall now explain why we need to demand uniformity in the last parts of Completeness Attempt 2 and Soundness Attempt 2 . For that purpose, assume for a moment that Example 1 has been redesigned so as to replace the event `done`

by the event `deposit(y)` where `y` is ignored; thus the slice set $\mathcal{L} = \{\texttt{T1}, \texttt{T3}, \texttt{T4}\}$ is not uniform since `deposit` occurs both in `T3` and `T5`.

First observe that with $E = \texttt{deposit(10)}, \texttt{deposit(20)}$ we have

$$1 \vdash_{\mathrm{ni}} [\texttt{T5}, \texttt{T3}] : (\texttt{S4}, s) \xrightarrow{E} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20])$$

and certainly we also have

$$2 \vdash [\texttt{T5}, \texttt{T3}] : (\texttt{S4}, s) \xrightarrow{E} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20])$$

(but we also have say $2 \vdash_{\mathrm{ni}} [\texttt{T5}, \texttt{T3}, \texttt{T5}, \texttt{T3}] : (\texttt{S4}, s) \xrightarrow{E} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 30])$).

Without the requirement of $\mathcal{L}$ being uniform, Completeness Attempt 2 would require (as $\mathsf{filter}(E) = E$) that

$$2 \vdash_{\mathrm{ni}} (\texttt{S4}, s) \xrightarrow{E} (\texttt{S4}, s[\texttt{x} \mapsto s(\texttt{x}) + 20])$$

but this is clearly *not* possible as a non-idle execution of the sliced program on E will add 30 to `x`.

Next observe that we have

$$2 \vdash_{\mathrm{ni}} [\texttt{T5}] : (\texttt{S4}, s) \xrightarrow{\varepsilon} (\texttt{S3}, s)$$

and certainly we also have

$$1 \vdash_{\mathrm{ni}} [\texttt{T5}] : (\texttt{S4}, s) \xrightarrow{\texttt{deposit(20)}} (\texttt{S3}, s).$$

Without the requirement of $\mathcal{L}$ being uniform, Soundness Attempt 2 would require $\mathsf{filter}(\texttt{deposit(20)}) = \varepsilon$ which is clearly *not* the case.

3.2 Relevant Variables

Our tentative correctness demands are still too restrictive, as they require the sliced EFSM to produce *exactly* the same store as the original. But since slicing removes transitions that are not "relevant", the sliced EFSM is likely (as in Example 3) to disagree with the original EFSM on variables that are not *relevant*:

Definition 12 *For a transition t or a state n, we define its relevant variables as follows:*

1. *We say that v is relevant for t wrt. $\mathcal{L}$, written $v \in Rv_{\mathcal{L}}(t)$, iff there exists $t' \in \mathcal{L}$ such that $v \in \mathsf{U}(t')$, and there exists a path $[t_1..t_k]$ with $k \geq 1$ and $t = t_1$ and $t' = t_k$ such that for all $j \in 1 \ldots k-1$, $v \notin \mathsf{D}(t_j)$.*
2. *We say that v is relevant for a state n wrt. $\mathcal{L}$, written $v \in Rv_{\mathcal{L}}(n)$, iff there exists t with $\mathsf{S}(t) = n$ such that $v \in Rv_{\mathcal{L}}(t)$.*

We write $Rv(t)$ and $Rv(n)$), rather than $Rv_{\mathcal{L}}(t)$ and $Rv_{\mathcal{L}}(n)$, when $\mathcal{L}$ is clear from context.

In Example 1 (with $\mathcal{L} = \{\texttt{T1}, \texttt{T3}, \texttt{T4}\}$), `x` is relevant for `T3` and `T4` since `x` is used there, and also for `T2` and `T5`, but `x` is not relevant for `T1` as `x` is defined there but not used. Thus `x` is relevant for all states except `S1`.

Fact 13 *If $t \in \mathcal{L}$ then $\mathsf{U}(t) \subseteq Rv_{\mathcal{L}}(t)$.*

We have motivated the following attempt to phrase completeness:

Completeness Attempt 3 *If $1 \vdash_{ni} C \xrightarrow{E} C_1'$ then $2 \vdash C \xrightarrow{E} C_2'$ for some C_2' with $C_1' \; Q \; C_2'$.*

Moreover, if $\mathcal{L}$ is uniform then $2 \vdash_{ni} C \xrightarrow{\mathsf{filter}(E)} C_2'$.

Here we use Q to relate configurations that are in the same state and whose stores agree on the relevant variables of that state:

Definition 14 *$(n_1, s_1) \; Q \; (n_2, s_2)$ iff $n_1 = n_2$ and $s_1(v) = s_2(v)$ for all $v \in Rv(n_1)$.*

In order to make that result compose over a sequence of moves, we need to relax the assumptions and allow also the input stores to disagree on irrelevant variables:

Completeness Attempt 4 *If $1 \vdash_{ni} C_1 \xrightarrow{E} C_1'$ and $C_1 \; Q \; C_2$ then $2 \vdash C_2 \xrightarrow{E} C_2'$ for some C_2' with $C_1' \; Q \; C_2'$.*

Moreover, if $\mathcal{L}$ is uniform then $2 \vdash_{ni} C_2 \xrightarrow{\mathsf{filter}(E)} C_2'$.

Similarly, we can take another stab at soundness:

Soundness Attempt 3 *If $2 \vdash_{ni} C_2 \xrightarrow{E_2} C_2'$ and $C_1 \; Q \; C_2$ then E_2 is a subsequence of some E_1 such that $1 \vdash_{ni} C_1 \xrightarrow{E_1} C_1'$ for some C_1' with $C_1' \; Q \; C_2'$.*

Moreover, if $\mathcal{L}$ is uniform then $\mathsf{filter}(E_1) = E_2$.

3.3 Completeness

We are now ready to give our final version of completeness, where we can actually assume that the sliced EFSM goes through the same path of transitions as the original:

Desideratum 1 (Completeness) *If $1 \vdash_{ni} \pi : \; C_1 \xrightarrow{E} C_1'$ and $C_1 \; Q \; C_2$ then $2 \vdash \pi : \; C_2 \xrightarrow{E} C_2'$ for some C_2' with $C_1' \; Q \; C_2'$.*

Moreover, if $\mathcal{L}$ is uniform then $2 \vdash_{ni} \pi : \; C_2 \xrightarrow{\mathsf{filter}(E)} C_2'$.

3.4 Soundness

Before giving our final version of soundness, we must deal with one crucial issue: if the sliced EFSM does an ε-transition, with $2 \vdash t_0 : (n, s) \xrightarrow{\varepsilon} (n_1, s)$ where $t_0 \notin \mathcal{L}$, there may not exist (n_2, s_2) (and E) such that that $1 \vdash t_0 : (n, s) \xrightarrow{E} (n_2, s_2)$, since $[\![\mathsf{G}(t_0)]\!]s$ may be false.

In that case, the original EFSM may still eventually "catch up" with the sliced EFSM, by doing some unobservable loop that eventually results in a store s' such that $[\![\mathsf{G}(t_0)]\!]s'$ is true. Restricting our attention (unlike previous attempts) to moves with *exactly one* observable step, this suggests:

Soundness Attempt 4 *If* $2 \vdash^t_{ni} C_2 \stackrel{E_2}{\Rightarrow\!\!\rightarrow} C_2'$ *and* $C_1\ Q\ C_2$ *then* $1 \vdash^t_{ni} C_1 \stackrel{E_1E_2}{\Rightarrow\!\!\rightarrow} C_1'$ *for some* C_1', E_1 *with* $C_1'\ Q\ C_2'$. *Moreover, if* $\mathcal{L}$ *is uniform then* $\mathsf{filter}(E_1) = \varepsilon$.

But it may also happen that the original EFSM gets *stuck* or *loops* from (n, s), as we shall now illustrate by Example 1 where for all s and z we have

$$2 \vdash^{\mathtt{T3}}_{\mathrm{ni}} (\mathtt{S2}, s) \stackrel{\mathtt{deposit(z)}}{\Rightarrow\!\!\rightarrow} (\mathtt{S4}, s') \text{ with } s' = s[\mathtt{x} \mapsto \mathtt{x} + z].$$

If $s(\mathtt{l})$ is the empty string then for all w we do indeed have for some s' with $s'(\mathtt{x}) = s(\mathtt{x}) + z$:

$$1 \vdash^{\mathtt{T3}}_{\mathrm{ni}} (\mathtt{S2}, s) \stackrel{\mathtt{language(w)},\mathtt{deposit(z)}}{\Rightarrow\!\!\rightarrow} (\mathtt{S4}, s').$$

But if $s(\mathtt{l}) \neq$ "" then the original EFSM gets stuck at (S2,s), and by extension from (S1,s), before reaching even the source of T3.

Definition 15 *[Stuck from] We say that the original EFSM gets stuck from* (n, s)*, avoiding the slice set* $\mathcal{L}$*, iff there exists* (n', s') *which is 1-stuck (cf. Definition 4) such that for some* π *and* E *we have* $1 \vdash \pi : (n, s) \xrightarrow{E} (n', s')$ *where neither* n'*, nor the source of a transition in* π*, is also the source of a transition in* $\mathcal{L}$ *(thus* π *is outside* $\mathcal{L}$*).*

Now imagine that at S2 we add a self-looping transition with guard $\mathtt{l} =$ "English", and look at what happens to a configuration $C_2 = (\mathtt{S2}, s)$ with $s(\mathtt{l}) =$ "English". If the action of the added transition is $\mathtt{l} :=$ "", then the original EFSM will indeed eventually catch up on the sliced EFSM on C_2, but if the action of the added transition is `skip`, then the original EFSM will do an infinite loop and never reach even the source of T3 — in which case we say that the original EFSM loops from C_2, avoiding {T3}, as formalized by:

Definition 16 *[Loop from] We say that the original EFSM loops from* (n, s)*, avoiding the slice set* $\mathcal{L}$*, if for all* $j \geq 1$ *there exists* t_j *whose source is not the source of a transition in* $\mathcal{L}$ *such that for all* $k \geq 1$*, for some* E_k, C_k*:* $1 \vdash [t_1..t_k] : C \xrightarrow{E_k} C_k$.

In Figure 3, the original EFSM loops from (S3,s) avoiding T1 and T2, but not avoiding T3 and not avoiding T5.

We have motivated a final version of soundness, called *weak soundness*:

Desideratum 2 (Weak Soundness) *If* $2 \vdash^t_{ni} C_2 \stackrel{E_2}{\Rightarrow\!\!\rightarrow} C_2'$ *and* $C_1\ Q\ C_2$ *then either*

1. $1 \vdash^t_{ni} C_1 \stackrel{E_1 E_2}{\Rightarrow\!\!\rightarrow} C_1'$ *for some* C_1', E_1 *with* $C_1'\ Q\ C_2'$
   (and if $\mathcal{L}$ *is uniform then* $\mathsf{filter}(E_1) = \varepsilon$*), or*
2. *the original EFSM gets stuck from* C_1 *avoiding* $\mathcal{L}$*, or*
3. *the original EFSM loops from* C_1 *avoiding* $\mathcal{L}$*.*

While this requirement allows for several kinds of behavior (hence the term "weak"), keep in mind what it rules out:

- that all moves from C_1 will eventually result in an observable step, but
- each such move has a transition other than t as its first observable step.

In some situations, it may not be desirable to slice away loops, for example if we want to ensure that certain temporal properties are preserved. If this is the case, we shall go for *strong* soundness:

Desideratum 3 (Strong Soundness) *If* $2 \vdash^t_{ni} C_2 \stackrel{E_2}{\Rightarrow\!\!\rightarrow} C_2'$ *and* $C_1\ Q\ C_2$ *then either*

1. $1 \vdash^t_{ni} C_1 \stackrel{E_1 E_2}{\Rightarrow\!\!\rightarrow} C_1'$ *for some* C_1', E_1 *with* $C_1'\ Q\ C_2'$
   (and if $\mathcal{L}$ *is uniform then* $\mathsf{filter}(E_1) = \varepsilon$*), or*
2. *the original EFSM gets stuck from* C_1 *avoiding* $\mathcal{L}$*.*

It may seem a natural final step to propose a "very strong soundness" requirement that would not allow the original program to get stuck. To ensure that, it appears we would need the slice set to contain all transitions with guards that are not always true. While this may be of interest in some applications, it may be considered orthogonal to our development which shows how Weak/Strong Soundness follows from general principles (first discovered by [9]).

Even though the requirements for Weak Soundness and Strong Soundness mention only moves with exactly one observable steps, they can in the natural way be extended to requirements about sequences of such moves (as the postcondition coincides with the precondition with both using Q).

4 Slicing an EFSM: How to get Correctness?

We shall develop conditions on the slice set $\mathcal{L}$ that ensure completeness and/or strong/weak soundness. In Section 4.1 we introduce the standard notion of data dependence which suffices for completeness. For soundness, in addition to data dependence we need two crucial conditions: in Section 4.3 we introduce the condition "weak commitment closed" ($\mathcal{WCC}$) which suffices for weak soundness, and in Section 4.4 we introduce the condition "strong commitment closed" ($\mathcal{SCC}$) which ensures strong soundness; both are expressed using the notion of "next observable" introduced in Section 4.2.

4.1 Data Dependence

Our definition is standard except that it relates transitions rather than states:

Definition 17 (Data Dependence) *We say that t' is data dependent on t, written $t \rightarrow_{\mathrm{dd}} t'$, iff there exists a variable $v \in \mathsf{D}(t) \cap \mathsf{U}(t')$ and a path $[t_1..t_k]$ ($k \geq 0$) from $\mathsf{T}(t)$ to $\mathsf{S}(t')$ such that for all $j \in 1 \ldots k$, $v \notin \mathsf{D}(t_j)$.*

For example, in Figure 3, `T4` is data dependent on `T1` and `T3`, and also on itself which is uninteresting for the purpose of computing a slice set that is closed under $\rightarrow_{\mathrm{dd}}$:

Definition 18 *Say $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}}$ iff $t \in \mathcal{L}$ whenever $t \rightarrow_{\mathrm{dd}} t'$ and $t' \in \mathcal{L}$.*

Being closed under data dependence is sufficient for completeness (cf. Desideratum 1):

Theorem 1 (Completeness) *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}}$. If*

- $1 \vdash_{ni} \pi : C_1 \xrightarrow{E} C_1'$ *and*
- $C_1 \; Q \; C_2$

then there exists C_2' such that

- $2 \vdash \pi : C_2 \xrightarrow{E} C_2'$ *and*
- $C_1' \; Q \; C_2'$.

Moreover, if $\mathcal{L}$ is uniform then $2 \vdash_{ni} \pi : C_2 \xrightarrow{\mathsf{filter}(E)} C_2'$.

The proof is in Appendix A.

4.2 Next Observable

Following recent trends in the theoretical foundation of slicing [1, 9] we shall employ the concept of "next observable" which allows us to express the classical notion of "control dependence" in a way that doesn't require the existence of a unique exit state. In our setting, where a slice set consists of transitions rather than states, we need to phrase "next observable" in a somewhat different way:

Definition 19 *For a slice set $\mathcal{L}$, for each state n we define $obs_{\mathcal{L}}(n)$ (written $obs(n)$ when $\mathcal{L}$ is given by the context) as the set of states n' such that*

- *there exists $t \in \mathcal{L}$ with $\mathsf{S}(t) = n'$, and*
- *there exists a path outside $\mathcal{L}$ from n to n'.*

For example, consider the EFSM in Figure 2, and assume that $\mathcal{L} = \{\mathtt{t5}, \mathtt{t6}\}$. Then for all n we have $obs_{\mathcal{L}}(n) \subseteq \{\mathsf{S}(\mathtt{t5}), \mathsf{S}(\mathtt{t6})\} = \{\mathtt{S2}, \mathtt{S3}\}$. In particular, $obs_{\mathcal{L}}(\mathtt{S1}) = \{\mathtt{S2}, \mathtt{S3}\}$ whereas $obs_{\mathcal{L}}(\mathtt{S2}) = \{\mathtt{S2}\}$ (since there is no path outside $\mathcal{L}$ from `S2` to `S3`) but $obs_{\mathcal{L}}(\mathtt{S3}) = \{\mathtt{S2}, \mathtt{S3}\}$ (since `[t4,t1]` is a path outside $\mathcal{L}$ from `S3` to `S2`).

4.3 Weak Commitment Closure

In order to obtain (weak) soundness, we in general cannot allow a set $obs_{\mathcal{L}}(n)$ to contain *two* (or more) states. To see this, consider Figure 2 with $\mathcal{L} = \{\texttt{t5}, \texttt{t6}\}$ so that $obs_{\mathcal{L}}(\texttt{S1}) = \{\texttt{S2}, \texttt{S3}\}$. Then the sliced EFSM may move to S3 (through ε-transitions) and perform the observable transition t6, while such a move may be impossible for the original EFSM (for example if G(t3) is false) which could instead move to S2 (if G(t1) is true) and perform t5. This motivates the notion of "weak commitment closed" ($\mathcal{WCC}$):

Definition 20 (WCC) *We say that $\mathcal{L}$ satisfies $\mathcal{WCC}$ iff for each state n, $obs_{\mathcal{L}}(n)$ is either empty or a singleton.*

We see that in Figure 2, the set $\{\texttt{t5}, \texttt{t6}\}$ does *not* satisfy $\mathcal{WCC}$.

Satisfying $\mathcal{WCC}$ (and being closed under data dependence) is sufficient for weak soundness (cf. Desideratum 2):

Theorem 2 (Weak Soundness) *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}}$ and satisfies $\mathcal{WCC}$.*

If $2 \vdash^t_{ni} C_2 \stackrel{E_2}{\Rightarrow\!\!\rightarrow} C_2'$ and $C_1\ Q\ C_2$ then there are 3 possibilities:

1. *$1 \vdash^t_{ni} C_1 \stackrel{E_1 E_2}{\Rightarrow\!\!\rightarrow} C_1'$ for some C_1', E_1 with $C_1'\ Q\ C_2'$*
   (and if $\mathcal{L}$ is uniform then $\mathsf{filter}(E_1) = \varepsilon$), or
2. *the original EFSM gets stuck from C_1 avoiding $\mathcal{L}$ (cf. Definition 15), or*
3. *the original EFSM loops from C_1 avoiding $\mathcal{L}$ (cf. Definition 16).*

The proof is in Appendix A.

4.4 Strong Commitment Closure

In order to obtain strong soundness, we cannot allow a state n to be part of an infinite path that avoids $obs_{\mathcal{L}}(n)$. To see this, consider Figure 2 but this time with $\mathcal{L} = \{\texttt{t9}\}$. This trivially satisfies $\mathcal{WCC}$ as $obs_{\mathcal{L}}(n)$ will always be either $\emptyset$ or $\{\texttt{S5}\}$, in particular $obs_{\mathcal{L}}(\texttt{S1}) = \{\texttt{S5}\}$ but there is an infinite path from S1 that avoids S5. Thus the sliced EFSM may move to S5 and perform the observable transition t9, while such a move may be impossible for the original EFSM (if say G(t2) and G(t3) are both false) which could instead (for certain values of the store) cycle infinitely between S1, S2, and S3. This would violate strong soundness, and motivates the notion of "strong commitment closed" ($\mathcal{SCC}$):

Definition 21 (SCC) *We say that $\mathcal{L}$ satisfies $\mathcal{SCC}$ iff for each state n, either*

- *$obs_{\mathcal{L}}(n)$ is empty, or*
- *$obs_{\mathcal{L}}(n)$ is a singleton n', and there is no infinite path from n that avoids n'.*

Clearly a slice set that satisfies $\mathcal{SCC}$ will also satisfy $\mathcal{WCC}$.

Satisfying $\mathcal{SCC}$ (and being closed under data dependence) is sufficient for strong soundness (cf. Desideratum 3):

Theorem 3 (Strong Soundness) *Assume $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}}$ and satisfies $\mathcal{SCC}$.*

If $2 \vdash^t_{ni} C_2 \stackrel{E_2}{\Rightarrow\!\!\rightarrow} C_2'$ and $C_1\ Q\ C_2$ then there are 2 possibilities:

1. *$1 \vdash^t_{ni} C_1 \stackrel{E_1 E_2}{\Rightarrow\!\!\rightarrow} C_1'$ for some C_1', E_1 with $C_1'\ Q\ C_2'$ (and if $\mathcal{L}$ is uniform then $\mathsf{filter}(E_1) = \varepsilon$), or*
2. *the original EFSM gets stuck from C_1 avoiding $\mathcal{L}$.*

The proof is in Appendix A.

5 Computing Least Slices

For a given EFSM, there may be many slice sets that satisfy $\mathcal{WCC}$ and are closed under data dependence.

Example 4 *Consider the EFSM given in Figure 5. If the slicing criterion is transition $t8$, and $t8$ is data dependent on $t3$ and $t6$ (but no other data dependences exist), then a superset of $\{t3, t6, t8\}$ is closed under data dependence, and will satisfy $\mathcal{WCC}$ iff it contains $t2$ and $t5$. (That is, any of $t1$, $t4$ or $t7$ may or may not be there, so there are 8 possible supersets.)*

However, among all the sets that satisfy $\mathcal{WCC}$ and are closed under data dependence, there will always be a *least* one (that is, one which is a subset of all other such sets). This follows since in Section 5.1 we shall present an algorithm that we can prove always returns that least set. In the above example, this will be $\{t2, t3, t5, t6, t8\}$ which is constructed as follows: we first close $\{t8\}$ under data dependence which yields $\{t3, t6, t8\}$ (in bold in Figure 5), so that the observable states (the sources of these transitions) are $\{S3, S5, S7\}$ (filled in Figure5); we then do a backwards search from these states and see that $S2$ has *two* next observables ($S3$ and $S5$) so we need to add $t2$ and $t5$, after which no more transitions need to be added. Note that even though also $S1$ has two next observables, it is important that $S2$ is considered first, as otherwise $t1$ is needlessly added.

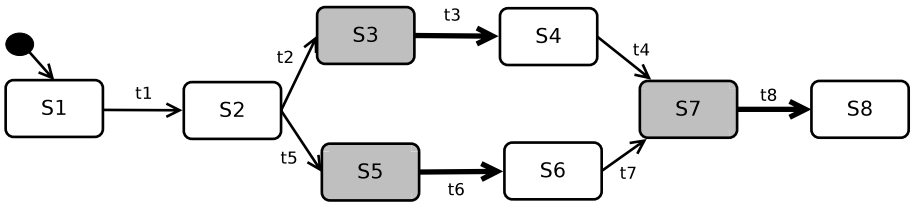

Fig. 5. An EFSM where the current slice set $\{t3, t6, t8\}$ (in bold) does not satisfy $\mathcal{WCC}$, as the corresponding set of observable states $\{S3, S5, S7\}$ (filled) allows some states ($S1$ and $S2$) to have *two* next observables.

Similarly, for a given EFSM there may be many slice sets that satisfy $\mathcal{SCC}$ and are closed under data dependence.

Example 5 *Consider the EFSM given in Figure 6. If the slicing criterion is transition $t4$ (and there are no data dependences), then the following supersets satisfy $\mathcal{SCC}$: $\{t3, t4\}$, $\{t1, t3, t4\}$, $\{t2, t3, t4\}$, and $\{t1, t2, t3, t4\}$.*

In Section 5.2, we shall present an algorithm that always returns the *least* such set. In the above example, this will be $\{t3, t4\}$ which is constructed as follows: with the initial observable state being $S3$ (filled in Figure 6), we again do a backwards search from $S3$ and see that $S2$ can avoid $S3$ and hence we need to add $t3$, after which no more transitions need to be added. Note that even though also $S1$ can avoid $S3$, it is important that $S2$ is considered first, as otherwise $t1$ is needlessly added.

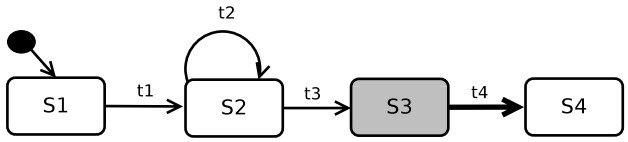

Fig. 6. An EFSM where the current slice set $\{t4\}$ (in bold) does not satisfy $\mathcal{SCC}$, as the corresponding set of observable states $\{S3\}$ (filled) allows some states ($S1$ and $S2$) to *avoid* that observable.

When analyzing the complexity of our algorithms, we assume that sets of states and sets of transitions are represented as bitmaps, allowing one element to be added in constant time. To prepare for our algorithms, we now address how to ensure data dependence; we shall assume a pre-computed table `DD` such that `DD(`t`,`u`)` is true iff $t \rightarrow_{\mathrm{dd}} u$ holds.

Lemma 22 *The table* `DD` *can be computed in time $O(a^2)$ where a is the number of transitions in the EFSM.*

To do so, for each transition t where $\mathsf{D}(t)$ is non-empty and thus a singleton v, we first mark all nodes except $\mathsf{T}(t)$ as unvisited and then do a depth-first search from $\mathsf{T}(t)$ to find how far this definition "propagates"; the procedure `Reaches` finds the transitions u that use v and which can be reached through transitions that do not redefine v.

We can use the table `DD` to add transitions while preserving the property of being closed under data dependence:

Lemma 23 *Given a table* `DD` *such that* `DD`(t, u) *holds iff $t \rightarrow_{\mathrm{dd}} u$, we can write a function* `DDclose` *that given a slice set L closed under data dependence, and another slice set L_2 with $L \cap L_2 = \emptyset$, returns the least slice set L' that contains $L \cup L_2$ and is closed under data dependence. The running time of that function is in $O(a \cdot (|L'| - |L|))$ where a is the number of transitions in the EFSM.*

Procedure Reaches(n,v)

```
;  /* Finds the transitions that use v and are reachable from n through
   an acyclic path that does not redefine v. */
1  foreach u with S(u) = n do
2      if v ∈ U(u) then
3          output u
4      if v ∉ D(u) and T(u) has not yet been visited then
5          mark T(u) as visited
6          Reaches(T(u), v)
```

The function `DDclose` is defined below. It works by maintaining a queue Q of transitions that have been added, but not yet examined to include any transitions they data depend on. Eventually, all |L'| - |L| new transitions will have be examined, with each examination taking time in $O(a)$.

Procedure DDclose(L,$L2$)

```
1  Q := L2
2  L := L ∪ L2
3  while Q ≠ ∅ do
4      remove an element, t2, from Q
5      foreach t1 with (t1, t2) ∈ DD do
6          if t1 ∉ L then
7              add t1 to Q and to L
8  return L
```

5.1 Computing Least $\mathcal{WCC}$-Satisfying Slices

We present an algorithm (Algorithm 1) that for a given slice set `L` returns the least superset of `L` that satisfies $\mathcal{WCC}$ and is closed under data dependence. The algorithm works by adding transitions to `L` until no new transitions need to be added. In each iteration, the algorithm computes in `B` the states that are sources of transitions in `L`, and does from `B` a backwards breadth-first search through transitions not in `L`, with `V` being the states that have been visited so far. For each $n \in$ `V`, the array entry $X[n]$ is defined as the state in `B` that can be reached (through transitions not in `L`) from n; that state will be a next observable of n. The current frontier of the search is called `C`, the exploration of which builds up `Cnew`, the next frontier. If there is a transition $t \notin$ `L` to a node $m \in C$, from a node n that already belongs to `V` but with $X[n] \neq X[m]$, we detect that n has *two* next observables and we infer that t has to be included in the slice set (as motivated in Example 4 and justified by the correctness result).

ALGORITHM 1: Computes the least $\mathcal{WCC}$-closed slice.

Input: An EFSM M; a set $\mathcal{L}$ of transitions in M
Output: the least $\rightarrow_{\mathrm{dd}}$-closed superset of $\mathcal{L}$ that satisfies $\mathcal{WCC}$

```
1  L := DDclose(∅,ℒ)
2  repeat
          /* L is closed under →dd, and is a subset of any →dd-closed
     superset of ℒ that satisfies 𝒲𝒞𝒞 */
3     Lnew := ∅
4     B := {n | ∃t ∈ L : n = S(t)}
5     foreach n ∈ B do
6        X[n] := n
7     V, C := B
8     while C ≠ ∅ and Lnew = ∅ do
9        Cnew := ∅
10       foreach m ∈ C do
11          foreach transition t ∉ L with T(t) = m do
12             n := S(t)
13             if n ∈ V then
14                if X[n] ≠ X[m] then
15                   Lnew := Lnew ∪ {t}
16             else
17                V := V ∪ {n}
18                Cnew := Cnew ∪ {n}
19                X[n] := X[m]
20       C := Cnew
21    L := DDclose(L,Lnew)
22 until Lnew = ∅
```

Example 6 *Let us apply Algorithm 1 to the EFSM in Figure 2, to find the least set that satisfies $\mathcal{WCC}$ and contains $t6$ and $t9$.*

In the first iteration, $B = \{S3, S5\}$. Since $S3$ can be reached from $S4$ by $t7$, and $S5$ can be reached from $S4$ by $t8$, there is a conflict at $S4$ so we add $t7$ (or $t8$) to L.

In subsequent iterations, with $B = \{S3, S4, S5\}$, we add $t8$ (due to the conflict at $S4$), as well as $t2$ and $t3$ (due to a conflict at $S1$).

We now have $B = \{S1, S3, S4, S5\}$, and may add $t4$ due to a conflict at $S3$. The next iteration adds $t1$ since from $S1$ one can reach $S3$ through the path $[t1, t5]$. The following iteration adds $t10$ since from $S4$ one can reach $S3$ through the path $[t10, t12, t5]$.

We are left with $\mathcal{L} = \{t1, t2, t3, t4, t6, t7, t8, t9, t10\}$, illustrated in Figure 7 at which point no new transitions can be added, since $\mathsf{S}(t5)$ does not have other observables than $S3$. And $\mathcal{L}$ does indeed satisfy $\mathcal{WCC}$: for all states n we have $obs_{\mathcal{L}}(n) = \{n\}$, except $obs_{\mathcal{L}}(S2) = obs_{\mathcal{L}}(S6) = \{S3\}$.

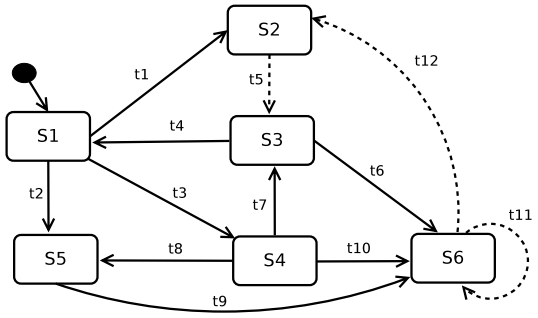

Fig. 7. Applying Algorithm 1 for computing the least WCC-closed set containing $t6$ and $t9$ to our running example (Figure 2). Transitions in $\mathcal{L}$ are shown using solid lines, and those not with dotted lines.

Theorem 4 (WCC Algorithm Complexity) *Algorithm 1 (including the construction of the table* `DD`*) can be implemented to run in time* $O(a^2)$ *where* a *is the number of transitions.*

Proof. By Lemma 22, the table `DD` can be computed in time $O(a^2)$. By Lemma 23, the total time of the calls to `DDclose` is in $O(a^2)$.

The outer loop eventually terminates; it continues only when `Lnew` $\neq \emptyset$ and as `Lnew` is disjoint from `L` by construction, the assignment `L := DDclose(L, Lnew)` will strictly increase the size of `L` but this cannot go on forever as the number of transitions is finite. In particular, we see that the outer loop iterates $O(a)$ times.

It is thus sufficient to show that each iteration of the outer loop, except for the call to `DDclose`, runs in time $O(a)$.

Towards this goal, first observe that the `while` loop will always terminate, as eventually no more states can be added to `V` in which case `C` will be empty and the loop thus exit (if it hasn't done already). In particular, each node is processed at most once by the outer **foreach** loop, and each transition is processed at most once by the inner **foreach** loop. As the body of the inner **foreach** loop executes in constant time, this yields the claim.

In Appendix B, we prove:

Theorem 5 (WCC Algorithm Correctness) *Assuming that the table* `DD` *correctly computes data dependence, Algorithm 1 returns in* `L` *the least superset of* $\mathcal{L}$ *that is closed under data dependence and satisfies* $\mathcal{WCC}$*.*

5.2 Computing Least $\mathcal{SCC}$-Closed Slices

We present Algorithm 2 that extends Algorithm 1 to compute the least superset of $\mathcal{L}$ that satisfies $\mathcal{SCC}$ (and is closed under data dependence): as we explore

each state n from which a state m' in `B` is reachable, we check if there from n is an infinite path that avoids m'; if that is the case, we can add to `L` the first transition in the path from n to m'.

Algorithm 2 is identical to Algorithm 1 except that two lines (13 and 14) have been added. The algorithm employs a pre-computed table `LoopAvoids` such that `LoopAvoids(`n`,`m`)` is true iff there is an infinite path from n that avoids m.

ALGORITHM 2: Computes the least $\mathcal{SCC}$-closed slice

Input: An EFSM M; a set $\mathcal{L}$ of transitions in M
Output: the least $\rightarrow_{\mathrm{dd}}$-closed superset of $\mathcal{L}$ that satisfies $\mathcal{SCC}$

```
1  L := DDclose(∅,ℒ)
2  repeat        /* L is closed under →dd, and a subset of any →dd-closed
     superset of ℒ that satisfies 𝒮𝒞𝒞 */
3     Lnew := ∅
4     B := {n | ∃t ∈ L : n = S(t)}
5     foreach n ∈ B do
6        X[n] := n
7     V, C := B
8     while C ≠ ∅ and Lnew = ∅ do
9        Cnew := ∅
10       foreach m ∈ C do
11          foreach transition t ∉ L with T(t) = m do
12             n := S(t)
13             if LoopAvoids(n,X[m]) then          /* See Algorithm 3 */
14                Lnew := Lnew ∪ {t}
15             if n ∈ V then
16                if X[n] ≠ X[m] then
17                   Lnew := Lnew ∪ {t}
18             else
19                V := V ∪ {n}
20                Cnew := Cnew ∪ {n}
21                X[n] := X[m]
22       C := Cnew
23    L := DDclose(L,Lnew)
24 until Lnew = ∅
```

The table `LoopAvoids` can be constructed by Algorithm 3. For each m, the transitions involving m are ignored, and a depth-first search is made; then `LoopAvoids(`n`,`m`)` is set to true iff a back edge is reachable from n.

Fact 24 *For all n, m,* `LoopAvoids(`n`,`m`)` *is true iff there is an infinite path from n that avoids m.*

Example 7 *Let us apply Algorithm 2 to the EFSM in Figure 2, to find the least set that satisfies $\mathcal{SCC}$ and contains $t9$.*

ALGORITHM 3: Constructing the table `LoopAvoids`, with procedure DFS.

```
1  foreach state m do
2      foreach state n do
3          LoopAvoids(n,m) := false
4      T_m := the transitions that do not involve m
5      foreach state n except m do
6          color[n] := white
7      while (exists n: color[n] = white) do
8          let n be a state with color[n] = white
9          call DFS(n);

10 Procedure DFS(n)
11     color[n] := gray
12     foreach t ∈ T_m with S(t) = n do
13         n' := T(t)
14         if color[n'] = gray then
15             LoopAvoids(n,m) := true
16         else if color[n'] = black then
17             if LoopAvoids(n',m) then
18                 LoopAvoids(n,m) := true
19         else if color[n'] = white then
20             DFS(n')
21             if LoopAvoids(n',m) then
22                 LoopAvoids(n,m) := true
23     color[n] := black
```

In the first iteration, $B = \{S5\}$. *Since* $t2$ *has source* $S1$ *and target* $S5$, *and there is an infinite path from* $S1$ *that avoids* $S5$, *we add* $t2$. *Similarly, since* $t8$ *has source* $S4$ *and target* $S5$, *and there is an infinite path from* $S4$ *that avoids* $S5$, *we add* $t8$.

Now, $B = \{S1, S4, S5\}$. *Since* $S4$ *can be reached from* $S1$ *by* $t3$, *we add* $t3$. *Since* $t4$ *has target* $S1$ *and there is an infinite path from* $S3$ *that avoids* $S1$, *we add* $t4$.

Now, $B = \{S1, S3, S4, S5\}$. *Since* $S3$ *can be reached from* $S4$ *by* $t7$, *we add* $t7$. *Since* $S3$ *can be reached from* $S1$ *by the path* $[t1, t5]$, *we add* $t1$. *Since* $S3$ *can be reached from* $S6$ *by the path* $[t12, t5]$, *and there is an infinite path from* $S6$ *(via* $t11$*) that avoids* $S3$, *we add* $t12$.

Now, $B = \{S1, S3, S4, S5, S6\}$. *Since* $S6$ *can be reached from* $S3$ *by* $t6$, *and from* $S4$ *by* $t10$, *we add* $t6$ *and* $t10$.

We are left with $\mathcal{L} = \{t1, t2, t3, t4, t6, t7, t8, t9, t10, t12\}$ *at which point no new transitions can be added. And* $\mathcal{L}$ *does indeed satisfy* $\mathcal{SCC}$: $obs_{\mathcal{L}}(S1) = \{S1\}$; $obs_{\mathcal{L}}(S2) = \{S3\}$; $obs_{\mathcal{L}}(S3) = \{S3\}$; $obs_{\mathcal{L}}(S4) = \{S4\}$; $obs_{\mathcal{L}}(S5) = \{S5\}$; $obs_{\mathcal{L}}(S6) = \{S6\}$ *but no infinite path from* $S2$ *avoids* $S3$.

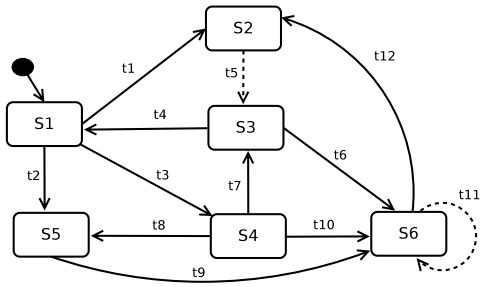

Fig. 8. Applying Algorithm 2 to our running example (Figure 2) for computing the least set that satisfies SCC that contains $t9$. Transitions in $\mathcal{L}$ are shown using solid lines, and those not in $\mathcal{L}$ with dotted lines.

Theorem 6 (SCC Algorithm Complexity) *Algorithm 2, including the construction of the tables* `DD` *and* `LoopAvoids`*, can be implemented to run in time* $O(a^2)$ *where* a *is the number of transitions.*

Proof. To construct the table `LoopAvoids`, Algorithm 3 will for each m use time in $O(a)$, and hence (since $|\hat{S}| \in O(a)$) have a total running time in $O(a^2)$. Next Algorithm 2 can use that table, and will also run in time $O(a^2)$ (by an analysis almost identical to the one given in the proof of Theorem 4 for Algorithm 1).

In Appendix B, we prove:

Theorem 7 (SCC Algorithm Correctness) *Assuming that the table* `DD` *correctly computes data dependence, Algorithm 2 (using Algorithm 3) returns in* `L` *the least superset of* $\mathcal{L}$ *that is closed under data dependence and satisfies* $\mathcal{SCC}$*.*

6 Comparing Control Dependence Definitions for EFSMs

There have been several previous attempts to define suitable notions of control dependence for EFSMs. These approaches, none of which have allowed a proof that the resulting slices are in some sense "correct", will in this section be described, and illustrated on various examples while comparing to $\mathcal{WCC}/\mathcal{SCC}$-based slicing.

6.1 Preliminary Definitions

Definition 25 (Post-Dominance [20]) *Given a machine with exactly one exit state (reachable from all other states), and two states* Y *and* Z*, we say that* Z *post-dominates* Y *iff* Z *is in every path from* Y *to the exit state.*

Definition 26 (Maximal Path) *A path in an EFSM is maximal iff it terminates in an exit state, or is infinite.*

Definition 27 (Control Sink) *A control sink in an EFSM is a set of transitions $\mathcal{K}$ that forms a strongly connected component such that, for each transition t in $\mathcal{K}$, each successor of t is also in $\mathcal{K}$.*

Definition 28 (Sink-bounded Paths) *We say that a path π joins a control sink $\mathcal{K}$ if π is maximal and $\mathcal{K} \cap \pi \neq \emptyset$.*

If π joins a control sink $\mathcal{K}$, we say that π is unbiased if π is finite or if all transitions in $\mathcal{K}$ occur infinitely often in π.

We say that a path is sink-bounded if it joins a control sink and is unbiased.

With some abuse of notation (for historical reasons), we shall say that π is an unfair sink-bounded path iff π joins a control sink (even if π is unbiased).

6.2 Definitions For EFSMs With Exactly One Exit State

Korel et al. [20] present a definition of control dependence for EFSMs that

- is insensitive to non-termination
- is phrased in terms of post-dominance, similar to what is standard for static backward slicing of programs [33, 25]
- is thus not applicable if there are multiple exit states, or none (in which case the machine is likely designed to be potentially non-terminating).

Definition 29 (Insensitive Control Dependence (ICD) [20]) *Transition T_k is control dependent on transition T_i if:*

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

6.3 Definitions For EFSMs With Arbitrary Number of Exit States

Ranganath et al. [26] were the first to propose notions of control dependence, one that is sensitive to non-termination and one which is not, that apply to programs with multiple or none exit points. Androutsopoulos et al. [4] adapted these definitions for EFSMs, yielding what is known as Non-Termination Sensitive Control Dependence (NTSCD) and Non-Termination Insensitive Control Dependence (NTICD), and also introduced Unfair Non-Termination Insensitive Control Dependence (UNTICD) that overcame a limitation of NTICD.

To express those definition, we first present a general definition, parametrized on the set *PATH* of paths we consider:

Definition 30 (Control Dependence (CD)) *A transition T_j is control dependent on a transition T_i, written $T_i \xrightarrow{CD} T_j$, iff:*

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

We may then instantiate as follows: with PATH the set of

– maximal paths, we get *NTSCD*;
– sink-bounded paths, we get *NTICD*;
– unfair sink-bounded paths, we get *UNTICD*.

NTICD cannot compute control dependences within control sinks. UNTICD overcomes this limitation by considering also paths that are not unbiased, allowing for extra control dependences within control sinks to be introduced; these dependences, however, are the same as for NTSCD.

Control dependence definitions for EFSMs are classified, in Table 1, according to whether they are "weak", that is not sensitive to non-termination (thus aiming only at something like "weak soundness"), or "strong", that is sensitive to non-termination (thus aiming at something like "strong soundness").

Weak Control Dependence	ICD [20], NTICD [4], UNTICD [4], WCC
Strong Control Dependence	NTSCD [4], SCC

Table 1. Control Dependence definitions for EFSMs

6.4 Examples of Slicing Wrt Various Definitions

Let us first consider the EFSM in Figure 9 where there are no data dependencies. Here S5 is the unique exit state. Apart from `start`, all other states form a strongly connected group, and each have a transition (invoked in case of "error") to S5; moreover, S2, S3 and S4 have self-transitions.

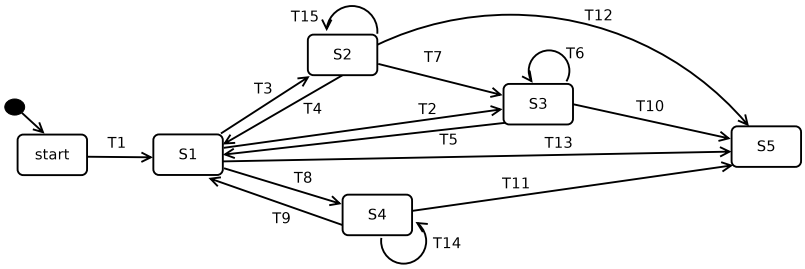

Fig. 9. An EFSM where NTSCD and SCC produce slices of the same size.

Consider the result of slicing the EFSM in Figure 9 wrt. criterion T2. WCC will add no other transitions (as only S1 can be a next observable), whereas SCC will add T4, T5, T9 (since from S2,S3,S4 there are infinite paths that avoid S1) and then (as now all of S1,S2,S3,S4 may be next observables) also T3, T7, T8. SCC thus yields the slice {T2, T3, T4, T5, T7, T8, T9}. Also NTSCD, as

well as NTICD and UNTICD (note that a path is sink-bounded iff it ends in S5), will produce that slice, since T2 depends on T4 and on T5 and on T9, and T4/T5/T9 depends on T3/T7/T8.

We conclude that for this example, WCC outperforms all other "weak" definitions.

Let us next consider the EFSM depicted in Figure 10 which illustrates, among other things, why it may happen that NTSCD produces a smaller slice than SCC but one which fails to satisfy our correctness property.

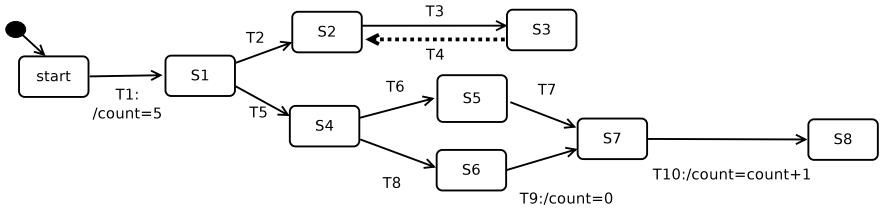

Fig. 10. An EFSM used to illustrate the difference between slices produced using SCC and NTSCD.

With T10 our slicing criterion, data dependence forces T9 and T1 to be included in the slice. For WCC and SCC, in order for S4 to have a unique next observable, we then need to include T6 and T8. For SCC, since there is an infinite path from S1 that avoids S4, we need to include T5.

Concerning NTSCD, NTICD and UNTICD, observe that in this case those dependences equal each other, since the maximal paths equal the ("fair" or not) sink-bounded paths: both are those that end in S8, or alternate between S2 and S3. Slicing using those dependences

- needs to include T8, since T9 depends on T8: the source of T9 can be avoided from the source of T8 but not from the target of T8;
- needs to include T5, since T10 depends on T5: the source of T10 can be avoided from the source of T5 (by choosing T2) but not from the target of T5;
- does *not* need to include T6 since only T7 (not in the slice) depends on T6.

We have explained how the various methods will produce slices for the EFSM in Figure 10; the results are summarized in Table 2.

Thus the slice produced by SCC includes both T6 and T8, rather than just T8; intuitively, this is because they both determine (or control) whether the execution of T10 will be preceded by execution of T9. If we slice away T6, that is replace it by an ε-transition, the sliced EFSM may bypass T9 and then the value of `count` at S8 will be 6. But if T6 has a strong guard, this behavior can*not* be replicated by the original EFSM (which may instead choose T8 and then the

Control Dependence	Slice wrt. T10
WCC	$\{T1, T6, T8, T9, T10\}$
SCC	$\{T1, T5, T6, T8, T9, T10\}$
NTSCD/NTICD/UNTICD	$\{T1, T5, T8, T9, T10\}$

Table 2. Slices produced, by various methods, for the EFSM in Figure 10 with respect to T10.

value of `count` at S8 will be 1). To slice away T6, as done by NTSCD, will thus violate Strong Soundness, as presented in this paper.

Observe that if in Figure 10 we remove (the dotted) transition T4 then the slices produced by the various methods remain the same, except that SCC no longer will include T5 since now there is no infinite path from S1 (that avoids S4).

6.5 Simple Quantitative Comparison of Various Approaches

Let us consider an EFSM for modeling door control (a subcomponent of an elevator control system [30]) which is shown in Figure 11. The door component controls the elevator door, i.e. it opens the door, waits for the passengers to enter or leave the elevator (by using a timer) and finally shuts the door. The data dependenceis, computed using Definition 17, for the door controller are given in Table 3. We adopt a short-hand notation when describing dependences, e.g., `T1` $\rightarrow_{\mathrm{dd}}$ `T2,T3` denotes `T1` $\rightarrow_{\mathrm{dd}}$ `T2` and `T1` $\rightarrow_{\mathrm{dd}}$ `T3`.

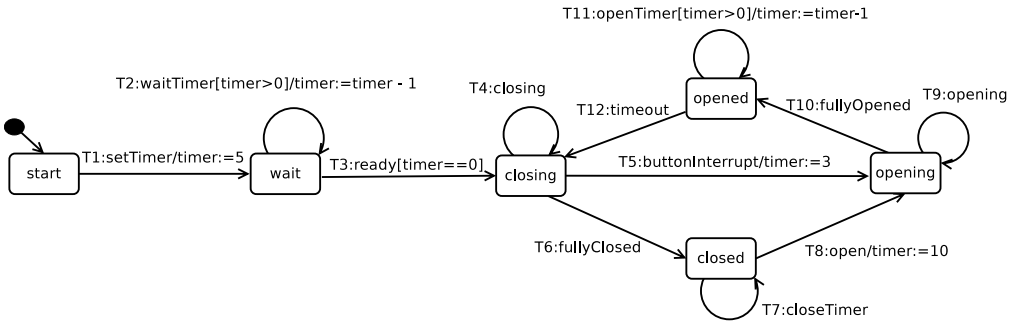

Fig. 11. An EFSM specification for the door controller of the elevator system.

Table 3 also lists the various control dependences computed for the door controller. ICD cannot be applied as there is no (unique) exit state. On the other hand, there is a "control sink" containing all transitions except `T1`, `T2` and `T3`.

NTICD has no dependences because there are no non-termination insensitive control dependences outside of the control sink and it cannot compute any

control dependences within control sinks. UNTICD has dependences within the control sink, due to the self-looping transitions T4, T7, T9, T11. NTSCD has the same dependences within the control sink as UNTICD, as is always the case (as shown in [4]), but because of the self-looping transition T2 in addition some outside: $\texttt{T3} \rightarrow_{\text{NTSCD}} \texttt{T4}, \texttt{T5}, \texttt{T6}$.

DD	$\texttt{T1} \rightarrow_{\text{dd}} \texttt{T2}, \texttt{T3}$	$\texttt{T2} \rightarrow_{\text{dd}} \texttt{T2}, \texttt{T3}$	$\texttt{T11} \rightarrow_{\text{dd}} \texttt{T11}$
	$\texttt{T5} \rightarrow_{\text{dd}} \texttt{T11}$	$\texttt{T8} \rightarrow_{\text{dd}} \texttt{T11}$	
ICD	Cannot be applied as no unique exit state		
NTICD	No dependences		
UNTICD	$\texttt{T5} \rightarrow \texttt{T9}, \texttt{T10}$	$\texttt{T6} \rightarrow \texttt{T7}, \texttt{T8}$	$\texttt{T8} \rightarrow \texttt{T9}, \texttt{T10}$
	$\texttt{T10} \rightarrow \texttt{T11}, \texttt{T12}$	$\texttt{T12} \rightarrow \texttt{T4}, \texttt{T5}, \texttt{T6}$	
NTSCD	$\texttt{T3} \rightarrow \texttt{T4}, \texttt{T5}, \texttt{T6}$	$\texttt{T5} \rightarrow \texttt{T9}, \texttt{T10}$	$\texttt{T6} \rightarrow \texttt{T7}, \texttt{T8}$
	$\texttt{T8} \rightarrow \texttt{T9}, \texttt{T10}$	$\texttt{T10} \rightarrow \texttt{T11}, \texttt{T12}$	$\texttt{T12} \rightarrow \texttt{T4}, \texttt{T5}, \texttt{T6}$

Table 3. Data dependence (DD) and control dependence for the door controller (Figure 11).

We cannot describe WCC and SCC as in Table 3 because WCC and SCC do not relate individual transitions, but describe properties that a slice set must satisfy. For ICD, NTICD, UNTICD and NTSCD, the slice set is the transitive closure, with respect to a slicing criterion, of that control dependence (with data dependence added).

We have computed the slice sets for the door controller EFSM, using data dependence and each control dependence definition, considering each transition in turn as the slicing criterion. In Table 4, we list the size of the slice sets in terms of the number of transitions. Since there are no NTICD dependences, using that dependence does not make any difference (and ICD is not applicable).

Observe that WCC will never add anything to a singleton set, but due to DD some transitions may need to be added, in particular when the slicing criterion is T11. Then DD will add T5 and T8, and for WCC to hold we need to add T6 (as otherwise `closing` will have `closed` as next observable in addition to itself) and T12 (as otherwise `opened` will have `closing` as next observable in addition to itself).

We see that the slice sets computed by UNTICD are typically much larger than those computed by WCC, with the slice sets computed by NTSCD even larger, and (in this case) always equal to those computed by SCC. We shall briefly explain each of these computations, again using T11 as the slicing criterion. In all cases, DD will cause T5 and T8 to be added to the slice.

For UNTICD, we see from Table 3 that we need to add T6, T10, and T12. The presence of T10 is the difference between UNTICD and WCC, and arises because UNTICD is sensitive to non-termination inside the control sink; from the source of T10 it is possible, due to the self-looping transition T9, to avoid the source of T12.

For NTSCD, we see from Table 3 that (due to the presence of `T5`) we need to add also `T3`, and by data dependence even `T1` and `T2`. (The only transitions not included in the slice are thus the self-looping `T4`, `T7`, and `T9`.)

For the least set satisfying SCC, observe that it must satisfy WCC and hence contain {`T5`, `T6`, `T8`, `T11`, `T12`}. With that slice set, the next observable of `opening` is `opened`, but (due to `T9`) there is an infinite path from `opening` that avoids `opened`, so `T10` must be added. Also, the next observable of `wait` is `closing`, but (due to `T2`) there is an infinite path from `wait` that avoids `closing`, so `T3` must be added, and by DD also `T1` and `T2`. The resulting slice thus equals the slice for NTSCD.

Slicing Criterion	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
DD, or NTICD + DD	1	2	3	1	1	1	1	1	1	1	3	1
WCC (+ DD)	1	2	3	1	1	1	1	1	1	1	5	1
UNTICD (+ DD)	1	2	3	6	5	5	6	5	6	5	6	5
SCC or NTSCD (+ DD)	1	2	3	9	8	8	9	8	9	8	9	8

Table 4. For each kind of control dependence, and each transition t in the EFSM of the door controller (Figure 11), we use that control dependence to slice with respect to a slicing criterion that consists of only t, and list the size (*i.e.*, the number of transitions) of the resulting slice.

7 Empirical Evaluation

We shall now explore the practical usefulness of using WCC and SCC to slice EFSMs. Our measure will be the size of the slices produced, since if slices are small compared to the original models then they provide advantages, such as aiding users in understanding the models. We thus pose the following research questions:

RQ1 What is the typical size of a backward slice produced using WCC or SCC?
RQ2 How do the sizes of backward slices produced by WCC and SCC compare to those produced using existing control dependence definitions for EFSMs?

We use thirteen EFSMs, described in Table 5, as subjects for our experiments. The EFSMs are drawn from a variety of sources. The first six, which we collectively label as group A, were used in the research of model-based slicing with traditional dependence analysis [20]; each has a unique exit state (reachable from all other states). The last seven EFSMs have no exit states; we have divided them into two groups, labeled B and C:

– group B consists of four machines, where TCP [36] is extracted from a SDL specification, and so is TCSbin which is a proprietary real-world model from Motorola; when extracting into EFSMs, we have done some simplifications,

including the omission of history states. DoorController has already been described (Section 6); INRES is an example of an EFSM specified protocol as described in [8].
– group C consists of Alarm, Tamagotchi and WaterPump which are flattened state machines from the study of clustering finite state machines [14]; the transitions in these machines contain only events (that is, not guards nor actions).

A visual inspection of the EFSMs in group A reveals that their structure varies even though they all have a unique exit state; of the EFSMs in group A, only FuelPump has a CFG-like structure (*i.e.*, almost all transitions belong to an cycle-free path from the initial state to the exit state) while the rest of the EFSMs in group A contain a few strongly connected components.

All of the EFSMs from group B and C, apart from DoorController (as in Figure 11), consist of a large control sink (of which all transitions are part).

Table 5. EFSMs used in our experiments.

EFSMs	$\lvert\hat{S}\rvert$	$\lvert\hat{T}\rvert$	$\lvert\hat{V}\rvert$	EXIT	Description
ATM	9	23	8	Yes	Automated Teller Machine [20]
Cashier	12	21	10	Yes	Cashier Machine [20]
CruiseControl	5	17	18	Yes	Cruise Control System [20]
FuelPump	13	25	12	Yes	Fuel Pump System [20]
PrinTok	11	89	5	Yes	Print Token [20]
VendingMachine	7	28	7	Yes	Vending Machine system [20]
INRES	8	18	6	No	INRES protocol [8]
DoorController	6	12	1	No	Door Controller of the Lift System [30]
TCP	12	57	31	No	TCP Standard(RFC793) [36]
TCSbin	24	65	61	No	Telephony Protocol (Motorola)
Alarm	10	25	0	No	Alarm Clock [14]
WaterPump	11	14	0	No	Water Pump Controller [14]
Tamagotchi	15	38	0	No	Virtual Pet [14]
Total	143	432	162		

7.1 Implementation and Tool

In [5], a tool was developed in Python to automatically compute slices (both forward and backward) for EFSMs by taking closures of data dependence, and any of the NTSCD, NTICD and UNTICD dependences. In this paper, to observe the effect of different control dependences we shall consider only backward slices.

For an EFSM M the tool will first compute, among the transitions in M, the data dependences, using Definition 17, and then the control dependences, using (based on the user's choice) one of the three control dependence definitions: NTSCD, NTICD, UNTICD. These dependences will be represented as a dependence graph, which is a directed graph whose nodes represent transitions in M

and whose edges represent either data or control dependences between transitions. The slicing algorithm marks all backwardly reachable transitions from the slicing criterion c, *i.e.*, the transitive closure of c wrt. control and data dependence All marked transitions will be in the slice set $\mathcal{L}$; all unmarked transitions become ε-transitions.

We have extended the tool described in [5] by adding Python implementations of the WCC and SCC algorithms. For an EFSM M the tool will first compute, using Definition 17, the data dependences among the transitions in M and produce the corresponding dependence graph. Then the slice set $\mathcal{L}$ will initially consist of the slicing criterion c and any transitions that are data dependent on c, *i.e.*, that are backwardly reachable from c in the dependence graph. Then according to the WCC or SCC algorithm as defined in Algorithm 1 and Algorithm 2, transitions will be added to the slice. For any new transitions that are added to the slice, the dependence graph will be checked and any backwardly reachable transitions from the new transitions will be added. This process will continue until no new transitions are added in $\mathcal{L}$; all transitions not in $\mathcal{L}$ become ε-transitions.

An algorithm for removing ε-transitions is described in [2]; it is an adaptation of Ilie and Yu's NFA minimisation algorithm [18]. This algorithm has been used with either NTSCD, NTICD, UNTICD to produce slices that are amorphous, in that they are not sub-graphs of the original. In fact, in some cases the number of transitions can be larger than the original, though empirical results in [2] show that in practice this did not occur. This algorithm could be applied to the slices produced with Algorithm 1 (for WCC) or Algorithm 2 (for SCC), to remove ε-transitions and merge equivalent states.

For the empirical evaluation, we consider the size of the slice sets for WCC and SCC, respectively, as well as for the slice sets for the three control dependences: NTSCD, NTICD, UNTICD. We do not consider the slice sizes of ICD, since ICD is applicable only for EFSMs with a unique exit state (group A), and for such EFSMs it is shown [4] that ICD is a special case of UNTICD and of NTICD (*i.e.*, ICD, UNTICD and NTICD produce the same slices for EFSMs that have a unique exit state).

7.2 Metrics

We shall now formalise the metrics used for measuring slice sizes for a given EFSM M with transitions $\hat{T}$. With m a slicing method, and c a slicing criterion, we shall define $\mathcal{L}(M, m, c)$ as the slice set (computed as sketched in Section 7.1) given by:

- if m is WCC or SCC, $\mathcal{L}(M, m, c)$ is the least set that satisfies that property, is closed under data dependence, and contains c;
- if m is NTSCD or NTICD or UNTICD, $\mathcal{L}(M, m, c)$ is the least set that is closed under that dependence and under data dependence, and contains c.

We shall define $\mathsf{ratio}(M, m, c)$, the *slicing ratio* of method m (on M) wrt. c, as the percentage of transitions that get included in the slice:

$$\mathsf{ratio}(M, m, c) = \frac{|\mathcal{L}(M, m, c)|}{|\hat{T}|}.$$

The number of possible slicing criteria is exponential in $|\hat{T}|$, but we shall expect slicing criteria to be "small", and restrict our attention to slicing criteria of a certain size; we define $\mathsf{avg\text{-}ratio}(M, m, sc)$ as the average (*i.e.*, mean) slicing ratio, taken over all slicing criteria of size sc. That is:

$$\mathsf{avg\text{-}ratio}(M, m, sc) = \frac{\sum_{c \in \mathcal{P}(\hat{T}) \ | \ |c|=sc} \mathsf{ratio}(M, m, c)}{\frac{|\hat{T}|!}{sc!\ (|\hat{T}|-sc)!}}.$$

In particular, we have:

$$\mathsf{avg\text{-}ratio}(M, m, 1) = \frac{\sum_{t \in \hat{T}} \mathsf{ratio}(M, m, \{t\})}{|\hat{T}|}.$$

With X the set of all EFSM models considered in this section (thus $|X| = 13$), we finally define $\mathsf{total\text{-}avg\text{-}ratio}(m, sc)$, the *average singleton slicing ratio* of method m for slicing criteria of size sc, as

$$\mathsf{total\text{-}avg\text{-}ratio}(m, sc) = \frac{\sum_{M \in X} \mathsf{avg\text{-}ratio}(M, m, sc)}{|X|}.$$

7.3 Results Concerning RQ1: Typical Slices Using WCC or SCC

To give a tentative answer to the first research question, we compute for each EFSM M (of the thirteen given): $\mathsf{avg\text{-}ratio}(M$,SCC,1), $\mathsf{avg\text{-}ratio}(M$,WCC,1), and $\mathsf{avg\text{-}ratio}(M$,WCC,2). (We could also have computed $\mathsf{avg\text{-}ratio}(M$,SCC,2) but even for a singleton slicing criteria, SCC produces slices that are not much smaller than the original EFSM.) The results, to be discussed in the following paragraphs, are given in Table 6, whose last line lists $\mathsf{total\text{-}avg\text{-}ratio}$(SCC,1), $\mathsf{total\text{-}avg\text{-}ratio}$(WCC,1), and $\mathsf{total\text{-}avg\text{-}ratio}$(WCC,2).

Discussing the results for WCC The results for WCC are depicted in Table 6. We first look at the case (second rightmost column) where the slicing criterion is a single transition, and observe that then the average slicing ratio for many EFSMs is very low (for FuelPump, PrinTok, Alarm, WaterPump and Tamagotchi, even less than 10%). On closer inspection, this is not surprising, since if the transition that forms the slicing criterion is not data dependent on anything then WCC will cause no other transitions to be added and the resulting slice will have size 1; indeed, 62.26% of all slices computed are of size one. (In particular, if the EFSM uses *no* variables, as is the case for the three EFSMs in group C (Alarm, Tamagotchi and WaterPump), then *all* slices will have size 1.)

Table 6. Average slicing ratios computed using WCC or SCC and data dependence for the thirteen EFSMs. The variable *sc* refers to the number of transitions used as a slicing criterion.

Group	EFSMs	SCC (sc=1)	WCC (sc=1)	WCC (sc=2)
A	ATM	33.46%	16.45%	29.11%
	Cashier	41.95%	14.74%	28.71%
	CruiseControl	46.37%	41.86%	58.43%
	FuelPump	28%	8.8%	19.11%
	PrinTok	60.45%	6.02%	43.02%
	VendingMachine	79.97%	32.40%	71.68%
B	INRES	60.19%	25.31%	40.96%
	DoorController	56.94%	13.19%	23.36%
	TCP	54.05%	41.36%	61.02%
	TCSbin	83.6%	42.84%	70.94%
C	Alarm	92.32%	4%	47.25%
	WaterPump	40.31%	7.14%	25.67%
	Tamagotchi	82.48%	2.63%	34.78%
	Total Average	58.46%	21.11%	42.62%

We therefore perform a second experiment (rightmost column in Table 6) where we consider slicing criteria that consist of *two* transitions. As expected, this results in substantially larger slices, but typically still containing less than half of the original transitions.

Discussing the results for SCC (with singleton slicing criteria) The results for SCC are also depicted in Table 6. Since a slice set that satisfies SCC also satisfies WCC, no SCC slice can be smaller than the corresponding WCC slice, and in fact, the typically SCC slice is significantly larger. In particular, only 3.01% of all SCC slices are of size 1 (recall that for WCC the number is 62.26%).

For CruiseControl, however, the average SCC slice is only slightly larger than the average WCC slice. In fact, a closer inspection reveals that the slices are identical, except when the slicing criterion consists of a certain transition (T17); in that situation, transitions need to be added because of an infinite path that avoids the next observable.

Considering Median Slicing Ratio (versus Average) One could argue that when measuring slicing ratios, the *median* is more interesting than the average (mean), as the latter may be severely impacted by a few extreme cases. We shall therefore also compute the median of the slicing ratios, but (in this paper) only when the slicing criterion is a singleton; we implement that by first sorting the $|\hat{T}|$ slicing ratios and then *(i)* picking the $(|\hat{T}|+1)/2$ largest when $|\hat{T}|$ is odd, and *(ii)* picking the mean of the $|\hat{T}|/2$ largest and the $|\hat{T}|/2+1$ largest when $|\hat{T}|$ is even.

The results are depicted in Table 7. Comparing with Table 6, we see that

– for SCC, the median slicing ratio is rather close to the average slicing ratio;
– for WCC, the median slicing ratio is often much smaller than the average slicing ratio.

These results are rather unsurprising, given that we saw (Section 7.3) that most WCC slices are very small; hence the median is also very small whereas the few large slices boost the average.

Table 7. The **median** of slicing ratios computed using WCC or SCC and data dependence is given for each of the thirteen EFSMs (with the slicing criterion always a singleton).

Group	EFSMs	SCC (sc = 1)	WCC (wc = 1)
A	ATM	34.78%	13.04%
	Cashier	42.86%	9.52%
	CruiseControl	29.41%	29.41%
	FuelPump	24%	4%
	PrinTok	60.67%	1.12%
	VendingMachine	82.14%	3.57%
B	INRES	61.11%	5.56%
	DoorController	66.67%	8.33%
	TCP	52.63%	52.63%
	TCSbin	84.21%	81.54%
C	Alarm	96%	4%
	WaterPump	42.86%	7.14%
	Tamagotchi	84.21%	2.63%

A third way of summarizing the data, not considered in this work, would be to compute the *geometric* mean, rather than the arithmetic mean.

7.4 Results Concerning RQ2(a): Comparing WCC to Previous Control Dependences

To give a tentative answer to the first part of the second research question, comparing WCC to previous definitions of non-termination *in*sensitive control dependence, we compute for each EFSM M (of the thirteen given): avg-ratio(M,NTICD,1) and avg-ratio(M,UNTICD,1); we even compute avg-ratio(M,DD,1) where DD generates slices using only data dependence. The results, together with the corresponding results for WCC (taken from Table 6), are given in Table 8, whose last line lists total-avg-ratio(WCC,1), total-avg-ratio(NTICD,1), total-avg-ratio(UNTICD,1), and total-avg-ratio(DD,1).

For the EFSMs in group A, each with a unique exit state, UNTICD coincides with NTICD (as the final transition that leads to the exit state forms a trivial control sink where "fairness" holds vacuously). On the other hand, WCC appears to produce slices that are somewhat smaller, at least on average. On closer inspection, we found that only in the VendingMachine are the sizes of some of

Table 8. Average slicing ratios, for singleton slicing criteria, for methods insensitive to non-termination.

Group	EFSMs	WCC	NTICD	UNTICD	DD
A	ATM	16.45%	23.25%	23.25%	14.18%
	Cashier	14.74%	67.57%	67.57%	11.34%
	CruiseControl	41.86%	70.24%	70.24%	31.49%
	FuelPump	8.8%	9.28%	9.28%	7.52%
	PrinTok	6.02%	60.45%	60.45%	1.88%
	VendingMachine	32.40%	34.69%	34.69%	18.88%
B	INRES	25.31%	15.74%	60.19%	15.74%
	DoorController	13.19%	11.81%	38.19%	11.81%
	TCP	41.36%	6.77%	42.97%	6.77%
	TCSbin	42.84%	5.61%	82.06%	5.61%
C	Alarm	4%	4%	92.32%	4%
	WaterPump	7.14%	7.14%	40.31%	7.14%
	Tamagotchi	2.63%	2.63%	82.48%	2.63%
	Total Average	21.11%	24.55%	54.15%	10.69%

the slices produced using NTICD smaller than those produced using WCC. This is because WCC includes "order" dependences, which NTICD/UNTICD don't (see Figure 10 and Table 2 for an example).

For the EFSMs in group B or C, the control sinks are typically large, and as UNTICD coincides with (the non-termination sensitive) NTSCD within a control sink, UNTICD produces large slices. On the other hand, NTICD (that considers only "fair" paths within control sinks) will add no transitions to the slices within control sinks (apart from what has already been added by DD). For the EFSMs in group B, it is thus no wonder that NTICD produces smaller slices than WCC does (e.g. see Table 4 for the size of the slices produced for the DoorController with respect to transition T11), but one should keep in mind that no correctness property has ever been proven, or even stated, for NTICD.

For the EFSMs in group C, having no variables and hence no data dependences, WCC as well as NTICD gives slices of size 1 when given a slicing criterion of size 1.

We have carried out further investigations where we focus on non-trivial slices, and therefore remove from consideration all slicing criteria that result in a slice of size one (in particular, only EFSMs from groups A and B were taken into account). Doing so, we found that the total average slicing ratio for WCC is 51.04% (still, 20.24% of all WCC slices are equal to those computed using only DD). We can compare this total to the results from [5] (which also does not consider slices of size 1, in particular not the EFSMs from group C, but does consider all the EFSMs from groups A and B); there it is found that NTICD produces an average slicing ratio of 49.48% (but remember that for group B this ratio does not mean much) while UNTICD produces an average slicing ratio of 66.83%.

7.5 Results Concerning RQ2(b): Comparing SCC to Previous Control Dependences

To give a tentative answer to the second part of the second research question, comparing SCC to previous definitions of non-termination *sensitive* control dependence, we compute avg-ratio(M,NTSCD,1) for each EFSM M (of the thirteen given). The results, together with the corresponding results for SCC (taken from Table 6) and UNTICD (taken from Table 8), are given in Table 9, whose last line lists total-avg-ratio(SCC,1), total-avg-ratio(NTSCD,1), and total-avg-ratio(UNTICD,1).

Table 9. Average slicing ratios, for singleton slicing criteria, for methods sensitive to non-termination.

Group	EFSMs	SCC	NTSCD	UNTICD
A	ATM	33.46%	32.70%	23.25%
	Cashier	41.95%	71.66%	67.57%
	CruiseControl	46.37%	70.24%	70.24%
	FuelPump	28%	26.72%	9.28%
	PrinTok	60.45%	60.45%	60.45%
	VendingMachine	79.97%	79.97%	34.69%
B	INRES	60.19%	60.19%	60.19%
	DoorController	56.94%	56.94%	38.19%
	TCP	54.05%	42.97%	42.97%
	TCSbin	83.6%	82.06%	82.06%
C	Alarm	92.32%	92.32%	92.32%
	WaterPump	40.31%	40.31%	40.31%
	Tamagotchi	82.48%	82.48%	82.48%
	Total Average	58.46%	61.46%	54.15%

First observe that since UNTICD coincides with NTSCD within a control sink, it is not surprising that UNTICD often equals NTSCD (but otherwise produces smaller slices). Next we compare SCC to NTSCD and see that on average, SCC produces much smaller slices for two EFSMs (Cashier and CruiseControl), but for four EFSMs produces slices that are slightly (except for TCP) larger; for the remaining EFSMs, the (average) slicing ratios are identical. Moreover, SCC has a slightly smaller total average slicing ratio than NTSCD.

It may seem discomforting that SCC sometimes produces slices that are larger than those produced by NTSCD. But keep in mind that no correctness property has been proven, or even stated, for NTSCD in the context of EFSMs. In the context of programs expressed as CFGs, the control dependence with that name [26] has been proven correct, but for irreducible CFGs only when applied together with a kind of "order dependence" as described in [27]. The properties WCC and SCC have been designed to include the notion of order dependence.

8 Conclusion

We have proposed algorithms, running in low polynomial time, for slicing EFSMs. The algorithms are based on notions (a "weak" and a "strong") of *commitment closure* adapted from the work by Danicic et al. [9] who originally phrased them for programs represented as control-flow graphs. We have proposed new semantic definitions of "correct slices" that integrate interaction, non-determinism and non-termination, and proved that these criteria are satisfied by slices produced by our algorithms.

We have conducted experiments using both benchmark and real world production EFSMs to measure the practical usefulness of our slicing algorithms and to compare them with slices computed using existing definitions of control dependence. Slicing wrt. "weak" commitment closure (WCC) will often significantly reduce the size of the EFSMs (the average relative slice size is 21% if the slicing criterion is a singleton), while slicing wrt. "strong" commitment closure will often give a modest size reduction (the average relative slice size is 58%). In both cases, this is typically smaller than what one gets by using algorithms based on previous definitions with similar aims, but not always — this reflects that we found that previous algorithms do not always satisfy the correctness properties presented in this paper.

To establish a firm foundation for our approach, we would like to eventually use a proof assistant to formally verify our algorithms, as was recently done by Léchenet et al. [22] for the computation of weak commitment closure (but not yet handling data dependence) in the original work by Danicic et.al. [9]. Also Wasserrab [32] and Blazy et al. [7] have formally verified algorithms for slicing.

To model bidirectional interaction with the environment, we would like to consider EFSMs that not just consume events but also *generate* them, and investigate which new notions are needed in order to accommodate the slicing of such EFSMs.

In order to make slicing even more useful, it is desirable to generate smaller slices than we currently do. To do so, without compromising correctness, recall (Section 4.3) that WCC was motivated as a sufficient condition for weak soundness, *no matter the values of the guards* (similarly for SCC wrt. strong soundness). While simple to implement, this is very conservative. We would like to investigate a more liberal version of WCC that while still sufficient for weak soundness may allow a node to have more than one observable, as long as certain guards have certain values. Equipped with a suitable data flow analysis, we believe this will result in significantly fewer transitions being "sucked" into a slice.

References

1. Amtoft, T.: Slicing for modern program structures: a theory for eliminating irrelevant loops. Information Processing Letters **106**(2), 45–51 (2008). https://doi.org/http://dx.doi.org/10.1016/j.ipl.2007.10.002

2. Androutsopoulos, K., Clark, D., Harman, M., Hierons, R.M., , Li, Z., Tratt, L.: Amorphous slicing of extended finite state machines. IEEE Transactions on Software Engineering **39**(7), 892–909 (2013)
3. Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., Tratt, L.: State-based model slicing: A survey. ACM Computing Surveys **45**(4), 53:1–53:36 (Aug 2013)
4. Androutsopoulos, K., Clark, D., Harman, M., Li, Z., Tratt, L.: Control dependence for extended finite state machines. In: Fundamental Approaches to Software Engineering (FASE), part of the European Joint Conferences on the Theory and Practice of Software (ETAPS), York, UK. LNCS, vol. 5503, pp. 216–230. Springer Berlin/Heidelberg (March 2009)
5. Androutsopoulos, K., Gold, N., Harman, M., Li, Z., Tratt, L.: A theoretical and empirical study of EFSM dependence. In: 25^{th} IEEE International Conference on Software Maintenance (ICSM 2009), September 20–26, 2009, Edmonton, Alberta, Canada. pp. 287–296. IEEE Computer Society (2009)
6. Ball, T., Horwitz, S.: Slicing programs with arbitrary control-flow. In: Fritzson, P. (ed.) 1^{st} International Workshop on Automated and Algorithmic Debugging. LNCS, vol. 749, pp. 206–222. Springer, Linköping, Sweden (1993), also available as University of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992
7. Blazy, S., Maroneze, A., Pichardie, D.: Verified validation of program slicing. In: Proceedings of the 2015 Conference on Certified Programs and Proofs. pp. 109–117. CPP '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2676724.2693169, http://doi.acm.org/10.1145/2676724.2693169
8. Bourhfir, C., Dssouli, R., Aboulhamid, E.M., Rico, N.: Automatic executable test case generation for extended finite state machine protocols. In: Kim, M., Kang, S., Hong, K. (eds.) Testing of Communicating Systems: IFIP TC6 10th International Workshop on Testing of Communicating Systems (IWTCS'97), 8–10 September 1997, Cheju Island, Korea. pp. 75–90. Springer US, Boston, MA (1997)
9. Danicic, S., Barraclough, R.W., Harman, M., Howroyd, J.D., Kiss, A., Laurence, M.R.: A unifying theory of control dependence and its application to arbitrary program structures. Theoretical Computer Science **412**(49), 6809–6842 (Nov 2011)
10. Derler, P., Lee, E.A., Sangiovanni-Vincentelli, A.L.: Modeling cyber-physical systems. Proceedings of the IEEE **100**(1), 13–28 (Jan 2012). https://doi.org/10.1109/JPROC.2011.2160929
11. Fragal, V.H., Simão, A., Mousavi, M.R.: Validated test models for software product lines: Featured finite state machines. In: Kouchnarenko, O., Khosravi, R. (eds.) Formal Aspects of Component Software - 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10231, pp. 210–227. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-57666-4_13
12. Ganapathy, V., Ramesh, S.: Slicing synchronous reactive programs. Electronic Notes in Theoretical Computer Science **65**(5), 50–64 (Jul 2002), SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of ETAPS 2002)
13. Gold, N.E., Binkley, D., Harman, M., Islam, S., Krinke, J., Yoo, S.: Generalized observational slicing for tree-represented modelling languages. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 547–558. ESEC/FSE 2017, ACM, New York, NY, USA (2017)

14. Hall, M., McMinn, P., Walkinshaw, N.: Superstate identification for state machines using search-based clustering. In: Pelikan, M., Branke, J. (eds.) Proceedings of the 12th annual conference on Genetic and Evolutionary Computation (GECCO 2010). pp. 1381–1388. ACM (2010)
15. Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts: The Statemate Approach. McGraw-Hill, Inc., New York, NY, USA, 1st edn. (1998)
16. Hatcliff, J., Corbett, J.C., Dwyer, M.B., Sokolowski, S., Zheng, H.: A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In: Static Analysis, 6th International Symposium (SAS'99). LNCS, vol. 1694, pp. 1–18. Springer (1999)
17. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher-Order and Symbolic Computation **13**(4), 315–353 (Dec 2000)
18. Ilie, L., Yu, S.: Reducing NFAs by invariant equivalences. Theoretical Computer Science **306**(1-3), 373–390 (2003)
19. Kamischke, J., Lochau, M., Baller, H.: Conditioned model slicing of feature-annotated state machines. In: Proceedings of the 4th International Workshop on Feature-Oriented Software Development. pp. 9–16. FOSD '12, ACM, New York, NY, USA (2012)
20. Korel, B., Singh, I., Tahat, L., Vaysburg, B.: Slicing of state-based models. In: IEEE International Conference on Software Maintenance (ICSM'03). pp. 34–43. IEEE Computer Society Press, Los Alamitos, California, USA (Sep 2003)
21. Labbé, S., Gallois, J.: Slicing communicating automata specifications: Polynomial algorithms for model reduction. Formal Aspects of Computing **20**(6), 563–595 (2008). https://doi.org/http://dx.doi.org/10.1007/s00165-008-0086-3
22. Léchenet, J.C., Kosmatov, N., Gall, P.L.: Fast computation of arbitrary control dependencies. In: FASE'18. LNCS, vol. 10802, pp. 207–224. Springer (2018)
23. Lity, S., Morbach, T., Thüm, T., Schaefer, I.: Applying incremental model slicing to product-line regression testing. In: Software Reuse: Bridging with Social-Awareness - 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016. Lecture Notes in Computer Science, vol. 9679, pp. 3–19. Springer, New York, NY, USA (2016)
24. Marwedel, P.: Embedded and cyber-physical systems in a nutshell. DAC.COM Knowledge Center Article **20** (2010)
25. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Transactions on Software Engineering **16**(9), 965–979 (1990)
26. Ranganath, V.P., Amtoft, T., Banerjee, A., Dwyer, M.B., Hatcliff, J.: A new foundation for control-dependence and slicing for modern program structures. In: 14th European Symposium on Programming (ESOP 2005), held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. LNCS, vol. 3444, pp. 77–93. Springer (2005)
27. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Transactions on Programming Languages and Systems **29**(5) (Aug 2007). https://doi.org/http://doi.acm.org/10.1145/1275497.1275502
28. Silva, J.: A vocabulary of program slicing-based techniques. ACM Computing Surveys **44**(3), 12:1–12:41 (Jun 2012)
29. Sivagurunathan, Y., Harman, M., Danicic, S.: Slicing, I/O and the implicit state. In: AADEBUG'97. Proceedings of the Third International Workshop on Automatic Debugging: Linköping, Sweden. pp. 59–67. Linköping Electronic Articles in Computer and Information Science (May 1997)

30. Strobl, F., Wisspeintner, A.: Specification of an elevator control system – an autofocus case study. Tech. Rep. TUM-I9906, Technische Universität München (1999)
31. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3**(3), 121–189 (Sep 1995)
32. Wasserrab, D.: From Formal Semantics to Verified Slicing. Ph.D. thesis, Karlsruher Institut für Technologie (2010)
33. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering **10**(4), 352–357 (1984)
34. Weiser, M.D.: Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, University of Michigan, Ann Arbor, MI (1979)
35. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes **30**(2), 1–36 (2005)
36. Zaghal, R.Y., Khan, J.I.: EFSM/SDL modeling of the original TCP standard (RFC793) and the congestion control mechanism of TCP Reno. Tech. Rep. TR2005-07-22, Networking and Media Communications Research Laboratories, Department of Computer Science, Kent State University (July 2005)

A Proofs of Completeness and (Weak/Strong) Soundness

We shall now prove the correctness theorems from Section 4, but shall first establish some generally applicable results.

A.1 General Results

Lemma 31 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}}$, and that $t \notin \mathcal{L}$. With $n = \mathsf{S}(t)$ and $n' = \mathsf{T}(t)$ we then have $Rv_{\mathcal{L}}(n') \subseteq Rv_{\mathcal{L}}(n)$, and if $i \vdash t : (n,s) \stackrel{E}{\rightarrow} (n',s')$ for some $i \in \{1,2\}$ then $s(v) = s'(v)$ for all $v \in Rv(n')$.*

Proof. Consider $v \in Rv(n')$. There exists a path $[t_1..t_k]$ with $k \geq 1$ from n' such that $t_k \in \mathcal{L}$ with $v \in \mathsf{U}(t_k)$ and such that $v \notin \mathsf{D}(t_j)$ for all $j \in 1 \ldots k-1$.

Assume, to get a contradiction, that $v \in \mathsf{D}(t)$. Then $t \rightarrow_{\mathrm{dd}} t_k$, which together with $t_k \in \mathcal{L}$ and $t \notin \mathcal{L}$ contradicts $\mathcal{L}$ being closed under $\rightarrow_{\mathrm{dd}}$.

Hence $v \notin \mathsf{D}(t)$. Thus the path $[t, t_1..t_k]$ will establish $v \in Rv(n)$, and if $i \vdash t : (n,s) \stackrel{E}{\rightarrow} (n',s')$ then $s'(v) = s(v)$ (no matter whether $i = 1$ or $i = 2$).

Lemma 32 *Assume that for $t \in \mathcal{L}$ we have*

$$1 \vdash t : (n, s_1) \stackrel{E}{\rightarrow} (n', s_1') \text{ and}$$
$$2 \vdash t : (n, s_2) \stackrel{E}{\rightarrow} (n', s_2')$$

where $s_1(v) = s_2(v)$ for all $v \in Rv(n)$. Then $s_1'(v) = s_2'(v)$ for all $v \in Rv(n')$.

Proof. Let $v \in Rv(n')$ be given, so as to show $s_1'(v) = s_2'(v)$. We split into two cases.

The first case is when $v \notin \mathsf{D}(t)$. Then $v \in Rv(n)$, implying the desired equality $s_1'(v) = s_1(v) = s_2(v) = s_2'(v)$.

The second case is when $v \in \mathsf{D}(t)$, where we must prove that $[\![A]\!]s_1 = [\![A]\!]s_2$ where $A = \mathsf{A}(t)[c/v_b]$ if E is of the form $e(c)$, and $A = \mathsf{A}(t)$ otherwise. But this follows from the fact that for all $w \in \mathit{fv}(A) = \mathit{fv}(\mathsf{A}(t)) \setminus \{v_b\}$ we have $s_1(w) = s_2(w)$, since $w \in \mathsf{U}(t)$ and thus (as $t \in \mathcal{L}$) also $w \in \mathit{Rv}(t) \subseteq \mathit{Rv}(n)$.

A.2 Proving Completeness

We shall first establish some intermediate results:

Lemma 33 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}}$. If with $t \notin \mathcal{L}$ we have*

- $1 \vdash t : (n, s_1) \xrightarrow{E_1} (n', s_1')$ *and*
- $s_1(v) = s_2(v)$ *for all* $v \in \mathit{Rv}(n)$

then

- $2 \vdash t : (n, s_2) \xrightarrow{\varepsilon} (n', s_2)$ *and*
- $s_1'(v) = s_2(v)$ *for all* $v \in \mathit{Rv}(n')$.

where $\mathsf{filter}(E_1) = \varepsilon$ *if $\mathcal{L}$ is uniform.*

Proof. Since $t \notin \mathcal{L}$, we see that $\mathsf{G}_2(t) = \mathit{true}$ and $\mathsf{E}_2(t) = \varepsilon$ and $\mathsf{D}_2(t) = \emptyset$ which yields the first claim. For the second claim, consider $v \in \mathit{Rv}(n')$: since $t \notin \mathcal{L}$, we infer by Lemma 31 that $s_1(v) = s_1'(v)$, and that $v \in \mathit{Rv}(n)$ which by assumption implies $s_1(v) = s_2(v)$; hence we can conclude that $s_1'(v) = s_2(v)$. The last claim follows from Fact 11.

Lemma 34 *If with $t \in \mathcal{L}$ we have*

- $1 \vdash t : (n, s_1) \xrightarrow{E} (n', s_1')$ *and*
- $s_1(v) = s_2(v)$ *for all* $v \in \mathit{Rv}(n)$

then there exists s_2' such that

- $2 \vdash t : (n, s_2) \xrightarrow{E} (n', s_2')$ *and*
- $s_1'(v) = s_2'(v)$ *for all* $v \in \mathit{Rv}(n')$.

Proof. Our assumptions entail $n = \mathsf{S}(t)$, $n' = \mathsf{T}(t)$, $\mathsf{G}_1(t) = \mathsf{G}_2(t) = \mathsf{G}(t)$, and $[\![g]\!]s_1 = \mathit{true}$ where $g = \mathsf{G}(t)[c/v_b]$ if E is of the form $e(c)$, and $g = \mathsf{G}(t)$ otherwise. For an arbitrary $w \in \mathit{fv}(\mathsf{G}(t)) \setminus \{v_b\}$ we infer by Fact 13 that $w \in \mathit{Rv}(t) \subseteq \mathit{Rv}(n)$, implying $s_1(w) = s_2(w)$. Hence also $[\![g]\!]s_2 = \mathit{true}$, implying that there exists s_2' such that $2 \vdash t : (n, s_2) \xrightarrow{E} (n', s_2')$. That $s_1'(v) = s_2'(v)$ for all $v \in \mathit{Rv}(n')$ now follows from Lemma 32.

Theorem 1 (Completeness): Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}}$. If

- $1 \vdash_{\mathrm{ni}} \pi : C_1 \xrightarrow{E} C_1'$ and
- $C_1 \; Q \; C_2$

then there exists C_2' such that

- $2 \vdash \pi : C_2 \xrightarrow{E} C_2'$ and
- $C_1' \; Q \; C_2'$.

Moreover, if $\mathcal{L}$ is uniform then $2 \vdash_{\mathrm{ni}} \pi : C_2 \xrightarrow{\mathsf{filter}(E)} C_2'$.

Proof. We do induction in the length of π. If π is empty, then $C_1' = C_1$ and $E_1 = \varepsilon$, and the claim is trivial with $C_2' = C_2$.

If π is a singleton t, we split into two cases: if $t \in \mathcal{L}$ then the claim follows from Lemma 34; if $t \notin \mathcal{L}$ then the claim follows from Lemma 33.

Now assume that π can be written $\pi = \pi''\pi'$ with π' and π'' not empty. It is easy to see that there exists E'' and E' with $E = E''E'$ such that $1 \vdash_{\mathrm{ni}} \pi'' : C_1 \xrightarrow{E''} C_1''$ and $1 \vdash_{\mathrm{ni}} \pi' : C_1'' \xrightarrow{E'} C_1'$. Inductively on π'', there exists C_2'' with $C_1'' \; Q \; C_2''$ such that $2 \vdash \pi'' : C_2 \xrightarrow{E''} C_2''$; inductively on π', there then exists C_2' with $C_1' \; Q \; C_2'$ such that $2 \vdash \pi' : C_2'' \xrightarrow{E'} C_2'$. But then it is easy to see that we have the desired $2 \vdash \pi : C_2 \xrightarrow{E} C_2'$. Finally, if $\mathcal{L}$ is uniform, we inductively get $2 \vdash_{\mathrm{ni}} \pi'' : C_2 \xrightarrow{\mathsf{filter}(E'')} C_2''$ and $2 \vdash_{\mathrm{ni}} \pi' : C_2'' \xrightarrow{\mathsf{filter}(E')} C_2'$ and thus clearly also $2 \vdash_{\mathrm{ni}} \pi : C_2 \xrightarrow{\mathsf{filter}(E'')\mathsf{filter}(E')} C_2'$ which yields the claim since $\mathsf{filter}(E) = \mathsf{filter}(E'')\mathsf{filter}(E')$.

A.3 Proving Soundness

As discussed in Section 3.4, we cannot quite hope for the converse of Theorem 1 in that the original EFSM may get stuck, or loop, rather than reach the next observable state. This is formalized by the following result:

Lemma 35 *Let $obs(n) = \{m\}$. Given s, one of the 3 cases below applies:*

1. *there exists E and s' such that $1 \vdash_{ni} (n, s) \stackrel{E}{\Longrightarrow} (m, s')$*
2. *the original EFSM gets stuck from (n, s) avoiding $\mathcal{L}$*
3. *the original EFMS loops from (n, s) avoiding $\mathcal{L}$.*

Proof. Consider the following iterative algorithm, incrementally constructing n_j, s_j, E_j for $j \geq 0$. With $n_0 = n$, $s_0 = s$ and $E_0 = \varepsilon$, the invariant is that

$$1 \vdash_{\mathrm{ni}} (n, s) \stackrel{E_j}{\Longrightarrow} (n_j, s_j) \text{ with } n_0..n_{j-1} \text{ not the source of a transition in } \mathcal{L}.$$

This trivially holds for $j = 0$. For each iteration (with $j \geq 0$), there are 3 possible actions: if $n_j = m$ we exit the loop, and we have established case 1 with $E = E_j$ and $s' = s_j$.

Otherwise, if $n_j \neq m$, we can infer that n_j is not the source of a transition in $\mathcal{L}$, since if $n_j = \mathsf{S}(t)$ with $t \in \mathcal{L}$ then $n_j \in obs(n) = \{m\}$. There are now two possibilities:

– if there exists t with $\mathsf{S}(t) = n_j$ such that $[\![\mathsf{G}(t)]\!]s_j = \mathit{true}$, or such that $[\![\mathsf{G}_i(t)[c/v_b]]\!]s = \mathit{true}$ for some c if $\mathsf{E}_i(t)$ is of the form $e(v_b)$, we define t_{j+1} as one such t (say the "smallest"). Then there exists n_{j+1} and s_{j+1} and E'_j such that
$$1 \vdash t_j : (n_j, s_j) \stackrel{E'_j}{\to} (n_{j+1}, s_{j+1})$$
Let $E_{j+1} = E_j E'_j$. We then increment j by one and repeat the loop; the invariant will be maintained since $t \notin \mathcal{L}$ and n_j is not the source of a transition in $\mathcal{L}$.
– otherwise, we exit the loop, concluding that (n_j, s_j) is 1-stuck which establishes case 2.

If we never exit the loop, this will establish case 3.

Theorem 2 (Weak Soundness): Assume that $\mathcal{L}$ is closed under $\to_{\mathrm{dd}}$ and satisfies $\mathcal{WCC}$. If $2 \vdash^t_{\mathrm{ni}} C_2 \stackrel{E_2}{\Rightarrow\!\!\to} C'_2$ and $C_1\ Q\ C_2$ then there are 3 possibilities:

1. $1 \vdash^t_{\mathrm{ni}} C_1 \stackrel{E_1 E_2}{\Rightarrow\!\!\to} C'_1$ for some C'_1, E_1 with $C'_1\ Q\ C'_2$
   (and if $\mathcal{L}$ is uniform then $\mathsf{filter}(E_1) = \varepsilon$), or
2. the original EFSM gets stuck from C_1 avoiding $\mathcal{L}$ (cf. Definition 15), or
3. the original EFSM loops from C_1 avoiding $\mathcal{L}$ (cf. Definition 16).

Proof. Let $C_2 = (n, s_2)$ and $C'_2 = (n', s'_2)$ and $C_1 = (n, s_1)$. From Definition 9 and Fact 8 we see that $t \in \mathcal{L}$ and that for some m: $2 \vdash_{\mathrm{ni}} (n, s_2) \stackrel{\varepsilon}{\Longrightarrow} (m, s_2)$ and $2 \vdash t : (m, s_2) \stackrel{E_2}{\to} (n', s'_2)$. Thus $m \in \mathit{obs}(n)$, and from the $\mathcal{WCC}$ property we infer $\mathit{obs}(n) = \{m\}$. Lemma 35 now tells us that either

1. there exists E_1 and s''_1 such that $1 \vdash_{\mathrm{ni}} (n, s_1) \stackrel{E_1}{\Longrightarrow} (m, s''_1)$, or
2. the original EFSM gets stuck from (n, s_1) avoiding $\mathcal{L}$, or
3. the original EFSM loops from (n, s_1) avoiding $\mathcal{L}$.

If case 2 or case 3 holds, we are done. We thus assume that case 1 holds (and from Fact 11 we see that if $\mathcal{L}$ is uniform then $\mathsf{filter}(E_1) = \varepsilon$).

By assumption, we have $s_1(v) = s_2(v)$ for all $v \in \mathit{Rv}(n)$. By repeated application of Lemma 31, we infer that $\mathit{Rv}(m) \subseteq \mathit{Rv}(n)$ and that $s''_1(v) = s_1(v)$ for all $v \in \mathit{Rv}(m)$. Thus $s''_1(v) = s_2(v)$ for all $v \in \mathit{Rv}(m)$, which establishes
$$(m, s''_1)\ Q\ (m, s_2).$$
From $t \in \mathcal{L}$ we have $\mathsf{G}_1(t) = \mathsf{G}_2(t) = \mathsf{G}(t)$, and from $2 \vdash t : (m, s_2) \stackrel{E_2}{\to} (n', s'_2)$ we know that $[\![g]\!]s_2 = \mathit{true}$ where $g = \mathsf{G}(t)[c/v_b]$ if E_2 is of the form $e(c)$, and $g = \mathsf{G}(t)$ otherwise. For an arbitrary $w \in \mathit{fv}(\mathsf{G}_1(t)) \setminus \{v_b\}$ we infer by Fact 13 that $w \in \mathit{Rv}(t) \subseteq \mathit{Rv}(m)$, implying $s''_1(w) = s_2(w)$. Hence also $[\![g]\!]s''_1 = \mathit{true}$, implying that there exists s'_1 such that $1 \vdash t : (m, s''_1) \stackrel{E_2}{\to} (n', s'_1)$, and thus with $C'_1 = (n', s'_1)$ we have $1 \vdash^t_{\mathrm{ni}} C_1 \stackrel{E_1 E_2}{\Rightarrow\!\!\to} C'_1$. That $C'_1\ Q\ C'_2$, that is $s'_1(v) = s'_2(v)$ for all $v \in \mathit{Rv}(n')$, now follows from Lemma 32.

If $\mathcal{L}$ satisfies not just $\mathcal{WCC}$ but also $\mathcal{SCC}$, we can rule out case 3, which establishes

Theorem 3 (Strong Soundness): Assume $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}}$ and satisfies $\mathcal{SCC}$.

If $2 \vdash^t_{\mathrm{ni}} C_2 \stackrel{E_2}{\Rightarrow\!\!\rightarrow} C_2'$ and $C_1\ Q\ C_2$ then there are 2 possibilities:

1. $1 \vdash^t_{\mathrm{ni}} C_1 \stackrel{E_1 E_2}{\Rightarrow\!\!\rightarrow} C_1'$ for some C_1', E_1 with $C_1'\ Q\ C_2'$ (and if $\mathcal{L}$ is uniform then $\mathsf{filter}(E_1) = \varepsilon$), or
2. the original EFSM gets stuck from C_1 avoiding $\mathcal{L}$.

B Proofs of Correctness for Algorithms Computing Least Slices

Theorem 5 (WCC Algorithm Correctness): Assuming that the table `DD` correctly computes data dependence, Algorithm 1 returns in `L` the least superset of $\mathcal{L}$ that is closed under data dependence and satisfies $\mathcal{WCC}$.

Proof. We shall first state a number of loop invariants for the inner loop; they are expressed in terms of k, the number of iterations so far. We shall use C^k to denote the value of `C` after k iterations; similarly we shall write V^k and X^k (it is convenient to let $V^{-1} = \emptyset$). For $k \geq 0$ and each state n we define $obs^k_L(n)$ as the set of states n' such that there exists t in `L` with $\mathsf{S}(t) = n'$, and there exists a cycle-free path of length $\leq k$ outside `L` from n to n'. We observe that $n' \in obs^0_L(n)$ iff $n' = n$ and n in `B`, that $obs^k_L(n) \subseteq obs^{k'}_L(n)$ if $k \leq k'$, and that $n' \in obs_L(n)$ iff there exists $k \geq 0$ such that $n' \in obs^k_L(n)$.

The **while** loop invariants are, with $k \geq 0$:

1. $V^k = \{n \mid X^k[n] \text{ is defined } \}$
2. V^k is the disjoint union of V^{k-1} and C^k
3. for all $n \in V^k$, $X^k[n] \in obs^k_L(n)$
4. for all n, if $obs^k_L(n) \neq \emptyset$ then $n \in V^k$
5. if $\texttt{Lnew}^k = \emptyset$ then $obs^k_L(n) = \{X^k[n]\}$ for all $n \in V^k$
6. if L' satisfies $\mathcal{WCC}$ with $\texttt{L} \subseteq L'$ then $\texttt{Lnew} \subseteq L'$

It is easy to see that all invariants hold at loop entry, with $k = 0$. We shall now argue that the invariants are maintained by the body of the while loop; in particular, that they hold for $k+1$ where we can assume that they hold for k.

1. This follows easily from inspecting the code.
2. This follows easily from inspecting the code.
3. The claim is obvious if $X^{k+1}[n] = X^k[n]$ as then $X^k[n] \in obs^k_L(n)$ (as invariant 3 holds before the iteration) and thus $X^{k+1}[n] \in obs^{k+1}_L(n)$.
   Next consider the situation where $X^{k+1}[n]$ assumes the value of $X^k[m]$ because there exists $t \notin \texttt{L}$ with $\mathsf{T}(t) = m$ and $\mathsf{S}(t) = n$ where $m \in C^k$. Since invariant 3 holds before the iteration, we know that $X^k[m] \in obs^k_L(m)$. As it is easy to see (as $t \notin \texttt{L}$) that $obs^k_L(m) \subseteq obs^{k+1}_L(n)$, this yields the desired $X^{k+1}[n] \in obs^{k+1}_L(n)$.

4. We assume that $obs_L^{k+1}(n) \neq \emptyset$. If also $obs_L^k(n) \neq \emptyset$, we know (since invariant 4 holds) that $n \in V^k$ and thus $n \in V^{k+1}$.

   We can thus assume that $obs_L^k(n) = \emptyset$, and infer that there exists $t \notin \mathtt{L}$ with $\mathsf{S}(t) = n$ such that with $m = \mathsf{T}(t)$ it holds that $obs_L^k(m)$ is non-empty, but $obs_L^{k-1}(m) = \emptyset$. From invariants 4, 1, 3 and 2 we infer $m \in V^k$ and even $m \in C^k$. But then the iteration will add n to $\mathtt{V}$ so that $n \in V^{k+1}$.

5. We assume that `Lnew` remains empty; since we have already established invariant 3, our task is to prove that if $n' \in obs_L^{k+1}(n)$ then $n' = X^{k+1}[n]$. If $n' \in obs_L^k(n)$, we know (from invariants 4 and 5) that $n' = X^k[n]$ and thus also $n' = X^{k+1}(n)$.

   So assume that $n' \notin obs_L^k(n)$. The situation is that there exists $t \notin \mathtt{L}$ with $n = \mathsf{S}(t)$ such that with $m = \mathsf{T}(t)$ we have $n' \in obs_L^k(m)$ but $n' \notin obs_L^{k-1}(m)$. From $n' \in obs_L^k(m)$, and `Lnew` being empty before the iteration (as otherwise the loop would exit), we see (as invariants 4 and 5 hold) that $m \in V^k$ and $X^k[m] = n'$ and $obs_L^k(m) = \{n'\}$. From $n' \notin obs_L^{k-1}(m)$ we infer that $obs_L^{k-1}(m) = \emptyset$, and thus (from invariants 3 and 2) we have $m \in C^k$. Thus t is considered during the iteration, and since `Lnew` remains empty, we infer that $X^{k+1}[n] = n'$ (as $X[n]$ will be assigned either due to t, or some other transition being considered before t).

6. Let t be a member of $\mathtt{Lnew}^{k+1}$. With $n = \mathsf{S}(t)$ and $m = \mathsf{T}(t)$, we have $n \in V^{k+1}$ and $m \in C^k$, and there exists n' and m' with $n' \neq m'$ such that $n' = X^{k+1}[n]$ and $m' = X^k[m]$. From invariant 3 we see that $n' \in obs_L^{k+1}(n)$ and that $m' \in obs_L^k(m)$ which implies $m' \in obs_L^{k+1}(n)$. There thus exists a cycle-free path π_1 from n to n', and a cycle-free path π_2 from n to m'.

   These paths can have nothing in common. For assume, to get a contradition, that some node n'' occurs on both paths. Then there exists j with $j \leq k$ such that $obs_L^j(n'')$ is not a singleton. From invariant 5 we infer that $\mathtt{Lnew}^j$ is non-empty, but then the loop would exit after j iterations, which is a contradiction.

   Now assume that L' contains $\mathtt{L}$ and satisfies $\mathcal{WCC}$. In particular, $obs_{L'}(n)$ has to be at most a singleton. But since $obs_{L'}(n)$ has to contain a node from π_1, and also a node from π_2, this is only possible if $obs_{L'}(n) = \{n\}$. But this can only happen if $t \in L'$.

After having stated and proved the invariants for the **while** loop, let us now address the invariant for the **repeat** loop. It obviously holds initially. We shall now argue it is maintained by an iteration, which is obvious for the part about being closed under data dependence (due to the call of `DDclose` at the end of the body).

Now let L' be a superset of $\mathcal{L}$ that is closed under data dependence and satisfies $\mathcal{WCC}$. Our goal is to show that after an iteration, $\mathtt{L} \subseteq L'$. From the invariant, we know that $\mathtt{L} \subseteq L'$ holds before the iteration. From invariant 6 for the **while** loop, we see that $\mathtt{Lnew} \subseteq L'$. By the specification of `DDclose`, we infer that $\mathtt{DDclose(L,Lnew)} \subseteq L'$. But this is what we aimed to prove.

We are left with proving that L does indeed satisfy $\mathcal{WCC}$ when the **repeat** loop exits, with $\texttt{Lnew} = \emptyset$. Let the last iteration have k iterations of the **while** loop; thus $C^k = \emptyset$ but $C^j \neq \emptyset$ for $j < k$.

Now assume, to get a contradiction, that L does *not* satisfy $\mathcal{WCC}$.

There thus exists i and m such that $obs^i_L(m)$ has at least two elements. We can assume that i is the least such number, that is: for $j < i$ and all n, $obs^j_L(n)$ is at most a singleton. We infer that there exists n' such that $n' \in obs^i_L(m)$, and that there exists m' with $m' \neq n'$ such that $m' \in obs^i_L(m)$ but $m' \notin obs^{i-1}_L(m)$. There thus exists $m_0..m_i$ with $m_i = m$ and $m_0 = m'$, and a cycle-free path through $m_i...m_0$ from m to m', such that for all $j \in 0 \ldots i$ it holds that $m' \in obs^j_L(m_j)$ but $m' \notin obs^{j-1}_L(m_j)$. For $j < i$ we know, as $obs^j_L(m_j)$ is at most a singleton, that $obs^j_L(m_j) = \{m'\}$, and thus $obs^{j-1}_L(m_j) = \emptyset$ from which we infer that $m_j \in V^j \setminus V^{j-1} = C^j$. Since $C^j \neq \emptyset$ for $j < i$, and $C^k = \emptyset$, we infer that $i \leq k$. Thus $obs^k_L(m)$ has at least two elements. But this conflicts with invariant 5 for the **while** loop, giving the desired contradiction.

Theorem 7 (SCC Algorithm Correctness): Assuming that the table DD correctly computes data dependence, Algorithm 2 (using Algorithm 3) returns in L the least superset of $\mathcal{L}$ that is closed under data dependence and satisfies $\mathcal{SCC}$.

Proof. The proof is quite similar to the proof of Theorem 5; we shall only list the features to be added.

First, we need to add the following invariant for the **while** loop:

$$\text{If } \texttt{Lnew}^k = \emptyset \text{ and } n' \in X^k[n] \text{ then no infinite path from } n \text{ avoids } n' \qquad (1)$$

To prove this, first observe that if $n = n'$, and hence when $k = 0$, the claim is trivial. We may thus assume that $k > 0$ and $n' \neq n$. By invariant 3 we have $n' \in obs^k_L(n)$. Thus there exists $t \notin \texttt{L}$ with $n = \mathsf{S}(t)$ such that with $m = \mathsf{T}(t)$ we have $n' \in obs^{k-1}_L(m)$ and by invariant 4 thus $m \in V^{k-1}$. We infer that $m \in C^j$ for some $j \leq k-1$ and hence the $(j+1)$th iteration will examine t. Since $\texttt{Lnew}^{j+1} = \emptyset$, we infer that $\texttt{LoopAvoids}(n,n')$ does not hold, and hence (by Fact 24) there is no infinite path from n that avoids n'.

Also, we need to modify Invariant 6 for the **while** loop into:

$$\text{if } L' \text{ satisfies } \mathcal{SCC} \text{ with } \texttt{L} \subseteq L' \text{ then } \texttt{Lnew} \subseteq L' \qquad (2)$$

To see that this invariant is maintained, assume that L' contains L and satisfies $\mathcal{SCC}$ (and thus also $\mathcal{WCC}$), and that t is a member of $\texttt{Lnew}^{k+1}$; we must show that $t \in L'$.

If t was added by line 17 of Algorithm 2, we can reason as in the proof of Theorem 7 to show that $t \in L'$.

We can therefore assume that t was added by Line 14. With $n = \mathsf{S}(t)$ and $m = \mathsf{T}(t)$ and $m' = X^k[m]$, so that there is a cycle-free path π outside L from n to m' that starts with t, there thus is a infinite path from n that avoids m'.

Assume, to get a contradiction, that $t \notin L'$. Then $obs_{L'}(n)$ (which obviously cannot be empty) would consist of a node in π different from n, say n'. There exists $i \leq k$ such that $m' = X^i[n']$, and as $\texttt{Lnew}^i = \emptyset$ we see by (1) that no infinite path from n' avoids m'. But then there is an infinite path from n that avoids n', as otherwise no infinite path from n would avoid m'. Since $obs_{L'}(n) = \{n'\}$, this contradicts L' satisfying $\mathcal{SCC}$. We conclude that $t \in L'$, as desired.

Having proved that (2) and (1) are indeed extra invariants for the **while** loop, we can now proceed as in the proof of Theorem 5 to show the correctness of the invariant listed in Algorithm 2 for the **repeat** loop.

We are left with proving that L does indeed satisfy $\mathcal{SCC}$ when the **repeat** loop exits, with $\texttt{Lnew} = \emptyset$. Let the last iteration have k iterations of the **while** loop; thus $C^k = \emptyset$ but $C^j \neq \emptyset$ for $j < k$.

Now assume, to get a contradiction, that L does *not* satisfy $\mathcal{SCC}$. There can be two reasons for this: if L does not satisfy $\mathcal{WCC}$, we can show as in the proof of Theorem 5 that this conflicts with invariant 5 for the **while** loop.

We can thus assume that L satisfies $\mathcal{WCC}$ but *not* $\mathcal{SCC}$. Then there exists n and n' such that $n' \in obs_L(n)$, and yet there is an infinite path from n that avoids n'. Let i be the smallest number such that $n' \in obs^i_L(n)$. There thus exists $n_0..n_i$ with $n_i = n$ and $n_0 = n'$, and a cycle-free path through $n_i...n_0$ from n to n', such that for all $j \in 0 \ldots i$ it holds that $n' \in obs^j_L(n_j)$ but $n' \notin obs^{j-1}_L(n_j)$. For $j \leq i$ we know, as $obs^j_L(n_j)$ is at most a singleton, that $obs^j_L(n_j) = \{n'\}$, and thus $obs^{j-1}_L(n_j) = \emptyset$ from which we infer that $n_j \in V^j \setminus V^{j-1} = C^j$. Since $C^j \neq \emptyset$ for $j \leq i$, and $C^k = \emptyset$, we infer that $i < k$. Thus $n' \in obs^k_L(n)$, and hence (by invariant 5 for the **while** loop) $n' = X^k[n]$. But then an infinite path from n that avoids n' conflicts with (1), giving the desired contradiction.