Correctness of Slicing Finite State Machines

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

[1]Department of Computing and Information Sciences,
Kansas State University, Manhattan KS 66506, USA

[2]Department of Computer Science,
Middlesex University, London, NW4 4BT, United Kingdom.

[3]CREST, Department of Computer Science,
University College London, United Kingdom.

December 29, 2013

Abstract

We consider slicing of extended finite state machines. These may be non-deterministic and hence standard techniques for slicing, developed for control flow graphs, are not immediately applicable.

In this paper we propose ways of expressing the semantic correctness of slicing. We shall demand that the sliced machine simulates the original machine, in that each "observable" step by the latter can also be done by the former. In the other direction, we cannot hope for a perfect simulation, but demand that for each observable step by the sliced machine, either the original machine simulates it or *(i)* it gets stuck, or *(ii)* it loops.

To ensure correctness, 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 "weak commitment" property highlighted by Danicic et al.

If the slice also has the "strong commitment" property, the case *(ii)* above can be ruled out, meaning that the original machine will simulate the sliced machine except that it may get stuck.

We prove that for each of the properties "weak commitment" and "strong commitment", there exists a least set with that property; we also give algorithms to compute that least set.

1 Introduction

The goal of program slicing [10, 9] is to remove the parts of a program that are irrelevant in a given context. Slicing has been applied for many purposes: compiler optimizations, debugging, model checking, protocol understanding, etc. Assuming that

the program is represented as a control flow graph (CFG), and that the *slicing criterion* (the set of "relevant" nodes) is given, slicing involves the following two steps:

1. compute the *slice*, a set of nodes which includes the slicing criterion as well as those nodes that the slicing criterion depends on (directly or indirectly, wrt. data or wrt. control);

2. create the *sliced program*, essentially by removing the nodes that are not in the slice and doing suitable "rewiring".

Correctness of slicing may be phrased in various ways but will typically involve the following properties:

1. The sliced program is well-defined; in particular, if a test that determines branching is removed then the two branches can be combined into one.

2. If the original program can do some observable action then also the sliced program can do that action; here an observable action may be defined either as one that is part of the slicing criterion, or as one that is part of the slice.

3. If the sliced program can do some observable action then also the original program can do that action.

The last property (3) is often considered optional since in order to keep the size of the slices managable one will allow to slice away loops.

Early work on the theoretical foundation of slicing [2, 5] is based on two underlying assumptions: that the CFG is deterministic, and that it has a unique end node. Then correctness properties (1) and (2) can be obtained by requiring the slice to be closed under *data dependency* and under a carefully defined notion of *control dependency*. In order also to have correctness property (3), one needs to employ a stronger version of control dependency, ironically called "weak control dependence" [6] as it will include *more* nodes in the slice.

An early attempt to handle non-determinism is [4] which considers a multi-threaded language and proposes several kinds of dependencies, and also proposes to phrase correctness in terms of bisimulation (as is known from concurrency) but does not prove any correctness results.

More recent work on slicing, while still requiring determinism, has allowed CFGs with no end nodes so as to help model indefinitely running reactive systems. In [7, 8] the authors propose a notion of control dependency that does not assume a unique end node (but if so will coincide with the standard notion), and establish the correctness properties (1)–(3) by means of a (weak) bisimulation result. That proof only works for *reducible* CFGs, however; to ensure correctness for also *irreducible* CFGs it is necessary [8] to introduce an additional kind of dependency that the slice must be closed under. On the other hand, if one doesn't want to enforce correctness property (3), the dependency to use (in addition to data dependence) is "weak order dependence" as introduced in [1].

A reader may by now feel perplexed by the numerous variants of control dependency, but [3] has managed to extract order out of chaos: rather than looking at when

one node depends on another node, the authors pinpoint two desirable properties of the resulting *slices*:

- *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.

[3] goes on to show that WCC is what is required (as already recognized in [1]) to get correctness properties (1) and (2), while SCC is what is required to also get correctness property (3). Furthermore, [3] shows that previous approaches from the literature will compute either a slice satisfying WCC (if loops may be sliced away) or a slice satisfying SCC (if loops may not be sliced away).

In this paper, we shall show that the notions of WCC and SCC are very helpful to rather smoothly extend slicing to a non-deterministic setting. To make the development concrete we shall phrase it for Extended Finite State Machines (EFSM). In this setting, it appears natural to let slices be sets of transitions rather than sets of nodes (states). After having defined the semantics such that correctness property (1) comes "for free", and having defined what constitutes an "observable", we can go on to prove the desired correctness properties. But while (2) is easy to prove, it is not as informative as in a deterministic setting, since it doesn't exclude the possibility that the sliced machine have choices not present for the original machine. We would therefore want to modify (3) so as to state that non-determinism does not increase, that is, if the sliced machine can do so observable action then also the original machine can — unless one of two happens: *(i)* the original machine loops, but that can be excluded by demanding that the slice satisfies SCC; *(ii)* the original machine gets stuck (which is a possibility since the non-determinism may allow not just more than one enabled transition but also zero enabled transitions).

This paper is organized as follows: In Sect. 2 we shall introduce EFSMs, in Sect. 3 their semantics, in Sect. 4 some basic properties of slices, and in Sect. 5 some properties of data flow. In Sect. 6 we shall set up the machinery for expressing correctness, with property (2) being proved in Sect. 7 and property (3) being proved in Sect. 8. In Sect. 9 we shall address how to find the least set that satisfies WCC, and in Sect. 10 how to find the least set that satisfies SCC.

2 Extended Finite State Machines (EFSMs)

An Extended Finite State Machine (EFSM) M is a tuple (S, T, E, V) where S is a finite set of states, T is a finite set of transitions, E is a finite set of events, and V is a finite set of variables. A state $n \in S$ (we shall also use the letter m to range over states) is considered atomic, and one state in S may be designated as the initial state. A transition $t \in T$ (we shall also use the letter u to range over transitions) has a source state $\mathsf{S}(t) \in S$, a target state $\mathsf{T}(t) \in S$, and a label of the form $e[g]/a$. Here $e \in E$ is an event, g is a guard, and a is an action which we wlog. can assume is an assignment to a variable in V; all parts of a label are optional (an omitted guard is the same as the guard `true`; an omitted action is the same as the action `skip`).

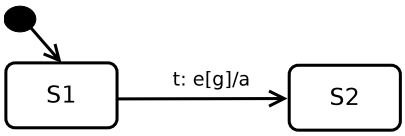

Figure 1: A simple extended finite state machine.

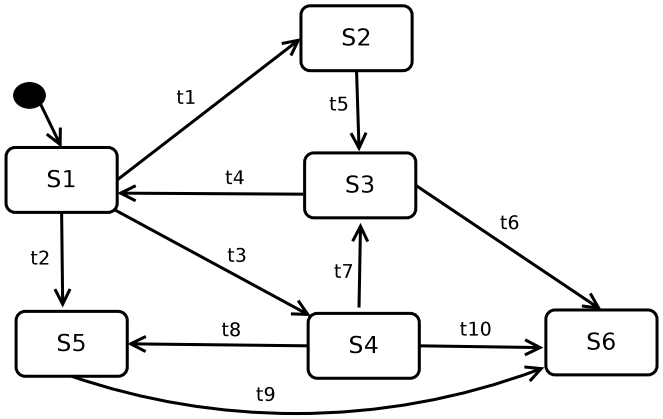

Figure 2: Our running example EFSM.

Figure 1 shows a graphical representation of a simple EFSM; while Figure 2 depicts a non-trivial EFSM that shall serve as our running example in this paper. In both cases, `S1` is supposed to be the initial state.

Some terminology about a transition t: it is a *self-looping transition* if $\mathsf{S}(t) = \mathsf{T}(t)$; it is an *$\varepsilon$-transition* if all parts of its label are empty; it is a *final transition* if $\mathsf{T}(t) = n$ with n an *exit* state, that is $n = \mathsf{S}(u)$ for no transition u.

Some terminology about a pair of transitions t and u: if $\mathsf{S}(u) = \mathsf{T}(t)$ then u is a *successor* of t; if $\mathsf{S}(t) = \mathsf{S}(u)$ and $\mathsf{T}(t) = \mathsf{T}(u)$ then t and u are *parallel transitions*; if $\mathsf{S}(t) = \mathsf{S}(u)$ but $\mathsf{T}(t) \neq \mathsf{T}(u)$ then t and u are *siblings*.

3 Semantics of EFSMs

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

We shall present a semantics that facilitates reasoning about slicing. With a slice set being a set of transitions, our development is relative to a fixed slice set $\mathcal{L}$. Slicing

amounts to keeping the transitions in $\mathcal{L}$ as they are, whereas transitions not in $\mathcal{L}$ are replaced by ε-transitions. In our definitions, we shall use the subscript 1 to refer to the original machine, and subscript 2 to refer to the sliced machine.

Hence the guard of a transition t as denoted $\mathsf{G}_1(t)$ which is a boolean expression, and its "enabling events" is denoted $\mathsf{E}_1(t)$ which is either a singleton $[e]$ (the transition consumes e) or $[\,]$ (the transition is "spontaneous"). If $t \in \mathcal{L}$ we have $\mathsf{G}_2(t) = \mathsf{G}_1(t)$ and $\mathsf{E}_2(t) = \mathsf{E}_1(t)$, otherwise $\mathsf{G}_2(t) = \mathit{true}$ and $\mathsf{E}_2(t) = [\,]$.

If enabled, t modifies the variables in $\mathsf{D}_1(t)$ which we shall assume is either the empty set or a singleton; in the latter case, with $\mathsf{D}_1(t) = \{v\}$, we shall write $\mathsf{A}_1(t)$ for the expression being assigned to v. If $t \in \mathcal{L}$ we have $\mathsf{D}_2(t) = \mathsf{D}_1(t)$ and $\mathsf{A}_2(t) = \mathsf{A}_1(t)$, otherwise $\mathsf{D}_2(t) = \emptyset$ and $\mathsf{A}_2(t)$ is undefined. For $i = 1, 2$ we define $\mathsf{U}_i(t)$, the variables used by t, as $\mathit{fv}(\mathsf{G}_i(t)) \cup \mathit{fv}(\mathsf{A}_i(t))$. Observe that we do not allow for transitions that produce events.

Definition 1 *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 evolves to (n', s') while consuming the events in the list E (which is $[\,]$ or a singleton). This happens when t is such that $\mathsf{S}(t) = n$, $\mathsf{T}(t) = n'$, $[\![\mathsf{G}_i(t)]\!]s = \mathit{true}$, $E = \mathsf{E}_i(t)$, and either $\mathsf{D}_i(t) = \emptyset$ and $s' = s$ or $\mathsf{D}_i(t)$ is a singleton $\{v\}$ and $s' = s[v \mapsto [\![\mathsf{A}_i(t)]\!]s]$.*

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

We say that $[t_1..t_k]$ ($k \geq 0$) is a path 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 $n = \mathsf{T}(t_j)$. We say that a path $[t_1..t_k]$ is outside $\mathcal{L}$ iff $t_j \notin \mathcal{L}$ for all $j \in 1 \ldots k$.

We say that $[t_1..t_k]$ is cycle-free iff for all $i, j \in 1 \ldots k$, $\mathsf{S}(t_i) = \mathsf{T}(t_j)$ implies $i = j + 1$. We shall often use the following observation: if there exists a path from n to m then there also exists a cycle-free path from n to m.

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 $j \geq 1$.

We say that (n, s) is i-stuck if $[\![\mathsf{G}_i(t)]\!]s = \mathit{false}$ for all t with $\mathsf{S}(t) = n$. We say that (n, s) gets i-stuck avoiding m if for some $k \geq 0$ there exists $n_1..n_k$, $s_1..s_k$, $t_1..t_k$, $E_1..E_k$ such that with $n_0 = n$ and $s_0 = s$ it holds that (n_k, s_k) is i-stuck with $n_k \neq m$, and for all $j \in 1 \ldots k$ we have $i \vdash t_j : (n_{j-1}, s_{j-1}) \xrightarrow{E_j} (n_j, s_j)$ and $n_{j-1} \neq m$.

We say that (n, s) i-loops avoiding m if for all $j \geq 1$ there exists n_j, s_j, t_j, E_j such that (with $n_0 = n$ and $s_0 = s$) $i \vdash t_j : (n_{j-1}, s_{j-1}) \xrightarrow{E_j} (n_j, s_j)$ and $n_{j-1} \neq m$.

4 Properties of Slice Sets

We first introduce the standard notion of data dependence:

Definition 3 *We say that t' is data dependent on t (in the i-semantics), written $t \rightarrow_{\mathrm{dd}i} t'$, iff there exists a variable $v \in \mathsf{D}_i(t) \cap \mathsf{U}_i(t')$ and a path $[t_1..t_k]$ ($k \geq 0$) from $\mathsf{T}(t)$ to $\mathsf{S}(t')$ such that for all $j \in 1 \ldots k$, $v \notin \mathsf{D}_i(t_j)$.*

Definition 4 *We say that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}i}$ iff $t \in \mathcal{L}$ whenever $t \rightarrow_{\mathrm{dd}i} t'$ and $t' \in \mathcal{L}$.*

The following result is not needed for our development, but it is reassuring that it holds.

Lemma 5 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}1}$. Then also $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}2}$.*

Proof: Assume not. Then there exists $t' \in \mathcal{L}$ and $t \notin \mathcal{L}$ such that $t \rightarrow_{\mathrm{dd}2} t'$. There thus exists a variable $v \in \mathsf{D}_2(t) \cap \mathsf{U}_2(t')$ and a path $[t_1..t_k]$ from $\mathsf{T}(t)$ to $\mathsf{S}(t')$ such that for all $j \in 1 \ldots k$, $v \notin \mathsf{D}_2(t_j)$. From the way the 2-semantics is constructed from the 1-semantics, we see that $v \in \mathsf{D}_1(t) \cap \mathsf{U}_1(t')$. If for all $j \in 1 \ldots k$ it holds that $v \notin \mathsf{D}_1(t_j)$ then $t \rightarrow_{\mathrm{dd}1} t'$ which contradicts that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd}1}$. Therefore there exists $j \in 1 \ldots k$ with $v \in \mathsf{D}_1(t_j)$; choose the largest such j. Thus $t_j \rightarrow_{\mathrm{dd}1} t'$. From $\mathcal{L}$ being closed under $\rightarrow_{\mathrm{dd}1}$ we infer $t_j \in \mathcal{L}$. But then $\mathsf{D}_1(t_j) = \mathsf{D}_2(t_j)$, contradicting $v \in \mathsf{D}_1(t_j)$ and $v \notin \mathsf{D}_2(t_j)$. ∎

Next we formalize the notion of "next observable state":

Definition 6 *For a slice set $\mathcal{L}$, for each state n we define $obs_{\mathcal{L}}(n)$ 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, let us consider the EFSM in Fig. 2. If $\mathcal{L} = \{t5, t6\}$ then $obs_{\mathcal{L}}(S1) = \{S2, S3\}$ whereas $obs_{\mathcal{L}}(S2) = \{S2\}$ and (since $[t4, t1]$ is a path from $S3$ to $S2$ outside $\mathcal{L}$) $obs_{\mathcal{L}}(S3) = \{S2, S3\}$.

We shall write $obs(n)$ for $obs_{\mathcal{L}}(n)$ when $\mathcal{L}$ is given by the context.

Lemma 7 *If $m \in obs(n)$ then $m \in obs(m)$.*

Proof: From $m \in obs(n)$ we see that there exists $t \in \mathcal{L}$ with $\mathsf{S}(t) = m$. But this shows $m \in obs(m)$ as we can use $k = 0$ in Definition 6. ∎

We can now define the notions of "weak commitment closed" ($\mathcal{WCC}$) and "strong commitment closed" ($\mathcal{SCC}$):

Definition 8 *We say that $\mathcal{L}$ satisfies $\mathcal{WCC}$ iff for each state n, $obs_{\mathcal{L}}(n)$ is either empty or a singleton.*

To motivate why $\mathcal{WCC}$ is a desirable property, let us again look at Fig. 2 with $\mathcal{L} = \{t5, t6\}$ so that $obs_{\mathcal{L}}(S1) = \{S2, S3\}$. Then the sliced machine may move to $S3$ and perform the observable transition $t6$, while such a move may be impossible for the original machine (for example if $\mathsf{G}_1(t3)$ is false) which could instead move to $S2$ (if $\mathsf{G}_1(t1)$ is true) and perform $t5$. This would thus conflict with our goal that for each observable step by the sliced machine, either the original machine simulates it or *(i)* it gets stuck, or *(ii)* it loops.

To rule out (ii), we need a stronger property:

Definition 9 *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 does not exist an infinite path from* n *that avoids* n'*.*

Clearly a slice set that satisfies $\mathcal{SCC}$ will also satisfy $\mathcal{WCC}$. To motivate why $\mathcal{SCC}$ is a desirable property, let us again look at Fig. 2, this time with $\mathcal{L} = \{t9\}$ so that $obs_{\mathcal{L}}(S1) = \{S5\}$ but there is an infinite path from $S1$ that avoids $S5$. Thus the sliced machine may move to $S5$ and perform the observable transition $t9$, while such a move may be impossible for the original machine (if $\mathsf{G}_1(t2)$ is false) which could instead (for certain values of the store) cycle infinitely between $S1$, $S2$, and $S3$.

5 Relevant Variables

In the following, we as usual assume a given slice set $\mathcal{L}$.

Definition 10 *We say that* v *is relevant for* t *(for the* i*-semantics), written* $v \in Rv_i(t)$*, iff there exists* $t' \in \mathcal{L}$ *such that* $v \in \mathsf{U}_i(t')$*, 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* $j \in 1 \ldots k-1$*,* $v \notin \mathsf{D}_i(t_j)$*.*

Fact 11 *If* $t \in \mathcal{L}$ *then* $v \in Rv_i(t)$ *for all* $v \in \mathsf{U}_i(t)$*.*

For a state n we define

$$Rv_i(n) = \bigcup_{\mathsf{S}(t)=n} Rv_i(t)$$

Observe that $v \in Rv_i(n)$ iff there exists a path $[t_1..t_k]$ with $k \geq 1$ from n such that $t_k \in \mathcal{L}$ with $v \in \mathsf{U}_i(t_k)$, and for all $j \in 1 \ldots k-1$ it holds that $v \notin \mathsf{D}_i(t_j)$.

Lemma 12 *Assume that for* $t \in \mathcal{L}$ *we have*

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

where $s_1(v) = s_2(v)$ *for all* $v \in Rv_1(n)$*. Then* $s_1'(v) = s_2'(v)$ *for all* $v \in Rv_1(n')$*.*

Proof: First observe that $\mathsf{D}_2(t) = \mathsf{D}_1(t)$, and $\mathsf{A}_2(t) = \mathsf{A}_1(t)$. Let $v \in Rv_1(n')$ be given, so as to show $s_1'(v) = s_2'(v)$. We split into two cases.

The first case is when $v \in \mathsf{D}_1(t)$, where we must prove that $[\![\mathsf{A}_1(t)]\!]s_1 = [\![\mathsf{A}_2(t)]\!]s_2$. But this follows since for all $w \in fv(\mathsf{A}_1(t))$ we have $w \in Rv_1(n)$ and thus $s_1(w) = s_2(w)$.

The other case is when $v \notin \mathsf{D}_1(t)$. Then $v \in Rv_1(n)$, implying the desired equality $s_1'(v) = s_1(v) = s_2(v) = s_2'(v)$. ∎

Lemma 13 *Assume that* $\mathcal{L}$ *is closed under* $\rightarrow_{\mathrm{dd1}}$*. Then* $Rv_1(t) = Rv_2(t)$ *for all* t*.*

Proof: First we consider $v \in Rv_1(t)$, so as to show $v \in Rv_2(t)$. There exists $t' \in \mathcal{L}$ with $v \in \mathsf{U}_1(t')$ and thus $v \in \mathsf{U}_2(t')$, and $t_1 \ldots t_k$ with $t = t_1$ and $t' = t_k$ such that for all $j \in 1 \ldots k-1$, $v \notin \mathsf{D}_1(t_j)$ and thus $v \notin \mathsf{D}_2(t_j)$. But this shows $v \in Rv_2(t)$.

Next we consider $v \in Rv_2(t)$ so as to show $v \in Rv_1(t)$. There exists $t' \in \mathcal{L}$ with $v \in \mathsf{U}_2(t')$ and thus $v \in \mathsf{U}_1(t')$, and $t_1 \ldots t_k$ with $t = t_1$ and $t' = t_k$ such that for all $j \in 1 \ldots k-1$, $v \notin \mathsf{D}_2(t_j)$. If $v \notin \mathsf{D}_1(t_j)$ for all $j \in 1 \ldots k-1$ then $v \in Rv_1(t)$ so we are left with the case where there exists $j \in 1 \ldots k-1$ such that $v \in \mathsf{D}_1(t_j)$; choose the largest such j. Then $t_j \rightarrow_{\mathrm{dd1}} t'$ and by assumption hence $t_j \in \mathcal{L}$. But then $\mathsf{D}_1(t_j) = \mathsf{D}_2(t_j)$, contradicting $v \in \mathsf{D}_1(t_j)$ and $v \notin \mathsf{D}_2(t_j)$. ∎

For $\mathcal{L}$ closed under $\rightarrow_{\mathrm{dd1}}$, the relevant variables of the sliced machine thus coincides with the relevant variables of the orginal machine. We may therefore write $Rv(t)$ for $Rv_1(t) = Rv_2(t)$, and $Rv(n)$ for $Rv_1(n) = Rv_2(n)$.

Lemma 14 *Let n be a state with $obs(n) = \emptyset$. Then $Rv_1(n) = \emptyset$.*

Proof: To get a contradiction, assume that $v \in Rv_1(n)$. That is, there exists a path $[t_1..t_k]$ ($k \geq 1$) from n with $t_k \in \mathcal{L}$ and $v \in \mathsf{U}_1(t_k)$ where for all $j \in 1 \ldots k-1$ we have $v \notin \mathsf{D}_1(t_j)$. There exists $q \in 1 \ldots k$ with $t_q \in \mathcal{L}$ but $t_j \notin \mathcal{L}$ for $j < q$. But then $[t_1..t_{q-1}]$ is a path from n to $\mathsf{S}(t_q)$ and thus $\mathsf{S}(t_q) \in obs(n)$, giving a contradiction. ∎

Lemma 15 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$. If $obs(n) = \{n'\}$ then $Rv(n) = Rv(n')$.*

Proof: Recall that by Lemma 13, for all n we have $Rv(n) = Rv_1(n) = Rv_2(n)$.

First consider $v \in Rv(n)$, so as to prove $v \in Rv(n')$. There exists a path $[t_1..t_k]$ with $k \geq 1$ from n such that $t_k \in \mathcal{L}$ with $v \in \mathsf{U}_1(t_k)$ and such that $v \notin \mathsf{D}_1(t_j)$ for all $j \in 1 \ldots k-1$. We can find q with $1 \leq q \leq k$ such that $t_q \in \mathcal{L}$ but $t_j \notin \mathcal{L}$ for $1 \leq j < q$. Since $obs(n)$ contains only n', we infer that $n' = \mathsf{S}(t_q)$. But hence $v \in Rv(n')$.

Next consider $v \in Rv(n')$, so as to prove $v \in Rv(n)$. There exists a path $[t_1..t_k]$ with $k \geq 1$ from n' such that $t_k \in \mathcal{L}$ with $v \in \mathsf{U}_1(t_k)$ and such that $v \notin \mathsf{D}_1(t_j)$ for all $j \in 1 \ldots k-1$. Also, there exists a path $[u_1..u_q]$ ($q \geq 0$) outside $\mathcal{L}$ from n to n'. This will establish $v \in Rv(n)$, provided that we can show for all $j \in 1 \ldots q$ that $v \notin \mathsf{D}_1(u_j)$. Assume, to get a contradiction, that there exists $j \in 1 \ldots q$ with $v \in \mathsf{D}_1(u_j)$; choose the largest such j. But then $u_j \rightarrow_{\mathrm{dd1}} t_k$ which contradicts $t_k \in \mathcal{L}$ and $u_j \notin \mathcal{L}$ since $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$. ∎

Lemma 16 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{WCC}$. If $obs(n) = obs(m)$ then $Rv(n) = Rv(m)$.*

Proof: Recall that since $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$, we can omit the subscript to Rv. From $\mathcal{L}$ satisfying $\mathcal{WCC}$ we see that there are two cases.

If $obs(n) = obs(m) = \emptyset$, we see from Lemma 14 that $Rv(n) = Rv(m) = \emptyset$.

If $obs(n) = obs(m)$ is a singleton, say $\{n'\}$, we infer from Lemma 15 that $Rv(n) = Rv(n') = Rv(m)$. ∎

Lemma 17 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{WCC}$. If for $i \in \{1,2\}$ we have $i \vdash t : (n,s) \stackrel{E}{\rightarrow} (n',s')$ with $t \notin \mathcal{L}$ and with $obs(n') \neq \emptyset$, then for all $v \in Rv(n)$ we have $s(v) = s'(v)$.*

Proof:. For $i = 2$, Fact 2 tells us that $s' = s$ yielding the claim. We can thus assume that $i = 1$.

From $t \notin \mathcal{L}$ we infer that $obs(n') \subseteq obs(n)$, and from $obs(n') \neq \emptyset$ and the $\mathcal{WCC}$ assumption there thus exists m such that $obs(n') = obs(n) = \{m\}$ and hence by Lemma 15 $Rv(n') = Rv(n)$.

Now consider $v \in Rv(n)$. Since $v \in Rv(n')$, there exists a path $[t_1..t_k]$ ($k \geq 1$) from n' such that $t_k \in \mathcal{L}$ with $v \in \mathsf{U}_1(t_k)$, and $v \notin \mathsf{D}_1(t_j)$ for $j \in 1 \ldots k-1$. Assume, to get a contradiction, that $v \in \mathsf{D}_1(t)$. Then $t \rightarrow_{\mathrm{dd1}} t_k$, which together with $t_k \in \mathcal{L}$ and $t \notin \mathcal{L}$ contradicts $\mathcal{L}$ being closed under $\rightarrow_{\mathrm{dd1}}$. Hence $v \notin \mathsf{D}_1(t)$, and thus $s'(v) = s(v)$. ∎

6 Correctness Criterion

We define a relation Q between configurations:

Definition 18 $(n_1,s_1)\ Q\ (n_2,s_2)$ *iff* $n_1 = n_2$ *and* $s_1(v) = s_2(v)$ *for all* $v \in Rv_1(n_1)$.

We now introduce a relation $i \vdash C \stackrel{E}{\Rightarrow} C'$, saying that C in the i-semantics evolves to C' through zero or more non-observable steps while consuming the events in E:

Definition 19 *We write $i \vdash C \stackrel{E}{\Rightarrow} C'$ iff for some $k \geq 1$ there exists $C_1 \ldots C_k$, with $C = C_1$ and $C' = C_k$, and $E_1 \ldots E_{k-1}$ with $E = E_1 \ldots E_{k-1}$, such that for all $j \in 1 \ldots k-1$ there exists $t_j \notin \mathcal{L}$ with $i \vdash t_j : C_j \stackrel{E_j}{\rightarrow} C_{j+1}$.*

Using Fact 2, we get:

Lemma 20 *If $2 \vdash (n,s) \stackrel{E}{\Rightarrow} (n',s')$ then $E = [\,]$ and $s' = s$.*

Finally, we define the reduction that forms the basic of simulation:

Definition 21 *We write $i \vdash^t C \stackrel{E}{\Rightarrow} C'$ if $t \in \mathcal{L}$ and there exists C_1, and E_1, E_2 with $E = E_1E_2$, such that $i \vdash C \stackrel{E_1}{\Rightarrow} C_1$ and $i \vdash t : C_1 \stackrel{E_2}{\rightarrow} C'$.*

The correctness results now state that the original and the sliced machines should simulate each other, as specified in the following.

7 Sliced Simulates Original

This is the easier direction. First a few auxiliary results.

Lemma 22 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{WCC}$. If with $t \notin \mathcal{L}$ and $obs(n') \neq \emptyset$ we have*

- $1 \vdash t : (n, s_1) \xrightarrow{E} (n', s_1')$ *and*
- $s_1(v) = s_2(v)$ *for all* $v \in \mathit{Rv}_1(n)$

then

- $2 \vdash t : (n, s_2) \xrightarrow{[]} (n', s_2)$ *and*
- $s_1'(v) = s_2(v)$ *for all* $v \in \mathit{Rv}_1(n')$.

Proof: Since $t \notin \mathcal{L}$, $\mathsf{G}_2(t) = \mathit{true}$ and $\mathsf{E}_2(t) = [\,]$ and $\mathsf{D}_2(t) = \emptyset$ which yields the first claim. For the second claim, consider $v \in \mathit{Rv}_1(n')$. Since $t \notin \mathcal{L}$, also $v \in \mathit{Rv}_1(n)$. By Lemma 17 we infer $s_1(v) = s_1'(v)$, and since $s_1(v) = s_2(v)$ holds by assumption, we can conclude $s_1'(v) = s_2(v)$. ∎

Lemma 23 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies WCC. If with $obs(n') \neq \emptyset$ we have*

- $1 \vdash (n, s_1) \xRightarrow{E} (n', s_1')$ *and*
- $s_1(v) = s_2(v)$ *for all* $v \in \mathit{Rv}_1(n)$

then

- $2 \vdash (n, s_2) \xRightarrow{[]} (n', s_2)$ *and*
- $s_1'(v) = s_2(v)$ *for all* $v \in \mathit{Rv}_1(n')$.

Proof: An easy induction in the "k" of Definition 19, using Lemma 22. ∎

Lemma 24 *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* $v \in \mathit{Rv}_1(n)$

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* $v \in \mathit{Rv}_1(n')$.

Proof: Our assumptions entail $n = \mathsf{S}(t)$, $n' = \mathsf{T}(t)$, $\mathsf{G}_1(t) = \mathsf{G}_2(t)$, and $[\![\mathsf{G}_1(t)]\!]s_1 = \mathit{true}$. For an arbitrary $w \in \mathit{fv}(\mathsf{G}_1(t))$ we infer by Fact 11 that $w \in \mathit{Rv}_1(t) \subseteq \mathit{Rv}_1(n)$ implying $s_1(w) = s_2(w)$. Hence also $[\![\mathsf{G}_1(t)]\!]s_2 = \mathit{true}$, 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 $v \in \mathit{Rv}_1(n')$ now follows from Lemma 12. ∎

Theorem 1 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies WCC. If*

- $1 \vdash^t C_1 \xRightarrow{E_1} C_1'$ *and*

- $C_1\ Q\ C_2$

then there exists C_2', and E_2 which is a subsequence of E_1, such that

- $2 \vdash^t C_2 \stackrel{E_2}{\Rightarrow} C_2'$ *and*
- $C_1'\ Q\ C_2'$

Proof: From $C_1\ Q\ C_2$ we see that there exists n, s_1, s_2 such that $C_1 = (n, s_1)$ and $C_2 = (n, s_2)$ where $s_1(v) = s_2(v)$ for all $v \in \mathit{Rv}_1(n)$. From $1 \vdash^t C_1 \stackrel{E_1}{\Rightarrow} C_1'$ we see that $t \in \mathcal{L}$ and that there exists n', s_1', n'', s_1'', E', E_2 such that with $C_1' = (n', s_1')$ and $E_1 = E'E_2$ we have $1 \vdash (n, s_1) \stackrel{E'}{\Rightarrow} (n'', s_1'')$ and $1 \vdash t : (n'', s_1'') \stackrel{E_2}{\rightarrow} (n', s_1')$.

Since $obs(n'')$ is not empty (it contains n'') we can apply Lemma 23 to infer that $2 \vdash (n, s_2) \stackrel{[]}{\Rightarrow} (n'', s_2)$ and $s_1''(v) = s_2(v)$ for all $v \in \mathit{Rv}_1(n'')$. We can now apply Lemma 24 to find s_2' such that $2 \vdash t : (n'', s_2) \stackrel{E_2}{\rightarrow} (n', s_2')$ and $s_1'(v) = s_2'(v)$ for all $v \in \mathit{Rv}_1(n')$.

We conclude that $2 \vdash^t (n, s_2) \stackrel{E_2}{\Rightarrow} (n', s_2')$, and that with $C_2' = (n', s_2')$ we do indeed get $2 \vdash^t C_2 \stackrel{E_2}{\Rightarrow} C_2'$ and $C_1'\ Q\ C_2'$ with E_2 a subsequence of E_1. ∎

8 Original Simulates Sliced

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

Lemma 25 *Let $obs(n) = \{m\}$. Given s, one of the 3 cases below applies:*

1. *there exists E and s' such that $1 \vdash (n, s) \stackrel{E}{\Rightarrow} (m, s')$*
2. *(n, s) gets 1-stuck avoiding m*
3. *(n, s) 1-loops avoiding m.*

Proof: Consider the following iterative algorithm, incrementally constructing n_j, s_j, E_j for $j \geq 1$. With $n_0 = n$, $s_0 = s$ and $E_0 = [\,]$, the invariant is that $1 \vdash (n, s) \stackrel{E_{j-1}}{\Rightarrow} (n_{j-1}, s_{j-1})$ which trivially holds for $j = 1$. For each j, there are 3 possible actions:

- if $n_{j-1} = m$ we exit the loop, and we have established case 1 with $E = E_{j-1}$ and $s' = s_{j-1}$.

  Otherwise, we can conclude that for all t with $\mathsf{S}(t) = n_{j-1}$ it holds that $t \notin \mathcal{L}$ (since it $t \in \mathcal{L}$ then $n_{j-1} \in obs(n) = \{m\}$).

- if there exists t with $\mathsf{S}(t) = n_{j-1}$ such that $[\![\mathsf{G}_1(t)]\!]s_{j-1} = \mathit{true}$ we define t_j as one such t (the "smallest") and define $n_j = \mathsf{T}(t_j)$, $E_j = E_{j-1}\mathsf{E}_1(t_j)$, and $s_j = s_{j-1}$ if $\mathsf{D}_1(t_j) = \emptyset$ but $s_j = s_{j-1}[v \mapsto [\![\mathsf{A}_1(t_j)]\!]s_{j-1}]$ if $\mathsf{D}_1(t_j)$ is a singleton $\{v\}$. We then increment j by one and repeat the loop.

- otherwise, we exit the loop, concluding that (n_{j-1}, s_{j-1}) is 1-stuck which establishes case 2.

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

Theorem 2 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{WCC}$. If with $C_2 = (n, s_2)$ we have $2 \vdash^t C_2 \stackrel{E_2}{\Rightarrow} C_2'$ and $C_1\ Q\ C_2$ then $obs(n)$ is a singleton $\{m\}$ and there are 3 possibilities:*

1. *there exists C_1' with $C_1'\ Q\ C_2'$, and E_1 of which E_2 is a subsequence, such that $1 \vdash^t C_1 \stackrel{E_1}{\Rightarrow} C_1'$*
2. *C_1 gets 1-stuck avoiding m*
3. *C_1 1-loops avoiding m.*

Proof: Let $C_2' = (n', s_2')$ and $C_1 = (n, s_1)$. From Definition 21 and Lemma 20 we see that there exists $t \in \mathcal{L}$, and a state m, such that $2 \vdash (n, s_2) \stackrel{[]}{\Rightarrow} (m, s_2)$ and $2 \vdash t : (m, s_2) \stackrel{E_2}{\rightarrow} (n', s_2')$. Thus $m \in obs(n)$, and from the $\mathcal{WCC}$ property we infer that $obs(n) = \{m\}$.

From Lemma 25, we infer that either

1. there exists E and s_1'' such that $1 \vdash (n, s_1) \stackrel{E}{\Rightarrow} (m, s_1'')$
2. (n, s_1) gets 1-stuck avoiding m
3. (n, s_1) 1-loops avoiding m.

If case 2 or case 3 holds, we are done. We thus assume that case 1 holds. That is, for some $k \geq 1$ there exists $n_1 \ldots n_k$ and $s_1 \ldots s_k$, with $n = n_1$ and $n_k = m$ and $s_k = s_1''$, and $E_1' \ldots E_{k-1}'$ with $E = E_1' \ldots E_{k-1}'$, such that for all $j \in 1 \ldots k-1$ there exists $t_j \notin \mathcal{L}$ with $i \vdash t_j : (n_j, s_j) \stackrel{E_j'}{\rightarrow} (n_{j+1}, s_{j+1})$. We infer that for all $j \in 1 \ldots k$ we have $obs(n_j) = \{m\}$ and by Lemma 15 thus $Rv(n_j) = Rv(m) = Rv(n)$. By repeated application of Lemma 17 we now infer that for all $v \in Rv(m)$ we have $s_1''(v) = s_1(v)$ and from $C_1\ Q\ C_2$ even $s_1''(v) = s_2(v)$.

From $t \in \mathcal{L}$ we have $\mathsf{G}_1(t) = \mathsf{G}_2(t)$, and from $2 \vdash t : (m, s_2) \stackrel{E_2}{\rightarrow} (n', s_2')$ we know that $[\![\mathsf{G}_1(t)]\!]s_2 = \mathit{true}$. For an arbitrary $w \in fv(\mathsf{G}_1(t))$ we infer by Fact 11 that $w \in Rv(t) \subseteq Rv(m)$ implying $s_1''(w) = s_2(w)$. Hence also $[\![\mathsf{G}_1(t)]\!]s_1'' = \mathit{true}$, implying that there exists s_1' such that $1 \vdash t : (m, s_1'') \stackrel{E_2}{\rightarrow} (n', s_1')$, and thus with $C_1' = (n', s_1')$ we have $1 \vdash^t C_1 \stackrel{EE_2}{\Rightarrow} C_1'$. We must also show $C_1'\ Q\ C_2'$, that is $s_1'(v) = s_2'(v)$ for all $v \in Rv(n')$, but this follows from Lemma 12. ∎

If $\mathcal{L}$ satisfies not just $\mathcal{WCC}$ but also $\mathcal{SCC}$, we can rule out case 3. We now state the resulting theorem.

Theorem 3 *Assume that $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{SCC}$. If with $C_2 = (n, s_2)$ we have $2 \vdash^t C_2 \stackrel{E_2}{\Rightarrow} C_2'$ and $C_1\ Q\ C_2$ then $obs(n)$ is a singleton $\{m\}$ and there are 2 possibilities:*

1. *there exists* C_1' *with* $C_1'\ Q\ C_2'$, *and* E_1 *of which* E_2 *is a subsequence, such that* $1 \vdash^t C_1 \stackrel{E_1}{\Rightarrow} C_1'$
2. C_1 *gets 1-stuck avoiding* m

9 Computing Least $\mathcal{WCC}$-Closed Slices

We shall now present an algorithm that for any $\mathcal{L}$ computes the least superset that is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{WCC}$. First some preparations.

Lemma 26 *Assume that for* $i = 1, 2$ *there is a cycle-free path* π_i *from* n *to* n_i, *and a transition* u_i *with* $\mathsf{S}(u_i) = n_i$; *also assume that* $n_1 \neq n_2$, *and that* n *is the only state that occurs in both* π_1 *and* π_2.

Let $\mathcal{L}$ *be a set that satisfies* $\mathcal{WCC}$ *and contains* u_1, u_2. *Let* $i \in \{1, 2\}$ *be such that* π_i *is non-empty (there exists at least one such* i*). Then the first transition in* π_i *will belong to* $\mathcal{L}$.

Proof: For each $i = 1, 2$, let N_i be the states that occur in π_i (if π_i is the empty path we let $N_i = \{n\}$); our assumption is that $N_1 \cap N_2 = \{n\}$. Since $\mathcal{L}$ contains u_1 and u_2, it is easy to see that $obs_{\mathcal{L}}(n)$ will contain at least one state from N_1 and at least one state from N_2; since $\mathcal{L}$ satisfies $\mathcal{WCC}$ and $N_1 \cap N_2 = \{n\}$, we infer that $obs_{\mathcal{L}}(n) = \{n\}$.

To show that $\mathcal{L}$ must contain the first transition in π_i, observe that otherwise $obs_{\mathcal{L}}(n)$ would contain a state in N_i different from n. ∎

Lemma 27 *Assume that* $\mathcal{L}$ *does not satisfy* $\mathcal{WCC}$. *Then there exists a state* n *with the following properties: for* $i = 1, 2$ *there exists* n_i *with* $n_1 \neq n_2$, *a cycle-free path* π_i *outside* $\mathcal{L}$ *from* n *to* n_i, *a transition* $u_i \in \mathcal{L}$ *with* $\mathsf{S}(u_i) = n_i$; *also,* n *is the only state that occurs in both* π_1 *and* π_2.

Proof: Our assumption is that there exists n_0, and $n_1 \neq n_2$, such that $n_1, n_2 \in obs_{\mathcal{L}}(n_0)$. Thus for each $i = 1, 2$ there exists a transition $u_i \in \mathcal{L}$ with $\mathsf{S}(u_i) = n_i$, and a cycle-free path π_i' outside $\mathcal{L}$ from n_0 to n_i.

Now let n be the last state on π_1' that occurs also on π_2' (it could happen that $n = n_0$), and for $i = 1, 2$ let π_i be the part of π_i' that starts from n. ∎

Theorem 4 *Given a slice set* $\mathcal{L}$, *there is a* least *superset of* $\mathcal{L}$ *that is closed under* $\rightarrow_{\mathrm{dd1}}$ *and satisfies* $\mathcal{WCC}$.

Proof: We shall use induction on the number of transitions not in $\mathcal{L}$. If $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{WCC}$ then $\mathcal{L}$ is obviously the least superset of $\mathcal{L}$ with these properties.

Otherwise, we shall establish the following property:

> there exists a transition $t \notin \mathcal{L}$ such that
> if $\mathcal{L}'$ is a superset of $\mathcal{L}$ that is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{WCC}$
> then $t \in \mathcal{L}'$.

Our claim will then follow if we can show that there exists a least superset of $\mathcal{L} \cup \{t\}$ that is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{WCC}$. But this follows by applying the induction hypothesis to $\mathcal{L} \cup \{t\}$.

To verify the above property, we split into two cases. If $\mathcal{L}$ is not closed under $\rightarrow_{\mathrm{dd1}}$, there exists $t \notin \mathcal{L}$ and $t' \in \mathcal{L}$ such that $t \rightarrow_{\mathrm{dd1}} t'$. This t will do the job, for if $\mathcal{L}'$ is a superset of $\mathcal{L}$ that is closed under $\rightarrow_{\mathrm{dd1}}$ then $t' \in \mathcal{L}'$ and hence also $t \in \mathcal{L}'$.

The other case is when $\mathcal{L}$ does not satisfy $\mathcal{WCC}$. By Lemma 27, there thus exists a state n with the following properties: for $i = 1, 2$ there exists n_i with $n_1 \neq n_2$, a cycle-free path π_i outside $\mathcal{L}$ from n to n_i, a transition $u_i \in \mathcal{L}$ with $\mathsf{S}(u_i) = n_i$; also, n is the only state that occurs in both π_1 and π_2. We can then apply Lemma 26 to see that there exists a transition $t \notin \mathcal{L}$ such that if $\mathcal{L}'$ is a superset of $\mathcal{L}$ that satisfies $\mathcal{WCC}$ then $\mathcal{L}'$ contains t. ∎

The proof of Theorem 4 suggests how to compute the least superset of $\mathcal{L}$ that satisfies $\mathcal{WCC}$: first find the states that are the source of a transition in $\mathcal{L}$; next find if two such states are reachable from the same state through a path not in $\mathcal{L}$; if so, add the first transition of that path to $\mathcal{L}$ and repeat, otherwise stop.

An algorithm based on this idea, also ensuring closedness under $\rightarrow_{\mathrm{dd1}}$, is depicted in Fig. 3. It assumes a pre-computed table `DDstar` such that `DDstar(`t`,`u`)` is true iff $t \rightarrow^*_{\mathrm{dd1}} u$ holds. The current slice set is `L` to which transitions may be added until `L` is stable; the algorithm maintains the invariant that `L` is closed under $\rightarrow_{\mathrm{dd1}}$. In each iteration, the algorithm computes in `B` the states that are sources of transitions in `L`, and does a backwards breadth-first search from `B` with `V` being the states that have been visited so far. For each $n \in$ `V`, the array entry obs$[n]$ is defined, and denotes the state in `B` that can be reached from n. The current frontier of the search is called `C`, the exploration of which builds up `Cnew`, the next frontier.

Example 1 *Let us apply the algorithm in Fig. 3 to the EFSM in Fig. 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 $t8$ (or $t7$) which adds $S4$ to B.

In the subsequent iterations we may add $t7$, $t2$, and $t3$ (the latter two due to conflicts at $S1$). We now have $B = \{S1, S3, S4, S5\}$ and add $t4$, before the next iteration adds $t1$ since from $S1$ one can reach $S3$ through the path $[t1, t5]$.

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

Theorem 5 *Assuming that the table `DDstar` is given, the algorithm in Fig. 3 can be implemented to run in time $O(a^2)$ where a is the number of transitions.*

Proof: We assume that sets of states and sets of transitions are represented as bitmaps, allowing one element to be added in constant time. The code before the main loop obviously runs in time $O(a^2)$. The last part of the main loop, the two nested `for` loops, will have a total running time in $O(a^2)$ since each transition occurs in `Lnew` at most once.

```
L := 𝓛
for each t ∈ L
  for each u ∉ L
    if DDstar(u,t)
      L := L ∪ {u}
repeat
  Lnew := ∅
  B := {n | ∃t ∈ L : n = S(t)}
  for each n ∈ B do
    obs[n] := n
  V := B
  C := B
  while C ≠ ∅ and Lnew = ∅ do
    Cnew := ∅
    for each m ∈ C do
      for each transition t ∉ L with T(t) = m do
        n := S(t)
        if n ∈ V
          if obs[n] ≠ obs[m]
            Lnew := Lnew ∪ {t}
        else
          V := V ∪ {n}
          Cnew := Cnew ∪ {n}
          obs[n] := obs[m]
    C := Cnew
  L := L ∪ Lnew
  for each t ∈ Lnew do
    for each u ∉ L do
      if DDstar(u,t)
        L := L ∪ {u}
until Lnew = ∅
```

Figure 3: Computing the least superset of $\mathcal{L}$ that is closed under data dependence and satisfies $\mathcal{WCC}$.

The outer loop iterates $O(a)$ times, as each iteration (except the last) will add at least one transition to L. It is thus sufficient to show that each iteration of the outer loop, except for the two nested `for` loops, runs in time $O(a)$. But this follows since each transition t is processed in constant time and at most once. ∎

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

We shall now present an algorithm that for any $\mathcal{L}$ computes the least superset that is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{SCC}$. First some preparations.

Lemma 28 *Assume that there is a cycle-free path $\pi_1 = [t_1..t_k]$ ($k \geq 1$) from n to n_1, and a transition u_1 with $\mathsf{S}(u_1) = n_1$; also assume that π_2 is an infinite path from n that avoids all states in π_1 except n.*

Let $\mathcal{L}$ be a set that satisfies $\mathcal{SCC}$ and contains u_1. Then t_1 will belong to $\mathcal{L}$.

Proof: Assume, to get a contradiction, that $t_1 \notin \mathcal{L}$. Then there exists $n' \neq n$ such that $n' \in obs_{\mathcal{L}}(n)$ and thus (since $\mathcal{L}$ satisfies $\mathcal{WCC}$) $obs_{\mathcal{L}}(n) = \{n'\}$. But since π_2 is an infinite path from n that avoids n', this contradicts $\mathcal{L}$ satisfying $\mathcal{SCC}$. ∎

Lemma 29 *Assume that $\mathcal{L}$ satisfies $\mathcal{WCC}$ but does not satisfy $\mathcal{SCC}$. Then there exists a state n with the following properties: there is a non-empty cycle-free path π_1 from n to n_1 outside $\mathcal{L}$, and a transition $u_1 \in \mathcal{L}$ with $\mathsf{S}(u_1) = n_1$; also there is an infinite path π_2 from n that avoids all states in π_1 except n.*

Proof: Our assumption entails that there exists n_0 and n_1 such that $obs_{\mathcal{L}}(n_0) = \{n_1\}$, that is there is a cycle-free path π'_1 outside $\mathcal{L}$ from n_0 to n_1 and a transition $u_1 \in \mathcal{L}$ with $\mathsf{S}(u_1) = n_1$, but also an infinite path π'_2 from n_0 that avoids n_1. Now let n be the last state on π'_1 that occurs also on π'_2 (it could happen that $n = n_0$), and for $i = 1, 2$ let π_i be the part of π'_i that starts from n. This will do the job; in particular π_1 is non-empty since π'_2 avoids n_1 and thus $n \neq n_1$. ∎

Theorem 6 *Given a slice set $\mathcal{L}$, there is a* least *superset of $\mathcal{L}$ that is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{SCC}$.*

Proof: We shall use induction on the number of transitions not in $\mathcal{L}$. If $\mathcal{L}$ is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{SCC}$ then $\mathcal{L}$ is obviously the least superset of $\mathcal{L}$ with these properties.

Otherwise, we shall establish the following property:

> there exists a transition $t \notin \mathcal{L}$ such that
> if $\mathcal{L}'$ is a superset of $\mathcal{L}$ that is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{SCC}$
> then $t \in \mathcal{L}'$.

Our claim will then follow if we can show that there exists a least superset of $\mathcal{L} \cup \{t\}$ that is closed under $\rightarrow_{\mathrm{dd1}}$ and satisfies $\mathcal{SCC}$. But this follows by applying the induction hypothesis to $\mathcal{L} \cup \{t\}$.

To verify the above property, we split into three cases. If $\mathcal{L}$ is not closed under $\rightarrow_{\mathrm{dd1}}$, or if $\mathcal{L}$ does not satisfy $\mathcal{WCC}$, we proceed as in the proof of Theorem 4. We therefore focus on the third case: that $\mathcal{L}$ satisfies $\mathcal{WCC}$ but not $\mathcal{SCC}$.

By Lemma 29, there thus exists a state n with the following properties: there is a cycle-free non-empty path π_1 outside $\mathcal{L}$ from n to n_1, and a transition $u_1 \in \mathcal{L}$ with $\mathsf{S}(u_1) = n_1$; also there is an infinite path π_2 from n that avoids all states in π_1 except n.

We can then apply Lemma 28 to see that there exists a transition $t \notin \mathcal{L}$ such that if $\mathcal{L}'$ is a superset of $\mathcal{L}$ that satisfies $\mathcal{SCC}$ then $\mathcal{L}'$ contains t. ∎

The proof of Theorem 6 suggests how to extend the algorithm of Fig. 3 to compute the least superset of $\mathcal{L}$ that satisfies $\mathcal{SCC}$: as we explore each state n from which a state m' in $\mathtt{B}$ is reachable, we check if there from n is a loop that avoids m'; if that

```
L := 𝓛
for each t ∈ L
  for each u ∉ L
    if DDstar(u,t)
      L := L ∪ {u}
repeat
  Lnew := ∅
  B := {n | ∃t ∈ L : n = S(t)}
  for each n ∈ B do
    obs[n] := n
  V := B
  C := B
  while C ≠ ∅ and Lnew = ∅ do
    Cnew := ∅
    for each m ∈ C do
      for each transition t ∉ L with T(t) = m do
        n := S(t)
    **  if LoopAvoids(n,obs[m])
    **    Lnew := Lnew ∪ {t}
        if n ∈ V
          if obs[n] ≠ obs[m]
            Lnew := Lnew ∪ {t}
        else
          V := V ∪ {n}
          Cnew := Cnew ∪ {n}
          obs[n] := obs[m]
    C := Cnew
  L := L ∪ Lnew
  for each t ∈ Lnew do
    for each u ∉ L do
      if DDstar(u,t)
        L := L ∪ {u}
until Lnew = ∅
```

Figure 4: Computing the least superset of $\mathcal{L}$ that is closed under data dependence and satisfies $\mathcal{SCC}$.

is the case, we can add to L the first transition in the path π from n to m', as justified by Lemma 28 (its condition about the loop avoiding all states in π but n is fulfilled, since if the loop doesn't avoid an intermediate state then the breadth-first search would already have added a transition to L).

The resulting algorithm is depicted in Fig. 4; it is identical to the algorithm of Fig. 3 except that two lines (marked with "**") have been added. The algorithm employs a pre-computed table LoopAvoids such that LoopAvoids(n,m) is true iff there is

```
for each state m do
  for each state n do
    LoopAvoids(n,m) := false
  T_m := the transitions that do not involve m
  for each state n except m do
    color[n] := white
  while (exists n:  color[n] = white) do
    let n be a node with color[n] = white
    call DFS(n)

procedure DFS(n) =
  color[n] := gray
  for each t ∈ T_m with S(t) = n do
    n' := T(t)
    if color[n'] = gray
      LoopAvoids(n,m) := true
    else if color[n'] = black
      if LoopAvoids(n',m)
        LoopAvoids(n,m) := true
    else // if color[n'] = white
      DFS(n')
      if LoopAvoids(n',m)
        LoopAvoids(n,m) := true
  color[n] := black
```

Figure 5: Constructing the table `LoopAvoids`.

a loop from n that avoids m.

The table `LoopAvoids` can be constructed by the algorithm in Fig. 5. 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.

Example 2 *Let us apply the algorithm in Fig. 4 to the EFSM in Fig. 2, to find the least set that satisfies $\mathcal{SCC}$ and contains $t9$.*

In the first iteration, $B = \{S5\}$. Since $t2$ has source $S1$ and target $S5$, and there is a loop from $S1$ that avoids $S5$, we add $t2$. Similarly, since $t8$ has source $S4$ and target $S5$, and there is a loop from $S4$ that avoids $S5$, we add $t8$. Now, $B = \{S1, S4, S5\}$. Since $S4$ can be reached from $S1$ by $t3$, also $t3$ is added; since $S1$ can be reached from $S4$ by $[t7, t4]$, also $t7$ is added.

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

Theorem 7 *Assuming that the table* `DDstar` *is given, the algorithm in Fig. 4 can be implemented to run in time* $O(a^2)$*, including the time for constructing the* `LoopAvoids` *table, where* a *is the number of transitions.*

Proof: To construct the table `LoopAvoids`, the algorithm in Fig. 5 will for each m use time in $O(a)$, and hence (since $m \in O(a)$) have a total running time in $O(a^2)$. Next the algorithm in Fig. 4 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 5 for the algorithm in Fig. 3). ∎

11 Conclusion

We have proposed algorithms for slicing extended finite state machines and proved that the resulting slices have well-defined semantic properties. Our development adapts to a non-deterministic setting a general methodology for describing the slicing of deterministic programs. It is left to future work to try our approach on realistic EFSMs and thus measure its practical usefulness. Future work also includes allowing transitions to produce events.

References

[1] Torben Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Inf. Process. Lett.*, 106(2):45–51, 2008.

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

[3] Sebastian Danicic, Richard W. Barraclough, Mark Harman, John D. Howroyd, kos Kiss, and Michael R. Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theoretical Computer Science*, 412(49):6809–6842, 2011.

[4] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Symposium on Static Analysis*, volume 1694 of *Springer Lecture Notes in Computer Science*, pages 1–18, 1999.

[5] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, December 2000.

[6] A. Podgurski and L.A. Clarke. 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.

[7] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *European Symposium on Programming*, volume 3444 of *LNCS*, pages 77–93. Springer-Verlag, 2005.

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

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

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