1 Introduction

Slicing is a program analysis technique that has been used to remove unnecessary program statements with respect to a criterion. It has been used to help with program debugging, testing, program comprehension, re-engineering and software measurement. Different types of slicing algorithms have been proposed, such as static, dynamic, conditional and amorphous slicing. These can be categorised as being syntactic or semantic preserving.

Recently, slicing has been applied at the specification level for making models more manageable by reducing them. There has been research on slicing extended finite state machines (EFSMs) [KSTV03], communicating automata [LGP07], and reducing the size of models for model-checking. In this paper, we give a new definition for control dependency and present an algorithm for slicing EFSMs. The slicing algorithm is amorphous and is categorised as being semantic preserving. We illustrate our algorithm on the EFSM of the ATM system.

2 Extended Finite State Machines

Finite State Machines (FSMs) have become widely used in specifications of embedded and reactive systems. They are often classed as a formal method because they allow the possibility of animation of specifications. As a formal method they have the advantage that they are a graphical formalism and their semantics is relatively intuitive. Their main drawback is that they require large FSM diagrams in order to specify moderately complicated systems. A large diagram is less comprehensible to a human being and tends to negate their advantages.

There have been many attempts in the last 30 years to retain the advantages of FSMs while ameliorating their disadvantages. Notable among these attempts has been Harel’s Statecharts which employed state abstraction. A second, widely used attempt has been Extended Finite State Machines (EFSMs). Normally these are used as communicating finite state machines and the size of the diagram is kept under control by extending an individual machine with a local program store, i.e. a set of (typed) program variables. The set of possible actions generated by taking a transition is extended from events by the addition of update actions on this store. Logical conditions which influence whether or not a transition is taken may also refer to the local store by referring variables. It is these machines that are the object of our research.

We formally define an EFSM as follows.

Definition 1 (Extended Finite State Machine) An Extended Finite State Machine (EFSM) is given by

- A set of States, $S$.
- A set of Transitions, $T$.
• A set of Events, \( E \).
• A store represented by a set of local variables, \( V \).
• A set of Phrases, \( P \) where \( P \) has grammar:

\[
C \in \text{Com} \quad x \in \text{Var} \quad E \in \text{Exp} \quad B \in \text{BExp} \quad n \in \mathbb{Z} \\
\begin{align*}
P & ::= C \mid B \\
C & ::= x := E \mid C_1; C_2 \\
E & ::= x \mid n \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \ast E_2 \mid \\
B & ::= \neg B \mid B_1 \land B_2 \mid E_1 < E_2 \mid E_1 == E_2
\end{align*}
\]

A Transition, \( t \in T \), is given by
• A source state \( \text{src}(t) \in S \).
• A label, \( \text{lbl}(t) \), where \( \text{lbl}(t) \) has the form \( e_1[b]/e_2.c \) where \( e_1, e_2 \in E, b \in B, c \in C \) and all parts of the label are optional.
• A target state \( \text{tgt}(t) \in S \).

States of \( S \) are atomic. Machines are possibly non-deterministic. Actions can involve store updates or generation of events or both. Logical guards, where they exist, refer to the store.

Some useful definitions for transitions are described below.

**Definition 2 (Successor)** A successor of a transition \( t \) is a transition whose source state is the target state of \( t \).

**Definition 3 (Sibling)** A sibling of a transition \( t \) is a transition that has the same source state as \( t \).

**Definition 4 (Final transition)** A final transition is a transition whose target state is the exit state or it has no successor.

### 3 Data dependency and control flow dependency

Traditionally, data dependency and control flow dependency definitions are given for control flow graphs (CFGs). In this section we first discuss the differences between CFGs and EFSMs and then we consider a suitable notion of data and control dependency for EFSMs.

#### 3.1 Differences between CFGs and EFSMs

A CFG is a labeled directed graph with a set of nodes that represent statements in a program and edges that represent the control flow. A node is either a *statement node*, i.e. it has a single successor, or a *predicate node* i.e. it has two successors, labeled with a \( T \) for the true case and \( F \) for the false case. A CFG has to have a start node \( n_s \) such that all nodes are reachable from \( n_s \) and it has no incoming edges. It may have a set of end nodes that have no successors.

The differences between CFGs and EFSMs are discussed (1 and 3 identified by Lu and Lam - need to cite) with the help of an example of a CFG (I) and EFSM (II) for a simple program illustrated in Figure 1. There is an embedding of CFG with system calls into the EFSM. Note that the simple program behaves in the same way in both (I) and (II) but is modelled slightly differently in (II) for the purpose of illustration, i.e. there are three transitions, \( T2, T3 \) and \( T4 \) in (II) instead of just the true and false edges in (I).
In EFSMs, the decision points are at the transitions, while in CFGs they are in the nodes (i.e. predicate nodes). For example, in Figure 1 the predicate node in the CFG (I) is \( b \) and it corresponds to the conditions of transitions \( T_2, T_3 \) and \( T_4 \) in the EFSM (II).

In CFGs predicate nodes only have two edges leading to two distinct nodes, i.e. one representing the true case and other the false case, while in EFSMs a state can be the source of many transitions, i.e. they are unbounded. In the CFG in Figure 1 the transitions representing true and false are labelled with \( T \) and \( F \) respectively and these are the only possible transitions from \( b \). In the EFSM, more than two transitions for decision points can occur, which is illustrated in (II), i.e. \( T_2, T_3 \) and \( T_4 \). Note that EFSMs are specifications that can be implemented in different ways, i.e. even though (I) and (II) represent the same simple program and will produce for any given input the same set of results, in (II) there is an additional condition \( w = 6 \) which has been made explicit with \( T_4 \).

In CFGs there is only one edge of each direction between two nodes, while in EFSMs there is no limitation on the number of transitions between states. For example, in the EFSM in Figure 1 there are two transitions from state \( S_2 \) to \( \text{start} \).

CFGs have only one start node, while EFSMs have a set of start nodes. Moreover, the start node of the CFG has no incoming edges, while the start nodes of an EFSM may have incoming transitions.

The nodes in CFGs do not have any self-looping edges, i.e. edges whose source and target nodes are the same. EFSMs allow for self-looping transitions.

CFGs have no notion of events for triggering edges.
3.2 Dependency Definitions for EFSMs

Batch programs which transform input to output have a single endpoint or exit point: the point of program termination. Traditionally control dependency definitions used in slicing algorithms have been couched in terms of control paths to the exit point of the program [Wei79]. Earlier work on slicing EFSMs also used the idea of an exit point for the control dependence definition [KSTV03]. For reactive systems this is too restrictive since such a system may have several termination points or be non-terminating.

This question of control dependency has been approached from the point of view of Java slicing in [RAB+05]. In what follows we have borrowed liberally from both [RAB+05] and [KSTV03] to produce a definition of control dependence suitable for possibly non-deterministic EFSMs as defined in section 2. The main difference between the work of [RAB+05] and what we present here is a change of emphasis from dependencies between nodes in graphs to dependencies between transitions.

Since a path is commonly presented as a list of nodes, a node is in a path if it is in the list. A transition is in a path if its source node is in the path and its target node is both in the path and immediately follows its source node.

Following [RAB+05] we define a control sink as a region of the graph which, once entered, is never left. These regions are always strongly connected components (SCCs), even if only the trivial SCC, i.e. a single transition with no successors.

**Definition 5 (Control Sink)** A control sink, \( K \), for an EFSM is a set of transitions that form a strongly connected component such that, for each transition \( t \) in \( K \), each successor of \( t \) is in \( K \).

The paths we consider in the EFSM are then no longer paths to an exit node as in [KSTV03] but sink-bounded paths, i.e. those that terminate in a control sink.

**Definition 6 (Maximal Path)** A maximal path is any path in an EFSM that terminates in a final transition or is infinite.

**Definition 7 (Sink-bounded paths)** A set of sink-bounded paths in an EFSM from a transition \( T \), SinkPaths(\( T \)), contains all maximal paths \( \pi \) from \( T \) with the property that there exists a control sink \( K \) such that

1. \( \pi \) contains transition \( T_s \) from \( K \);
2. If \( \pi \) is infinite then all transitions in \( K \) occur infinitely often.

The ‘fairness’ assumption about sink-bounded paths is not strictly necessary at this point. Note that for maximal paths \( \pi_1 \) and \( \pi_2 \), if \( \pi_1 \) is a suffix of \( \pi_2 \), then \( \pi_1 \) is sink-bounded if and only if \( \pi_2 \) is sink-bounded. See the discussion in Section ??.

We now define the notion of control dependence. Following [RAB+05], we define a non-termination insensitive version of control dependency. (That paper also defines a non-termination sensitive control dependency, which can be used to produce slices that contain all loops. This is suitable in the case where certain liveness properties must hold in the slice).

**Definition 8 (Control Dependency)** In an EFSM, a transition \( T_j \) is (directly) control dependent on a transition \( T_i \) if and only if \( T_i \) has at least one sibling \( T_k \) such that

1. For all paths \( \pi \in \text{SinkPaths}(T_i) \), the source of \( T_j \) belongs to \( \pi \);
2. there exists a path \( \pi \in \text{SinkPaths}(T_k) \) such that the source node of \( T_j \) does not belong to \( \pi \).

We adopt the data dependence definition of [KSTV03] for an EFSM. It is stated as being a definition-clear path between a variable’s definition and use.

**Definition 9 (Variable definitions/uses)** Let \( T_i \) be a transition.
• $D(T_i)$ is a set of variables defined by transition $T_i$, i.e. variables defined by actions and variables defined by the event of $T_i$ that are not redefined in any action of $T_i$.

• $U(T_i)$ is a set of variables used in a condition and actions of transition $T_i$.

**Definition 10 (Data Dependency)** Let $D(T_i)$ denote a set of variables defined by transition $T_i$ and $U(T_k)$ denote a set of variables used in transition $T_k$. Then, transitions $T_i$ and $T_k$ are data dependent with respect to variable $v$ if:

1. $v \in D(T_i)$;
2. $v \in U(T_k)$;
3. there exists a path in an EFSM from the src($T_i$) to the tgt($T_k$) whereby $v$ is not modified (definition-clear path).

In [RAB+07], a new notion of dependence called weak order dependence is introduced to define ordering relationships between nodes in irreducible regions of CFGs in which previous definitions prove to be insufficient for obtaining dependences. We also need to define order dependence for EFSMs in order to be able to slice transitions within strongly connected components. In many systems, it is common that the entire EFSM could be a strongly connected component.

**Definition 11 (Weak Order Dependency)** Let $T_i, T_j, T_k$ be distinct transitions in an EFSM. The pair of transitions $T_j$ and $T_k$ are weakly order-dependent on $T_i$ if:

1. there exists a path from the src($T_i$) to the tgt($T_j$) not containing $T_k$;
2. there exists a path from the src($T_i$) to the tgt($T_k$) not containing $T_j$;
3. $T_i$ has a successor $T'$ such that either:
   (a) $T_j$ is reachable from $T'$ and all paths from the src($T'$) to the tgt($T_k$) contain $T_j$
   (b) $T_k$ is reachable from $T'$ and all paths from the src($T'$) to the tgt($T_j$) contain $T_k$

### 4 Slicing and the slicing algorithm

The objective of the slicing algorithm is to automatically compute the slice of an EFSM model $F$ with respect to the slicing criterion.

**Definition 12 (Slicing Criterion)** A slicing criterion for an EFSM is a pair $(t, V)$ where transition $t \in T$ and variable set $V \subseteq \text{Var}$. It designates the point in the evaluation immediately after the execution of the action contained in transition $t$.

An EFSM slice $F'$ is an amorphous machine that contains all transitions that affect the values of $V$ at $t$.

**Definition 13 (Amorphous Machine)** An EFSM $F'$, is an amorphous version of $F$, iff it satisfies a criteria $C$.

Note that this definition does not define what the criteria $C$ is. In other words, there are various types of amorphous machines. Furthermore we define no formal, or informal, notion that an amorphous machine $F'$ must be a submachine of $F$. 


4.1 Slicing algorithm

Given a starting EFSM $F$ and its slicing criterion, the basic idea of the slicing algorithm is as follows:

1. Mark transitions of $F$.
2. Construct a new EFSM $F'$ with unmarked transitions from $F$ removed.
3. Garbage collect $F'$, removing all unreachable nodes and transitions.
4. Merge reducible states in $F'$.

In the following subsections we expand upon these points.

4.1.1 Mark transitions

The dependency graph for the EFSM is constructed using the data and control dependency definitions. Then, using the dependency graph, all backwardly reachable transitions from the slicing criterion are marked. These marked transitions may affect the values of the set of variables $V$ at $t$. All unmarked transitions, i.e. transitions that do not affect the slicing criterion, are anonymised.

4.1.2 Removing unmarked transitions

In order to remove unmarked transitions, we use exactly the same approach as is traditionally used to convert a non-deterministic finite automata to a deterministic finite automata. The classical approach removes silent transitions (that is a transition which requires no symbols to be read in order to be taken) in a way that is guaranteed to produce an equivalent deterministic finite automata. In our approach unmarked transitions are equivalent to silent transitions. We therefore reuse the classical algorithm as it is traditionally presented.

4.1.3 Garbage collection

In the garbage collection phase, we wish to remove orphaned nodes and transitions, which are nodes and transitions not reachable from the original transition of interest after unmarked transitions have been removed. Since such orphaned nodes and transitions may be organised into cycles we use a standard mark and sweep garbage collection (as described in e.g. [JL99]) to remove them from the EFSM.

4.1.4 Merge states

We use the standard FSM minimization algorithm for merging states, an efficient version of which is described in [Hop71]. Intuitively, two states are merged if they have identical outward transitions; this process is repeated until no further states can be merged.

4.2 Detailed slicing algorithm

```plaintext
1: $N_{pre} \leftarrow$ Set of nodes
2: $T_{pre} \leftarrow$ Set of transitions. Each transition has a source, target, condition, and body.
3: stack $\leftarrow$ all starting states
4: $NT \leftarrow \emptyset$ (this is a set of tuples $(t, n)$ of new transitions we’re going to create; the new transition will be a copy of $t$ with the difference that the copies’ source node will be $n$)
5: while stack not empty do
6:   $wn \leftarrow$ pop(stack)
7:   for $t \in$ unmarked transitions for which this node is a route do
8:     for $mt \in$ nearest_marked_transitions($wn$) do
```

9: \( \text{add } (mt, wn) \text{ to } NT \)
10: \textbf{end for}
11: \textbf{end for}
12: \text{add all immediately reachable nodes from } wn \text{ that have not already been visited to } stack \)
13: \textbf{end while}
14: 
15: \( T_{post} \leftarrow T_{pre} \)
16: \textbf{for } (t, n) \in NT \textbf{ do}
17: \( a \leftarrow \text{copy } t, \text{ with the copies source node being } n \)
18: \text{remove } t \text{ from } T_{post} \text{ if it exists}
19: \text{add } a \text{ to } T_{post}
20: \textbf{end for}
21: 
22: \text{remove all unmarked transitions from } T_{post}
23: \( N_{post} \leftarrow \text{garbage\_collect}(N_{pre}, T_{post}) \)
24: \( N_{post}, T_{post} \leftarrow \text{merge\_states}(N_{pre}, T_{post}) \)

\( \text{nearest\_marked\_transitions}(n) \) follows each possible path from \( n \), returning the first marked transition(s) it finds. As it visits a node, it marks it as ‘visited’ to ensure it doesn’t loop infinitely.

\( \text{garbage\_collect}(N, T) \) is a standard text book mark and sweep garbage collect.

\( \text{merge\_states}(N, T) \) is a standard algorithm - there’s a lot of work on ‘state merging’ and ‘state reduction’. Basically it continually iterates over every node in \( N \). If nothing can be merged, it finishes. If something can be merged, it then starts again. At a minimum, two equivalent nodes can be merged, that is ‘nodes that have identical next states and outputs’ (http://www.ee.ic.ac.uk/pcheung/teaching/ee3_DSD/4-FSM.pdf).

### 4.3 Properties of the slicing algorithm

This is semi-obvious:

- \( \mid N_{post} \mid \leq \mid N_{pre} \mid \). This says “we don’t add any new nodes to the system”. The \( \leq \) is necessary because not everything is sliceable, so sometimes slicing will return the original model without changing it.

We need to show:

- \( \mid T_{post} \mid \leq \mid T_{pre} \mid \). It says that the number of transitions in the slice is never more than in the unsliced model.

Here’s something that’s worth noting:

- if \( \mid N_{post} \mid \leq \mid N_{pre} \mid \) then \( \mid T_{post} \mid < \mid T_{pre} \mid \) is legal. Some slices don’t delete nodes, but do delete transitions.

Finally we have a corollary:

- \( \mid N_{post} \mid < \mid N_{pre} \mid \Rightarrow \mid T_{post} \mid < \mid T_{pre} \mid \). If a slice removes nodes, it also removes transitions (the inverse though is not always true).

### 5 Slicing an ATM system

We use the same simplified EFSM of an ATM system (see Figure 2) as defined by Korel et al in [KSTV03] to illustrate how our slicing algorithm works. The ATM system allows a user to perform the following transactions: withdrawal, deposit and check balance, on two possible
accounts: checking and savings accounts. Prior to performing these transactions, a user needs to have entered a valid PIN that is matched against a PIN that is stored on the card. The user is only allowed a maximum of three attempts to enter a correct PIN.

In Figures 3, 4, 5, and 6 we show how the different steps of the slicing algorithm are applied to the EFSM of the ATM system. The slicing criterion is \((s_b, T_{18})\). The EFSM is expressed in terms of states and labelled transitions rather than named transitions (for obvious reasons).
Figure 4: Then the traditional algorithm for converting non-deterministic finite automata to deterministic finite automata is applied and unmarked transition are removed.

Figure 5: After garbage collection

Figure 6: After merging
6 Conclusions

In this paper, we have presented data and control dependency definitions for EFSMs and an amorphous slicing algorithm. The slicing algorithm automatically slices EFSM models with respect to the set of variables of a transition of interest. It is amorphous because the structure of the slice produced is not necessarily the same as that of the original, however the behaviour of the slice is the same as the original (i.e. it is semantic preserving). We have illustrated the workings of our algorithm by applying it to the EFSM of an ATM system.

References


