# Slicing Software for Model Construction

Matthew B. Dwyer* John Hatcliff†
Kansas State University
234 Nichols Hall, Manhattan KS, 66506, USA
{dwyer,hatcliff}@cis.ksu.edu

## Abstract

Applying finite-state verification techniques (*e.g.*, model checking) to software requires that program source code be translated to a finite-state transition system that safely models program behavior. Automatically checking such a transition system for a correctness property is typically very costly, thus it is necessary to reduce the size of the transition system as much as possible. In fact, it is often the case that much of a program's source code is irrelevant for verifying a given correctness property.

In this paper, we apply program slicing techniques to remove automatically such irrelevant code and thus reduce the size of the corresponding transition system models. We give a simple extension of the classical slicing definition, and prove its safety with respect to model checking of linear temporal logic (LTL) formulae. We propose various refinements to slicing that take advantage of common structural patterns appearing in LTL software specifications. Finally, we discuss how this slicing strategy fits into a general methodology for deriving effective software models using abstraction-based program specialization.

## 1 Introduction

Modern software systems are highly complex, yet they must be extremely reliable and correct. In recent years, finite-state verification techniques, including model checking techniques, have received much attention as a software validation method. These techniques have been effective in validating crucial properties of concurrent software systems in a variety of domains including: network protocols [23], railway interlocking systems [5], and industrial control systems [3]. Despite this success, the high cost of automatically checking a given correctness property against a software system (which typically has an enormous state space) casts doubt on whether broad application of finite-state verification to software systems will be cost-effective.

Most researchers agree that the best way to attack the state-explosion problem is to construct a finite-state transition system that safely abstracts the software semantics [7, 10, 26]. The transition system should be small enough to make automatic checking tractable, yet it should large enough to capture all information relevant to the property being checked. One of the primary difficulties is determining which parts of the program are relevant to the property being checked. In this paper, we show how slicing can automatically throw away irrelevant portions of the software code, and hence safely reduce the size of the transition systems that approximates the software's behavior.

We envision slicing as one of a collection of tools for translating program source code to models that are suitable for verification. We previously illustrated how techniques from abstract interpretation and partial evaluation can be integrated and applied to help automate construction of abstract transition systems [11, 20, 21]. Applying these techniques on several realistic software systems [12, 13] has revealed an interesting interaction between slicing and abstraction building: people currently perform slicing-like operations manually to determine the portions of code that are relevant for verifying a given property. Thus, preprocessing software using slicing before applying partial-evaluation-based abstraction techniques can: *(i)* provide a safe approximation of the relevant portions of code, *(ii)* enable scaling of current manual techniques to significantly larger and more complex systems, *(iii)* reduce the number of components for which abstractions must be selected and help guide that selection, and *(iv)* reduce the size of the program to be treated by abstraction-based partial evaluation tools.

This work is part of a larger project on engineering high-assurance software systems. We are building a set of tools that implements the transition system construction methodology above for Ada and Java. In this paper, we use a simple flowchart language in order to formally investigate fundamental issues. We have implemented a prototype for the slicing system in the paper, and based on this we are scaling up the techniques. We refer the reader to the project web-site `http://www.cis.ksu.edu/santos/bandera` for the extended version of this paper (which contains more examples, technical extensions, and proofs), for the prototype, and for applications of the abstraction techniques to concurrent Ada systems.

In the next section, we describe the flowchart language that we use throughout the paper. We then present, in Section 3, the definition of slicing for this language. We discuss a specific finite-state verification technique, LTL model checking, and our approach to constructing safely abstracted transition systems from source code in Section 4. Section 5 describes how slicing can be applied as a pre-phase to transition system construction. Section 6 sketches several methods for deriving slicing criteria from temporal logic specifi-

*Supported in part by NSF and DARPA under grants CCR-9633388, CCR-9703094, CCR-9708184, and NASA under award NAG 21209.

†Supported in part by NSF under grant CCR-9701418, and NASA under award NAG 21209.

cations based on the shape of commonly-used formula patterns. Section 7 discusses related work on slicing, and Section 8 summarizes and concludes with a description of future work.

## 2 The Flowchart Language FCL

### 2.1 Syntax

We take as our source language the simple flowchart language FCL of Gomard and Jones [18, 25, 19]. Figure 1 presents an FCL program that computes the power function. The input parameters the program are `m` and `n`. These variables can be referenced and assigned to throughout the program. Other variables such as `result` can be introduced at any time. The initial value of a variable is 0. The output of program execution is the state of memory when the **return** construct is executed.

Figure 2 presents the syntax of FCL. FCL programs are essentially lists of *basic blocks*. The initial basic block to be executed is specified immediately after the parameter list. In the power program, the initial block is specified by the line `(init)`. Each basic block consists of a *label* followed a (possibly empty) list of *assignments* (we write $\cdot$ for the empty list, and this is elided when the list is non-empty). Each block concludes with a *jump* that transfers control from that block to another one. Instead of including boolean values, any non-zero value represents *true* and zero represents *false* in the test of conditionals.

In the presentation of slicing, we need to reason about nodes in a *statement-level control-flow graph* (CFG) (*i.e.*, a graph where there is a separate node for each assignment and jump) for given program $p$. We will assume that each statement has a unique index $i$ within each block. Then, nodes can be uniquely identified by a pair $[l.i]$ where $l$ is block label and $i$ is an index value. In Figure 1, statement indices are given as annotations in brackets $[\cdot]$. For example, the second assignment in the `loop` block has the unique identifier (or node number) [`loop`.2].

The following definition introduces notions related to statement-level control-flow graphs.

**Definition 1**

- *A* flow graph $G = (N, E, s, e)$ *consists of a set $N$ of statement nodes, a set $E$ of directed control-flow edges, a unique start node $s$, and unique end node $e$.*
- *The* inverse $G^{-1}$ *of a flowgraph $(N, E, s, e)$ is the flowgraph $(N, E^{-1}, e, s)$ (i.e., all edges are reversed and start/end states are swapped).*
- *Node $n$* dominates *node $m$ in $G$ (written $dom(n, m)$) if every path from the start node $s$ to $m$ passes through $n$. (note that this makes the dominates relation reflexive).*
- *Node $m$* post-dominates *node $n$ in $G$ (written post-$dom(m, n)$) if every path from node $m$ to the end node $e$ passes through $n$ (equivalently, $dom(n, m)$ in $G^{-1}$).*
- *Node $n$ is* control-dependent *on $m$ (some intuition follows this definition) if*
  1. *there exists a non-trivial path $p$ from $m$ to $n$ such that every node $m' \in p - \{m, n\}$ is post-dominated by $n$, and*
  2. *$m$ is not post-dominated by $n$. [33]*

  *We write $cd(n)$ for the set of nodes on which $n$ is control-dependent.*

Control dependence plays an important role in the rest of the paper. Note that for a node $n$ to be control-dependent on $m$, $m$ must have a least two exit edges, and there must be two paths that connect $m$ with $e$ such that one contains $n$ and the other does not. For example, in the power program of Figure 1, [`loop`.1], [`loop`.2], and [`loop`.3] are control-dependent on [`test`.1], but [`end`.1] is not since it post-dominates [`test`.1] (*i.e.*, all paths from [`test`.1] to `halt` go through it).

Extracting the CFG from an FCL program $p$ is straightforward. The only possible hitch is that some programs do not satisfy the "unique end node" property required by the definition (for example, the program may have multiple **return**'s). To work around this problem, we assume that when we extract the CFG from a program $p$, we insert an additional node labeled `halt` that has no successors and its predecessors are all the **return** nodes from $p$.

### 2.2 Semantics

The semantics of an FCL program $p$ is expressed as transitions on program states $([l.n], \sigma)$ where $l$ is the label of a block in $p$, $n$ is the index of the statement in block $l$, and $\sigma$ is a store mapping variables to values. A series of transitions gives an *execution trace* through a program's statement-level control flow graph. For example, Figure 3 gives a trace of the power program computing $5^2$. Formally, a trace is finite nonempty sequence of states written $\pi = s_1, s_2, \ldots, s_k$. We write $\pi^i$ for the suffix starting at $s_i$, *i.e.*, $\pi^i = s_i, s_{i+1}, \ldots$ [1]. We omit a formal definition of the transition relation for FCL programs since it is intuitively clear (a formalization can be found in [19, 20]).

## 3 Slicing

### 3.1 Program slices

A *program slice* consists of the parts of a program $p$ that (potentially) affect the variable values that flow into some program point of interest [31]. A *slicing criterion* $C = (n, V)$ specifies the program point $n$ (a node in $p$'s CFG) and a set of variables $V$ of interest.

For example, slicing the *power* program with respect to the slicing criterion $C = ([\texttt{loop}.2], \{\texttt{n}\})$ yields the program in Figure 4. Note that the assignments to variables `m` and `result` and the declaration of `m` as an input parameter have been sliced away since they do not affect the value of `n` at line [`loop`.2]. In addition, block `init` is now trivial and can be removed, *e.g.*, in a post-processing phase.

Slicing a program $p$ yields a program $p_s$ such that the traces of $p_s$ are projections of corresponding traces of $p$. For example, the following trace of $p_s$ is a projection of the trace

[1] Here, we consider only finite traces (corresponding to terminating executions). The extended version of the paper treats infinite executions, which are best expressed using co-inductive reasoning

```
(m n)
(init)

  init: result := 1;   [1]          loop: result := *(result m);   [1]
        goto test;     [2]                n := -(n 1);             [2]
                                          goto test;               [3]
  test: if <(n 1)      [1]
           then end
           else loop;                end:  return;                  [1]
```

Figure 1: An FCL program to compute $m^n$

*Syntax Domains*

| | | | | | |
|---|---|---|---|---|---|
| $p$ | $\in$ | Programs[FCL] | $x$ | $\in$ | Variables[FCL] |
| $b$ | $\in$ | Blocks[FCL] | $e$ | $\in$ | Expressions[FCL] |
| $l$ | $\in$ | Block-Labels[FCL] | $c$ | $\in$ | Constants[FCL] |
| $a$ | $\in$ | Assignments[FCL] | $j$ | $\in$ | Jumps[FCL] |
| $al$ | $\in$ | Assignment-Lists[FCL] | $o$ | $\in$ | Operations[FCL] |

*Grammar*

| | | | | | |
|---|---|---|---|---|---|
| $p$ | ::= | $(x^*)\ (l)\ b^+$ | $a$ | ::= | $x$ **:=** $e$; $\mid$ **skip**; |
| $b$ | ::= | $l:\ al\ j$ | $e$ | ::= | $c \mid x \mid o(e^*)$ |
| $al$ | ::= | $a\ al \mid \cdot$ | $j$ | ::= | **goto** $l$; $\mid$ **return**; $\mid$ **if** $e$ **then** $l_1$ **else** $l_2$; |

Figure 2: Syntax of the Flowchart Language FCL

$$
\begin{aligned}
& ([\texttt{init}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 0]) \\
\to\ & ([\texttt{init}.2], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) \\
\to\ & ([\texttt{test}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) \\
\to\ & ([\texttt{loop}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) \\
\to\ & ([\texttt{loop}.2], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 5]) \\
\to\ & ([\texttt{loop}.3], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 5]) \\
\to\ & ([\texttt{test}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 5]) \ldots
\end{aligned}
\qquad
\begin{aligned}
\ldots \to\ & ([\texttt{loop}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 5]) \\
\to\ & ([\texttt{loop}.2], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 25]) \\
\to\ & ([\texttt{loop}.3], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 0, \texttt{result} \mapsto 25]) \\
\to\ & ([\texttt{test}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 0, \texttt{result} \mapsto 25]) \\
\to\ & ([\texttt{end}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 0, \texttt{result} \mapsto 25]) \\
\to\ & (\texttt{halt}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto 0, \texttt{result} \mapsto 25])
\end{aligned}
$$

Figure 3: Trace of power program with $\texttt{m} = 5$ and $\texttt{n} = 2$

```
(n)
(init)

  init: goto test;     [2]          loop: n := -(n 1);             [2]
                                          goto test;               [3]
  test: if <(n 1)      [1]
           then end
           else loop;                end:  return;                  [1]
```

Figure 4: Slice of *power* with respect to criterion $C = ([\texttt{loop}.2], \{\texttt{n}\})$

in Figure 3.

$$\begin{array}{rl} & ([\texttt{init}.2], [\texttt{n} \mapsto 2]) \\ \rightarrow & ([\texttt{test}.1], [\texttt{n} \mapsto 2]) \\ \rightarrow & ([\texttt{loop}.2], [\texttt{n} \mapsto 2]) \\ \rightarrow & ([\texttt{loop}.3], [\texttt{n} \mapsto 1]) \\ \rightarrow & \cdots \\ \rightarrow & ([\texttt{end}.1], [\texttt{n} \mapsto 0]) \\ \rightarrow & (\texttt{halt}, [\texttt{n} \mapsto 0]) \end{array}$$

Intuitively, a trace $\pi_2$ is a projection of a trace $\pi_1$ if the sequence of program states in $\pi_2$ can be embedded into the sequence of states in $\pi_1$. To formalize this, let $\sigma\,|_V$ denote the restriction of the domain of $\sigma$ to the variables in $V$. Then, the definition of projection is as follows.

**Definition 2 (projection)** *Let $p$ be a program. A projection function $\downarrow[M, \nu]$ for $p$-traces is determined by*

- *a set of nodes $M$ from $p$'s CFG, and*
- *a function $\nu$ that maps each node in $M$ to a set of variables $V$*

*and is defined by induction on the length of traces as follows:*

$$\downarrow[M,\nu]((n,\sigma),\, s_2,\, \ldots\,, s_k) = \begin{cases} (n, \sigma\,|_{\nu(n)}), \downarrow[M,\nu](s_2,\, \ldots\,, s_k) & \text{if } n \in M \\ \downarrow[M,\nu](s_2,\, \ldots\,, s_k) & \text{if } n \notin M \end{cases}$$

In the classical definition [31, 32] of slicing criterion, one specifies exactly one point node of interest in the CFG along with a set of variables of interest at that node. This was the case with the example slice of the power program above.

For our applications, we may be interested in *multiple* program points, and so we generalize the notion of slicing criterion as follows.

**Definition 3 (slicing criterion)** *A slicing criterion $C$ for a program $p$ is a non-empty set of pairs*

$$\{(n_1, V_1),\, \ldots\,,\, (n_k, V_k)\}$$

*where each $n_i$ is a node in $p$'s statement flow-graph and $V_i$ is a subset (possibly empty) of the variables in $p$. The nodes $n_i$ are required to be pairwise distinct.*

Note that a criterion $C$ can be viewed as a function from $\{n_1, \ldots, n_k\}$ to $\wp$(Variables[FCL]). In this case, we write *domain*$(C)$ to denote $\{n_1, \ldots, n_k\}$. Thus, a slicing criterion $C$ determines a projection function $\downarrow[\mathit{domain}(C), C]$ which we abbreviate as $\downarrow[C]$. We can now formalize the notion of program slice.

**Definition 4 (program slice)** *Given program $p$ with an associated CFG, let $C$ be a slicing criterion for $p$. Then a program $p_s$ (also called the* residual *program) is slice of $p$ with respect to $C$ if for any $p$ execution trace $\pi = s_0, \ldots, s_k$,*

$$\downarrow[C](\pi) \;= \downarrow[C](\pi_s)$$

*where $\pi_s$ is the execution trace of $p_s$ running with initial state $s_0$.*

For example, let $C = \{([\texttt{loop}.2], \{\texttt{n}\})\}$, and let $\pi$ and $\pi_s$ be the execution traces of the power program (Figure 1) and the slice of the power program (Figure 4), respectively. Then,

$$\begin{array}{rl} & \downarrow[C](\pi) \\ = & \downarrow[C](\pi_s) \\ = & ([\texttt{loop}.2], [\texttt{n} \mapsto 2]), ([\texttt{loop}.2], [\texttt{n} \mapsto 1]). \end{array}$$

## 3.2 Computing slices

Given a program $p$ and slicing criterion $C$, Definition 4 admits many programs $p_s$ as slices of $p$ (in fact, $p$ itself is a (trivial) slice of $p$). Weiser notes that the problem of finding a statement minimal slice of $p$ is incomputable [32]. Below we give a minor adaptation of Weiser's algorithm for computing conservative slices, *i.e.*, slices that may contain more statements than necessary.[2]

### 3.2.1 Initial approximation of a slice

Computing a slice involves (among other things) identifying assignments that can affect the values of variables given in the slicing criterion. To do this, one computes information similar to *reaching definitions*. This requires keeping track of the variables referenced and the variables defined at each node in the CFG.

**Definition 5 (definitions and references)**

- *Let def$(n)$ be the set of variables defined (i.e., assigned to) at node $n$ (always a singleton or empty set).*
- *Let ref$(n)$ be the set of variables referenced at node $n$.*

Figure 5 shows the *def* and *ref* sets for the power program of Figure 1.

Next, for each node in the CFG we compute a set of *relevant variables*. Relevant variables are those variables whose values must be known so as to compute the values of the variables in the slicing criterion.

**Definition 6 (initially relevant variables)** *Let $C = \{(n_1, V_1), \ldots, (n_k, V_k)\}$ be a slicing criterion. Then $R^0_C(n)$ is the set of all variables $v$ such that either:*

1. *$n = n_i$ and $v \in V_i$ for some $i \in \{1, \ldots, k\}$, or*
2. *$n$ is an immediate predecessor of a node $m$ such that either:*
   - *(a) $v \in$ ref$(n)$ and def$(n) \cap R^0_C(m) \neq \emptyset$, or*
   - *(b) $v \notin$ def$(n)$ and $v \in R^0_C(m)$.*

Intuitively, a variable $v$ is relevant at node $n$ if (1) we are at the line of the slicing criterion and we are slicing on $v$, or (2) $n$ immediately precedes a node $m$ such that (a) $v$ is used to define a variable $x$ that is relevant at $m$ (*i.e.*, the value of $x$ depends on $v$), or (b) $v$ is relevant at $m$ and it is not "killed off" by the definition at line $n$. Figure 5 presents the initial sets of relevant variables sets for the power program of Figure 1 with slicing criterion $C = ([\texttt{loop}.2], \{\texttt{n}\})$. Intuitively, n is relevant along all paths leading into node [loop.2]. In the end block, n is a dead variable and thus it is no longer relevant.

The classical definition of slicing does not require nodes mentioned in the slice criterion to occur in the computed slice. To force these nodes to occur, we define a set of *obligatory nodes* — nodes that must occur in the slice even if they fail to define variables that are eventually deemed relevant.

**Definition 7 (obligatory nodes)** *The set $O_C$ of obligatory nodes is defined as follows:*

$$O_C \;=\; \{n \in N \;\mid\; (n, V) \in C\}$$

[2] The algorithm we give actually is based on Tip's corrected version of Weiser's algorithm [31].

| *Node* | *def* | *ref* | *cd* | $R_C^0$ | $R_C^1$ |
|---|---|---|---|---|---|
| [init.1] | {result} | $\emptyset$ | $\emptyset$ | {n} | {n} |
| [init.2] | $\emptyset$ | $\emptyset$ | $\emptyset$ | {n} | {n} |
| [test.1] | $\emptyset$ | {n} | {[test.1]} | {n} | {n} |
| [loop.1] | {result} | {result, m} | {[test.1]} | {n} | {n} |
| [loop.2] | {n} | {n} | {[test.1]} | {n} | {n} |
| [loop.3] | $\emptyset$ | $\emptyset$ | {[test.1]} | {n} | {n} |
| [end.1] | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

$O_C = \{[\texttt{loop.2}]\}$ $\quad S_C^0 = \{[\texttt{loop.2}]\}$ $\quad B_C^0 = \{[\texttt{test.1}]\}$ $\quad S_C^1 = \{[\texttt{loop.2}], [\texttt{test.1}]\}$ $\quad B_C^1 = \{[\texttt{test.1}]\}$

Figure 5: Results of the slicing algorithm for the power program and slicing criteria $C = ([\texttt{loop.2}], \{\texttt{n}\})$

The initial slice set $S_C^0$ is the set of nodes that define variables that are relevant at a successor.

**Definition 8 (initial slice set)** *The initial slice set $S_C^0$ is defined as follows:*

$$\{n \in N \mid \exists m \,.\, (n,m) \in E \wedge R_C^0(m) \cap \mathit{def}(n) \neq \emptyset\}$$

Figure 5 presents the initial slice set $S_C^0$ for the power program of Figure 1. Node [loop.2] is the only node in $S_C^0$ since it is the only node that defines a variable that is relevant at a successor.

Note that $S_C^0$ does not include any conditionals since conditionals make no definitions. How do we tell what conditionals should be added? Intuitively, a conditional at node $n$ should be added if $m \in S_C^0 \cup O_C$ and $m$ is control-dependent on $n$. This set of conditionals $B_C^0$ is called the *branch set*.

**Definition 9 (branch set)** *The initial branch set $B_C^0$ is defined as follows:*

$$B_C^0 = \bigcup_{n \in (S_C^0 \cup O_C)} \mathit{cd}(n)$$

Figure 5 presents the control-dependence information and the initial branch set $B_C^0$ for the power program of Figure 1. As explained in Section 2.1, [loop.1], [loop.2], and [loop.3] are control-dependent on [test.1]. Since [loop.2] is in $S_C^0 \cup O_C$, control-dependency dictates that [test.1] be included in the $B_C^0$.

## 3.3 Iterative construction

Now we have to keep iterating this process. That is, we add the conditionals that influence nodes already in the slice. Then, we must add to the slice nodes that are needed to compute expressions in the tests of conditionals, and so on until a fixed point is reached.

**Definition 10 (iterations)**

- *relevant variables*

$$R_C^{i+1}(n) = R_C^i(n) \cup \bigcup_{b \in B_C^i} R^0_{bc(b)}(n)$$

*where the branch criterion $bc(b) = \{(b, \mathit{ref}(b))\}$* That is, the relevant variables at node $n$ are those that were relevant in the previous iteration, plus those that are needed to decide the conditionals that control definitions in the previous slice set. Finding such nodes for a branch $b$ is equivalent to slicing the program with the criterion $\{(b, \mathit{ref}(b))\}$.

- *slice set*

$$S_C^{i+1} = \{n \in N \mid (n \in B_C^i) \vee (\exists m \,.\, (n,m) \in E \wedge R_C^{i+1}(m) \cap \mathit{def}(n) \neq \emptyset)\}$$

That is, the slice set contains all the conditionals that controlled nodes in the previous slice set, and all nodes that define relevant variables.

- *branch set*

$$B_C^{i+1} = \{b \in N \mid \exists n \in S_C^{i+1} \cup O_C \,.\, b \in \mathit{cd}(n)\}$$

That is, the conditionals required are those that control nodes in the current slice set or obligatory nodes.

Figure 5 presents the sets $R_C^1$, $S_C^1$, and $B_C^1$ which result from the second iteration of the algorithm. On the next iteration, a fixed point is reached since n is the only variable required to compute the conditional test at [test.1] and it is already relevant at [test.1].

In the iterations, the size of $R_C^i(n)$ for all nodes $n$ and $S_C^i$ is increasing, and since $R_C^i(n)$ is bounded above by the number of variables in the program and $S_C^i$ is bounded above by the number of nodes in the CFG, then the iteration eventually reaches a fixed points $R_C^i(n)$ and $S_C^i$.

## 3.4 Constructing a residual program

Given $R_C$ and $S_C$, the following definition informally summarizes how a residual program is constructed. The intuition is that if an assignment is in $S_C$, then it must appear in the residual program. If the assignment is not in $S_C$ but in $O_C$, then the assignment must be to an irrelevant variable. Since the node must appear in the residual program, the assignment is replaced with a **skip**. All **goto** and **return**

jumps must appear in the residual program. However, if an **if** is not in $S_C$, then no relevant assignment or obligatory node is control dependent upon it. Therefore, it doesn't matter if we take the true branch or the false branch. In this case, we can simply jump to the point where the two branches merge.

**Definition 11 (residual program construction)** *Given a program $p = (x^*)\ (l)\ b^+$ and slicing criterion $C$, let $R_C$, $S_C$, $O_C$ be the sets constructed by the process above. A residual program $p_s$ is constructed as follows.*

- *For each parameter $x$ in $p$, $x$ is a parameter in $p_s$ only if $x \in R_C([l.1])$ where $l$ is the label of the initial block of $p$.*
- *The label of the initial block of $p_s$ is the label of the initial block of $p$.*
- *For each block $b$ in $p$, form a residual block $b_s$ as follows.*
  - *For each assignment line $a$ (with identifier $[l.i]$), if $[l.i] \in S_C$ then assignment $a$ appears in the residual program with identifier $[l.i]$, otherwise if $[l.i] \in O_C$ then the assignment becomes a* **skip** *with identifier $[l.i]$ in the residual program, otherwise the node is left out of the residual program.*
  - *For jump $j$ in $b$, if $j =$* **goto** *$l'$; or $j =$* **return**; *then $j$ is the jump in $b_s$, otherwise we must have $j =$* **if** *$e$* **then** *$l_1$* **else** *$l_2$; with some identifier $[l.i]$. Now if $[l.i] \in S_C$ then $j$ is the jump in $b_s$, otherwise the jump in $b_s$ is* **goto** *$l'$; with identifier $[l.i]$ where $l'$ is the label of the nearest post-dominating block for both $l_1$ and $l_2$.*

*Finally, post-processing removes all blocks that are not targets of jumps in $p_s$ (these have become unreachable).*

## 4 Finite-state Verification

As noted in the introduction, a variety of finite-state verification techniques have been used to verify properties of software. To make our presentation more concrete, we consider a single finite-state verification technique: model checking of specifications written in linear temporal logic (LTL). LTL model checking has been used to reason about properties of a wide range of real software systems; we have used it, for example, to validate properties of a programming framework that provides parallel scheduling in a variety of systems (e.g., parallel implementations of finite-element, computational fluid dynamics, and program flow analysis problems) [16, 15].

### 4.1 Linear temporal logic

Linear temporal logic [27] is a rich formalism for specifying state and action sequencing properties of systems. An LTL specification describes the intended behavior of a system on all possible executions.

The syntax of LTL includes primitive propositions $P$ with the usual propositional connectives, and three temporal operators.

$$\begin{array}{rcl} & & \textit{(propositional connectives)} \\ \psi & ::= & P \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 \Rightarrow \psi_2 \mid \\ & & \textit{(temporal operators)} \\ & & \Box\psi \mid \Diamond\psi \mid \psi_1\ \mathcal{U}\ \psi_2 \end{array}$$

When specifying properties of software systems, one typically uses LTL formulas to reason about execution of particular program points (*e.g.*, entering or exiting a procedure) as well as values of particular program variables. To capture the essence of this for FCL, we use the following primitive propositions.

$$P \quad ::= \quad [l.i] \mid [x\ rop\ c]$$

- Intuitively, $[l.i]$ holds when execution reaches node $i$ in the block labeled $l$.
- Intuitively, $[x\ rop\ c]$ holds when the value of variable $x$ at the current node is related to $[\![c]\!]$ by the relational operator *rop* (*e.g.*, [x=0] where *rop* is =).

Formally, the semantics of a primitive proposition is defined with respect to states.

$$[\![[l.i]]\!](n,\sigma) = \begin{cases} \mathit{true} & \text{if } n = [l.i] \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$[\![[x\ rop\ c]]\!](n,\sigma) = \begin{cases} \mathit{true} & \text{if } \sigma(x)\ [\![rop]\!]\ [\![c]\!] \\ \mathit{false} & \text{otherwise} \end{cases}$$

The semantics of a formula is defined with respect to a trace. The temporal operator $\Box$ requires that its argument be true from the current state onward, the $\Diamond$ operator requires that its argument become true at some point in the future, and the $\mathcal{U}$ operator requires that its first argument is true up to the point where the second argument becomes true. Formally [24], let $\pi = s_1, \ldots, s_k$. Then,

$$\begin{array}{rcl} \pi \models [l.i] & \text{iff} & [\![[l.i]]\!]s_1 = \mathit{true} \\ \pi \models [x\ rop\ c] & \text{iff} & [\![[x\ rop\ c]]\!]s_1 = \mathit{true} \\ \pi \models \psi_1 \wedge \psi_2 & \text{iff} & \pi \models \psi_1 \text{ and } \pi \models \psi_2 \\ \pi \models \psi_1 \vee \psi_2 & \text{iff} & \pi \models \psi_1 \text{ or } \pi \models \psi_2 \\ \pi \models \psi_1 \Rightarrow \psi_2 & \text{iff} & \pi \models \psi_1 \text{ implies } \pi \models \psi_2 \\ \pi \models \Box\psi & \text{iff} & \pi^i \models \psi \text{ for all } i \\ \pi \models \Diamond\psi & \text{iff} & \pi^i \models \psi \text{ for some } i \\ \pi \models \psi_1\ \mathcal{U}\ \psi_2 & \text{iff} & \text{there exists an } i \text{ such that} \\ & & \pi^i \models \psi_2, \text{ and for all} \\ & & j = 1, \ldots, i-1, \pi^j \models \psi_1 \end{array}$$

Here are some simple specifications using the logic:

- $\Diamond$[l5.1]
  eventually block l5 will be executed
- $\Box([\text{l2.1}] \Rightarrow \Diamond[\text{l3.1}])$
  whenever block l2 is executed, block l3 is always subsequently executed
- $\Box([\text{l5.1}] \Rightarrow \neg(\text{x} = 0))$
  whenever block l5 is executed x is non-zero
- $\Box(\text{x<10})$
  x is always less than 10

### 4.2 Software model construction

To apply finite-state verification to a software system, one must construct a finite-state transition system that safely abstracts the software semantics. The transition system should be small enough to make automatic checking tractable, yet it should be large enough to capture all information relevant to the property being checked. Relevant information can by extracted by an appropriate abstract interpretation (AI) [9].

In our approach [12, 20], the user declares for each program variable an abstract domain to be used for interpreting operations on the variable. Using a process that combines abstract interpretation and partial evaluation (which we call *abstraction-based program specialization* (ABPS)), a residual program is created by propagating abstract values and specializing each program point with respect to these abstract values [20, 21]. In the residual program, concrete constants are replaced with abstract constants. The residual program is a safely approximating finite-state program with a fixed number of variables defined over finite abstract domains. This program can then be submitted to a toolset [8, 14] that generates input for existing model checking tools, such as SMV [28] and SPIN [23]. This approach has been applied to verify correctness properties of several software systems written in Ada [12, 13].

In the steps described above, *the user's main task is to pick appropriate AI's*, *i.e.*, AI's that extract relevant information, but throw away irrelevant information. The general idea behind our methodology for chosing AI's is to start simple (use an AI's that throw all information about dataflow away) and incrementally refine the AI's based on information from the specification to be verified and from the program.

1. **Start with the point AI:** Initially all variables are modeled with the *point* AI (*i.e.*, a domain with a single value $\top$ where all operations return $\top$). In effect, this throws away all information about a variable's value.
2. **Identify semantic features in the specification:** The specification formula to be checked includes, in the form of propositions, different semantic features of the program (e.g., valuations of specific program variables). These features must be modeled precisely by an AI to have any hope of checking the property. For example, if the formula includes a proposition [$x$=0], then instead of using the point AI for $x$, one must use *e.g.*, an AI with the domain {zero, pos, $\top$} that we refer to as a *zero-pos* AI.
3. **Select controlling variables:** In addition to variables mentioned explicitly in the specification, we must also use refined AI's for variables on which specification variables are control dependent. The predicates in the controlling conditionals suggest semantic features that should be modeled by an AI. For example, if a specification variable $x$ is control-dependent upon a conditional **if**`even`?($y$) ... one should use an even/odd AI for $y$.
4. **Select variables with broadest impact:** When confronted with multiple controlling variables to model, select the one that appears most often in a conditional.

To illustrate the methodology, Figure 6 presents an FCL rendering of an Ada process that controls readers and writers of a common resource [8]. In the Ada system, this server process runs concurrently with other client processes, and requests such as `start-read`, `stop-read` are entry points (rendezvouz points) in the control process. In the FCL code of Figure 6, requests are given in the program parameter `reqs` – a list of values in the subrange [1..4]. Figure 7 presents the block-level control-flow graph for the FCL program.

Assume we are interested in reasoning about the invariance property

$$\Box([\texttt{start-read.1}] \Rightarrow [\texttt{WriterPresent=0}]).$$

The key features that are mentioned explicitly in this specification are values of variable `WriterPresent` and execution of the `start-read` block. The point AI does not provide enough precision to determine the states where `WriterPresent` has value zero. An effective AI for `WriterPresent` must be able to distinguish zero values from positive values; we choose the *zero-pos* AI.

At this point we could generate an abstracted model and check the property or consider additional refinements of the model; we choose the latter for illustrating our example. We now determine the variables upon which the node [`start-read.1`] and nodes with assignments to `WriterPresent` are control dependent. In our example, there are three such variables: `WriterPresent`, `ActiveReaders` and `req`. We are already modeling `WriterPresent` and `req` is being used to model external choice of interactions with the control program via input. We could choose to bind `ActiveReaders` to a more refined AI than point. Given that the conditional expressions involving that variable are `ActiveReaders=0` and `ActiveReaders>0`, we might also choose a *zero-pos* AI. Thus, only `ErrorFlag` is abstracted using the point AI.

At this point, we would generate an abstracted model and check the property. If a true result is obtained then we are sure that the property holds on the program, even though the finite-state system only models two variables with any precision. If a false result is obtained then we must examine the counter-example produced by the model checker. It may reveal a true defect in the program or it may reveal an infeasible path through the model. In the latter case, we identify the variables in the conditionals along the counter-example's path as candidates for binding to more precise AIs.

This methodology is essentially a heuristic search to find the variables in the program that can influence the execution behavior of the program relative to the property's propositions. When a variable is determined to be potentially influential, its abstraction is refined to strengthen the resulting system model. In the absence of such a determination, the variable is modeled with a *point* abstraction which essentially ignores any effect it may have; although in the future it may be determined to have an influence in which case its abstraction will be refined.

## 5 Reducing Models Using Slicing

As illustrated above, picking appropriate abstractions is nontrivial and could benefit greatly from some form of automated assistance. The key aspects of the methodology for picking abstractions included

1. picking out an initial set of relevant variables $V$ and relevant statements (*i.e.*, CFG nodes $N$) mentioned in the LTL specification,

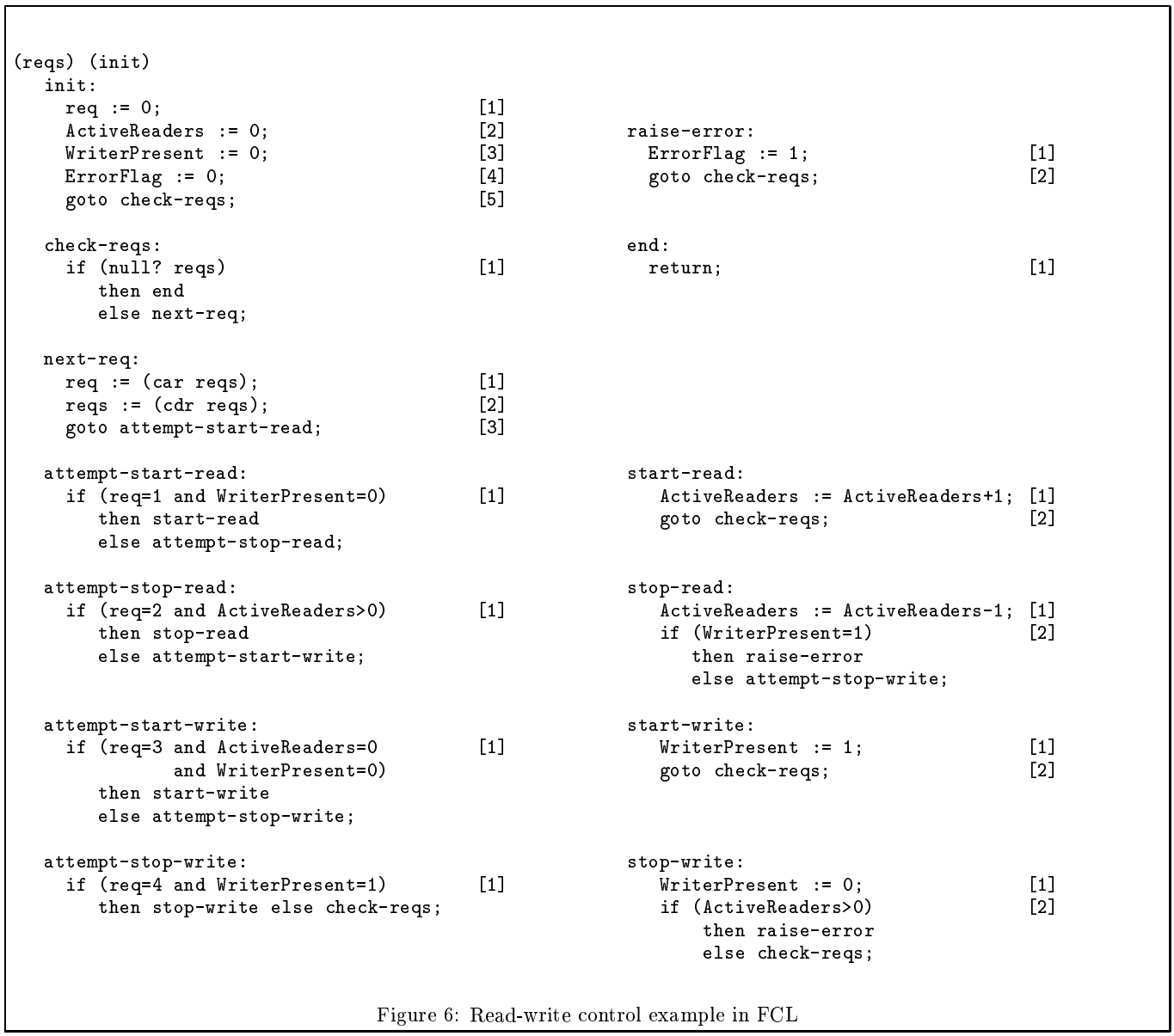


Figure 6: Read-write control example in FCL

2. identifying appropriate AI's for variables in $V$,
3. using control dependence information, picking out an additional set(s) of variables $W$ that indirectly influence $V$ and $N$, and
4. identifying appropriate AI's for variables in $W$.

Intuitively, all variables not in $V \cup W$ are irrelevant and can be abstracted with the point AI.

Clearly, item (1) can be automated by a simple pass over the LTL specification. Moreover, the information in item (3) is *exactly* the information that would be produced by slicing the program $p$ based on a criterion generated from information in (1). Thus, pre-processing a program to be verified using slicing provides automated support for our methodology. Specifically, slicing can *(i)* identify relevant variables (which require AI's other than the point AI), *(ii)* eliminate irrelevant program variables from consideration in the abstraction selection process (they will not be present in the residual program $p_s$ yielded by slicing), and *(iii)* reduce the size of the software and thus the size of the transition system to be analyzed. Other forms of support are needed for items (2) and (4) above.

For this approach, given a program $p$ and a specification $\psi$, we desire a *criterion extraction function* extract that extracts an appropriate slicing criterion $C$ from $\psi$. Slicing $p$ with respect to $C$ should yield a smaller residual program $p_s$ that (a) preserves and reflects the satisfaction of $\psi$, and (b) has as little irrelevant information as possible.

The following requirement expresses condition (a) above.

**Requirement 1 (LTL-preserving** extract**)** *Given program $p$ and a specification $\psi$, let $C = \mathsf{extract}(\psi)$, and let $p_s$ the result of slicing $p$ with respect to $C$. Then for any $p$ execution trace $\pi = s_1, \ldots, s_k$,*

$$\pi \models \psi \ \textit{ iff } \ \pi_s \models \psi$$

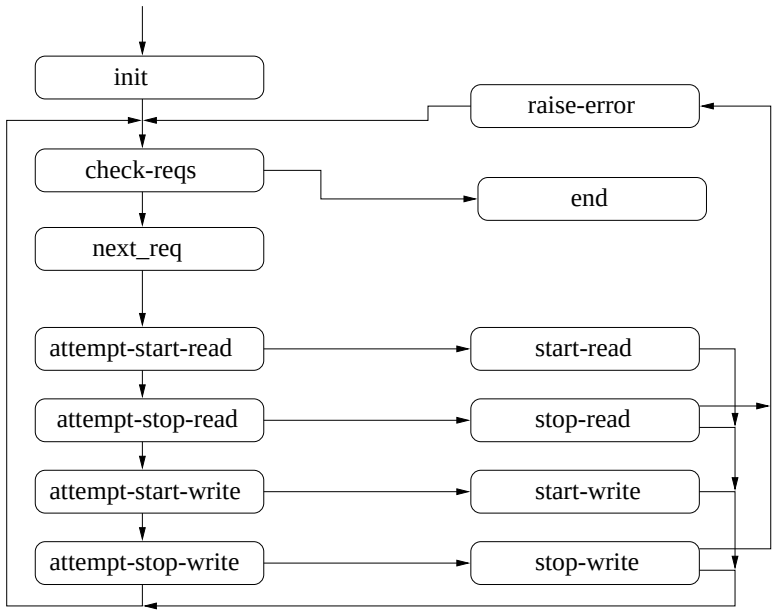


Figure 7: Read-write control flowchart

*where $\pi_s$ is the execution trace of $p_s$ running with initial state $s_1$.*

## 5.1 Proposition-based slicing criterion

We now consider some technical points that will guide us in defining an appropriate extraction function. As stated above, we want to preserve the satisfaction of the formula $\psi$ yet remove as much irrelevant information from the original trace $\pi$ as possible. We have already discussed the situation where certain variables' values can be eliminated from the states in a trace $\pi$ because they do not influence the satifaction of the formula $\psi$ under $\pi$. What is important in this is that we have used purely syntactic information (the set of variables mentioned in $\psi$) to reduce the state space.

Let's explain this reduction in more general terms. Consider a trace

$$\pi \;=\; s_1, s_2, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n.$$

Assume that the state transition $s_i, s_{i+1}$ does not influence the statisfaction of $\psi$. Formally, $\pi \models \psi$ iff $\pi_s \models \psi$ where $\pi_s$ is the compressed trace (the transition $s_i, s_{i+1}$ has been compressed)

$$\pi_s \;=\; s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n.$$

Another view of the change from $\pi$ to $\pi_s$ is that the action $\alpha$ that causes the change from $s_{i-1}$ to $s_i$ and the action $\alpha'$ that causes the change from $s_i$ to $s_{i+1}$ have been combined into an action $\alpha''$ that moves from $s_{i-1}$ to $s_{i+1}$. Intuitively, the formula $\psi$ "doesn't need to know" about the intermediate state $s_i$. For example, the irrelevant transition might be an assignment to an irrelevant variable, or a transition between nodes $[l.i]$ and $[l.(i+1)]$ not mentioned in $\psi$.

What is the technical justification for identifying compressible transitions using a purely syntactic examination of only the propositions in a formula $\psi$? The answer lies in the fact that, for the temporal operators we are treating, state transitions that don't change the satisfication of the primitive propositions of the formula $\psi$ do not influence the satisfaction of $\psi$ itself. This means that we can justify many trace compressions by reasoning about only single transitions and satisfaction of primitive propositions. We will see below that this property does not hold when one includes other temporal operators such as the *next state* operator $\circ$.

We now formalize these notions. The following definition gives a notion of proposition invariance with respect to a particular transition.

**Definition 12 (P-stuttering transition)** *Let $P$ be a primitive proposition, and let*

$$\pi \;=\; s_1, s_2, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n.$$

*The transition $s_i \rightarrow s_{i+1}$ is said to be* $P$-stuttering *when*

$$[\![P]\!]s_i \;=\; [\![P]\!]s_{i+1}.$$

*If $\mathcal{P}$ is a set of primitive propositions and for each proposition $P \in \mathcal{P}$ the transition $s_i \rightarrow s_{i+1}$ is $P$-stuttering, then the transition is said to be $\mathcal{P}$-stuttering.*

The following lemma states that the satisfaction of a formula $\psi$ containing primitive propositions $\mathcal{P}$ is invariant with respect to expansion and compression of $\mathcal{P}$-stuttering steps.

**Lemma 1** *Let $\psi$ be a formula and let $\mathcal{P}$ be the set of primitive propositions appearing in $\psi$. For all traces*

$$\pi \;=\; s_1, s_2, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n,$$

*if $s_i \rightarrow s_{i+1}$ is $\mathcal{P}$-stuttering, then*

$$\pi \models \psi \quad \textit{iff} \quad \pi_s \models \psi$$

*where*

$$\pi_s \;=\; s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n.$$

This lemma fails when one includes the next state operator $\circ$ [23] with the following semantics

$$s_1, s_2, ... \models \circ\psi \quad \text{iff} \quad s_2, ... \models \psi.$$

For example, consider the trace

$$\pi = ([l.1], \sigma_1), ([l.2], \sigma_2), ([l.3], \sigma_3), ([l.4], \sigma_4).$$

Let $P$ be the proposition $[l.3]$ and note $\pi \models \circ\circ P$. Now $([l.1], \sigma_1) \rightarrow ([l.2], \sigma_2)$ is $P$-stuttering ($P$ is false in both states), but compressing the transition to obtain

$$\pi_s = ([l.2], \sigma_2), ([l.3], \sigma_3), ([l.4], \sigma_4).$$

does not preserve satification of the formula (*i.e.*, $\pi_s \not\models \circ\circ P$).

Intuitively, the next state operator allows one to count states, and thus any attempt to optimize by compressing transitions in this setting is problematic. For this reason, some systems like SPIN [23] do not guarantee that the semantics of $\circ$ will be preserved during, *e.g.*, partial-order reduction optimizations.

Given a formula $\psi$ where $\mathcal{P}$ is the set of propositions in $\psi$, we now want to define an extraction function that guarantees that transitions that are not $\mathcal{P}$-stuttering are preserved in residual program traces.

- For variable propositions $P = [x \; rop \; c]$, observe that only definitions of the variable $x$ may cause the variable to change value (*i.e.*, cause a transition to be non-$P$-stuttering). This suggests that for each proposition $[x \; rop \; c]$ in a given specification $\psi$, each assignment to $x$ should be included in the residual program. Moreover, $x$ should be considered relevant.
- For a proposition $P = [l.i]$, entering or leaving CFG node $[l.i]$ can cause the proposition to change value (*i.e.*, cause the transition to be non-$P$-stuttering). One might imagine that we only need the slice to include the statement $[l.i]$ for each such proposition in the formula. However, it is possible that compression might remove all intermediate nodes between two occurences of the node $[l.i]$. This, as well as similar situations, do not preserve that state changes associated with entering and exiting the node. Therefore, in addition to the node $[l.i]$, we must ensure that all immediate successors and all immediate predecessors of $[l.i]$ are included in the slice.

Based on these arguments, we define an extraction function as follows.

**Definition 13 (Proposition-based extraction)** *Given a program $p$ and specification $\psi$, let $V$ be the set of all program variables occurring in $\psi$, and let $\{n_1, \ldots, n_k\}$ be the set of all nodes that contain assignments to variables in $V$ unioned with the set $N_P$ of all nodes appearing in node propositions of $\psi$ together the successors and predecessors of each node in $N_P$. Then* $\mathsf{extract}(\psi) \stackrel{\text{def}}{=} \{(n_1, V), \ldots, (n_k, V)\}$.

**Property 1** *The extraction function* extract *satisfies Requirement 1.*

As an example,

$$\mathsf{extract}(\Box([\texttt{start-read.1}] \Rightarrow [\texttt{WriterPresent=0}]))$$

yields the following criterion $C_1$:

$$\left\{ \begin{array}{l} ([\texttt{start-read.1}], \{\texttt{WriterPresent}\}), \\ ([\texttt{attemp-start-read.1}], \{\texttt{WriterPresent}\}), \\ ([\texttt{start-read.2}], \{\texttt{WriterPresent}\}), \\ ([\texttt{init.3}], \{\texttt{WriterPresent}\}), \\ ([\texttt{start-write.1}], \{\texttt{WriterPresent}\}), \\ ([\texttt{stop-write.1}], \{\texttt{WriterPresent}\}) \end{array} \right\}.$$

Here, the first three lines of the criterion are the [`start-read`.1] node mentioned in the formula, along with its predecessor and successor. The last three lines are the nodes where `WriterPresent` is assigned a value.

Figure 8 presents the resulting slice. The slice is identical to the original program except that the variable `ErrorFlag` and the block `raise-error` disappear from the program. Thus, slicing automatically detects what our abstracting methodology yielded in the previous section: for the given specification, only `ErrorFlag` is irrelevant. The previous conditional jumps in `stop-read` and `stop-write` to `raise-error` are replaced with unconditional jumps to `check-req`. In this case, the slicing algorithm has detected that the nodes in the `raise-error` block are irrelevant, and the conditional jumps are replaced with unconditional jumps to the node where the true and false paths leading out of the conditionals meet.

As a second example, consider the specification $\psi = \Diamond[\texttt{check-reqs.1}]$ ([`check-reqs`.1] is eventually executed). In this case $\mathsf{extract}(\psi)$ yields the criterion $C_2$:

$$\left\{ \begin{array}{l} ([\texttt{check-reqs.1}], \emptyset), \\ ([\texttt{init.5}], \emptyset), \\ ([\texttt{end.1}], \emptyset), \\ ([\texttt{next-req.1}], \emptyset) \end{array} \right\}.$$

Here, the lines of the criterion are the [`check-reqs`.1] node mentioned in the formula, along with its predecessor and successors. Since there are no variable propositions in the specification, no variables are specified as relevant in the criterion.

Figure 9 presents the resulting slice. It is obvious that the residual program is sufficient for verifying the reachability of [`check-req`.1] as given in the specification. All variables are eliminated except `reqs` which appears in the test at [`check-reqs`.1]. Even though it not strictly necessary for verifying the property, this conditional is retained by the slicing algorithm since it is control-dependent upon itself. In addition, the slicing criterion dictates that the node [`next-req`.1] should be in the slice. However, since the assignment at this node does not assign to a relevant variable, the assignment can be replaced with **skip**. Finally, the jump to `check-reqs` at node [`next-req`.3] in the residual program is the result of chaining through a series of trivial **goto**'s during post-processing.

## 6 Future Work

The previous criteria have considered individual propositions. Many property specifications, however, describe states using multiple propositions or state relationships between states that are characterized by different propositions. In this section, we give some informal suggestions about how the structure of these complex specifications may be exploited to produce refined slicing criterion.

```
(reqs) (init)
init:
  req := 0;                                [1]
  ActiveReaders := 0;                      [2]
  WriterPresent := 0;                      [3]

  goto check-reqs;                         [5]

check-reqs:                                              end:
  if (null? reqs)                          [1]           return;                                  [1]
     then end
     else next-req;

next-req:
  req := (car reqs);                       [1]
  reqs := (cdr reqs);                      [2]
  goto attempt-start-read;                 [3]

attempt-start-read:                                      start-read:
  if (req=1 and WriterPresent=0)           [1]           ActiveReaders := ActiveReaders+1;  [1]
     then start-read                                     goto check-reqs;                   [2]
     else attempt-stop-read;

attempt-stop-read:                                       stop-read:
  if (req=2 and ActiveReaders>0)           [1]           ActiveReaders := ActiveReaders-1;  [1]
     then stop-read                                      goto check-reqs;                   [2]
     else attempt-start-write;


attempt-start-write:                                     start-write:
  if (req=3 and ActiveReaders=0            [1]           WriterPresent := 1;                [1]
            and WriterPresent=0)                         goto check-reqs;                   [2]
     then start-write
     else attempt-stop-write;

attempt-stop-write:                                      stop-write:
  if (req=4 and WriterPresent=1)           [1]           WriterPresent := 0;                [1]
     then stop-write else check-reqs;                    goto check-reqs;                   [2]
```

Figure 8: Slice of read-write control program with respect to $C_1$

```
(reqs) (init)
init:
  goto check-reqs;                         [5]

check-reqs:                                              end:
  if (null? reqs)                          [1]           return;                                  [1]
     then end
     else next-req;

next-req:
  skip;                                    [1]
  reqs := (cdr reqs);                      [2]
  goto check-reqs;                         [3]
```

Figure 9: Slice of read-write control program with respect to $C_2$

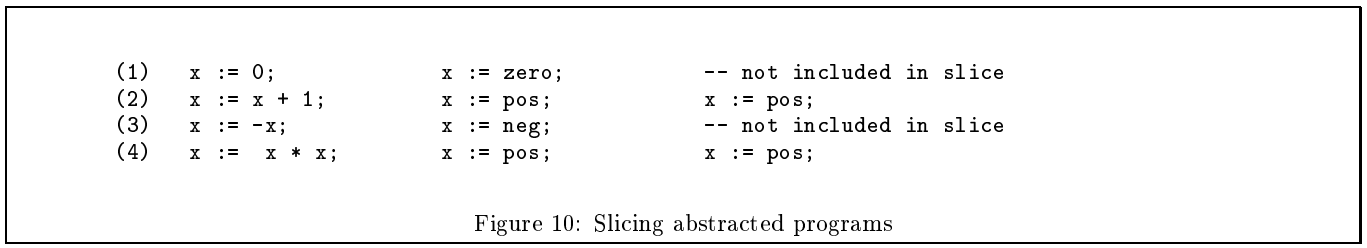


Figure 10: Slicing abstracted programs

Consider a simple conjunction of propositions appearing in an eventuality specification

$$\Diamond([1.1] \wedge [x{=}0]).$$

Rather than slicing on the propositions separately, we can use the semantics of $\wedge$ to refine the slicing criterion. For this property, we are not interested in all assignments to x but only those that can influence the value at [1.1]. Thus, our slicing criterion would be: $\mathsf{extract}(\psi) \stackrel{\text{def}}{=} \{([1.1], x)\}$. This approach generalizes in any setting where the program point proposition occurs positively with any number of variable propositions as conjuncts.

Thus far, we have considered slicing as a prelude to ABPS. Application of ABPS can, however, reveal semantic information about variable values in statement syntax, thereby making it available for use in slicing.

Figure 10 illustrates a sequence of assignments to x, on the left, and the abstracted sequence assignments, in the middle, resulting from binding of the classic signs AI [1] to x during ABPS. In such a situation we can determine transitions in the values of propositions related to x (e.g., x>0) syntactically.

Consider a response property [15] of the form

$$\Box(\psi_1 \Rightarrow \Diamond\psi_2)$$

Our proposition slicing criterion would be based on solely on $\psi_1$ and $\psi_2$. As with the conjunctions above, we observe two facts about the structure of this formula that can be exploited.

1. Within the $\Box$ is an implication, thus we need only reason about statements that cause the value of $\psi_1$ to become true (since false values will guarantee that the entire formula is true).

2. Since the right-hand side of the $\Rightarrow$ is a $\Diamond$, we need only reason about the first statement, in a sequence of statements, that causes $\psi_2$ to become true.

The right column of Figure 10 illustrates the effect of applying observation 1 to eliminate assignments that do not cause a positive transition in $\psi_1 = x > 0$ from the sliced program. Note that if a proposition involving x appears in $\psi_2$ then the slicing criterion may be expanded to include additional statements.

In addition, a program point where $\psi_1$ holds which is post-dominated by a point at which $\psi_2$ holds need not be considered for the purpose of checking response, since the existence of this relationship implies that the response holds for this occurrence of $\psi_1$.

Observation 2 can be exploited using post-domination information. A program point where $\psi_2$ holds which is post-dominated by another point where $\psi_2$ holds does not need to be included in the slice. This is because only one program point at which $\psi_2$ holds is required on any path for the $\Diamond$ formula to become true. Thus, any post-dominated $\psi_2$ nodes may be eliminated.

This refined slicing criteria defined above requires the use of auxiliary information, such as post-domination information, that needs to be available prior to slicing. While the cost of gathering this information and processing it to compute slicing criteria may be non-trivial, it will be dominated by the very high cost of performing model checking on the sliced system. In most cases, the cost of reducing the the size of the system presented to the model checker will be more than offset by reduced model check time.

We have discussed two refined criteria based on structural properties of the formula being checked. Similar refinements can be defined for a number of other classes of specifications including precedence and chain properties [15]. These refinements use essentially the same information as described above for response properties; precedence properties require dominator rather than post-dominator information.

We note that the refined response criteria is applicable only when the property to be checked is of a very specific form, even slight variations in the structure of the formula may render the sliced program unsafe. A recent survey of property specification for finite-state verification showed that response and precedence properties of the form described above occur quite frequently in practice [16]; 48% of 555 real-world specifications fell into these two categories. For this reason, we believe that the effort to define a series of special cases for extracting criteria based on formula structure is justified despite its apparent narrowness.

## 7 Related Work

Program slicing was developed as a technique for simplifying programs for debugging and for identifying parts of programs that can execute in parallel [32]. Since its development the concept of slicing has been applied to a wide variety of problems including: program understanding, debugging, differencing, integration, and testing [31]. In these applications, it is crucial that the slice preserve the exact execution semantics of the original program with respect to the slicing criterion. In our work, we are interested only preserving the ability to successfully model check properties that are correct; this weakening allows for the refinement of slicing criteria based on the property being checked.

Slicing has been generalized to other software artifacts [30] including: attribute grammars, requirements models [22] and formal specifications [4]. Cimitile et. al. [6] use Z specifications to define slicing criteria for identifying reusable code in legacy systems. In their work, they use a combination of symbolic execution and theorem proving to process

the specifications and derive the slicing criteria. In contrast, we identify necessary conditions for sub-formula of commonly occurring patterns of specifications and use those conditions to guide safe refinement of our basic proposition slicing criteria.

Our work touches on the relationship between program specialization and slicing. We use slicing as a prelude to specialization and suggest that abstraction-based specialization may reveal semantic features in the residual program's syntax that could be used by refined slicing criteria. Reps and Turnidge [29] have studied this relationship from a different perspective. They show that while similar the techniques are not equivalent; not all slicing transformations can be achieved with specialization and vice versa.

While slicing can be viewed as a state-space reduction technique it has a number of important theoretical and practical differences from other reduction techniques appearing in the literature. State-space reduction, such as [17], preserve correctness with respect to a specific class of correctness properties. In contrast, our approach to slicing based on criteria extracted from formulae yields compressed traces that contain the state changes relevant to propositions contained in the temporal logic formula. Our approach yields programs that remain both sound and complete with respect to property checking. This is in sharp contrast to the many abstraction techniques developed in the literature (*e.g.*[7]) which sacrifice completeness for tractability. Finally, even though significant progress has been made on developing algorithms and data structures to reduce model checking times, such as OBDDs [2], those techniques should be seen as a complement to slicing. If slicing removes variables from the system that do not influence the behavior to be checked then the model checker will run faster regardless of the particular implementation techniques it employs.

## 8 Conclusion

We have presented a variation of program slicing for a simple imperative language. We have shown how slicing criteria can be defined that guarantee the preservation of model check semantics for LTL specifications in the sliced program. We have implemented a prototype tool that performs this slicing and experimented with a number of examples. Based on this work we are scaling up the prototype to handle significantly more complex features of programs including: structured data, treatment of procedures, and multi-threaded programs that communicate through shared data. While these extensions are non-trivial they will build of the solid base laid out in the work reported in this paper.

## Acknowledgements

Thanks to James Corbett, Michael Huth, and David Schmidt for several very illuminating discussions. Thanks also to Hongjun Zheng for helpful comments on an earlier draft.

## References

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[3] T. Cattel. Process control design using spin. In *Proceedings of the First SPIN Workshop*, October 1995.

[4] J. Chang and D.JH. Richardson. Static and dynamic specification slicing. In *Proceedings of the Fourth Irvine Software Symposium*, April 1994.

[5] A. Cimatti, F. Giunchiglia, G. Mongardi, F. Torielli, and P. Traverso. Model checking safety critical software with spin: an application to a railway interlocking system. In *Proceedings of the Third SPIN Workshop*, April 1997.

[6] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: A case study. In Gianluigi Caldiera and Keith Bennett, editors, *Proceedings of the International Conference on Software Maintenance*, pages 124–133, Washington, October 1995. IEEE Computer Society Press.

[7] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[8] J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), March 1996.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[10] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[11] M.B. Dwyer, J. Hatcliff, and M. Nanda. Using partial evaluation to enable verification of concurrent software. *ACM Computing Surveys*, 30(3es), Sept, 1998.

[12] M.B. Dwyer and C. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 1998.

[13] M.B. Dwyer and C. Pasareanu. Model checking generic container implementations. Technical Report 98-10, Kansas State University, Department of Computing and Informatio n Sciences, 1998.

[14] M.B. Dwyer, C. Pasareanu, and J. Corbett. Translating ada programs for model checking : A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Informatio n Sciences, 1998.

[15] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.

[16] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999. (to appear).

[17] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Second Workshop on Computer Aided Verification*, pages 417–428, July 1991.

[18] C. Gomard and N.D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[19] J. Hatcliff. An introduction to partial evaluation using a simple flowchart language. In John Hatcliff, Peter Thiemann, and Torben Mogensen, editors, *Proceedings of the 1998 DIKU International Summer School on Partial Evaluation*, number (to appear) in Tutorials in Computer Science, Copenhagen, Denmark, June 1998.

[20] J. Hatcliff, M.B. Dwyer, and S. Laubach. Staging static analysis using abstraction-based program specialization. In *LNCS 1490*. Principles of Declarative Programming 10th International Symposium, PLILP'98, September 1998.

[21] J. Hatcliff, M.B. Dwyer, Shawn Laubach, and Nanda Muhammad. Specializing configurable systems for finite-state verification. Technical Report 98-4, Kansas State University, Department of Computing and Information Sciences, 1998.

[22] Mats P. E. Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 450–467. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, September 1997.

[23] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[24] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 1999. To appear.

[25] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[26] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajiani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[28] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[29] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429. Berlin: Springer-Verlag, 1996.

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

[31] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995. Surveys the state-of-the-art in program slicing and gives many references to the literature.

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

[33] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.