Abstract Slicing: A New Approach to Program Slicing Based on Abstract Interpretation and Model Checking

Hyoung Seok Hong
CIISE
Concordia University
Montreal, Quebec, Canada
hshong@ciise.concordia.ca

Insup Lee, Oleg Sokolsky
Dept. of CIS
University of Pennsylvania
Philadelphia, PA, USA
{lee,sokolsky}@cis.upenn.edu

Abstract

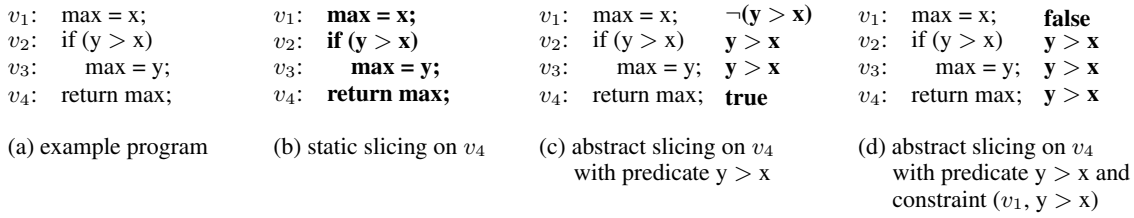
This paper proposes a new approach to program slicing based on abstract interpretation and model checking. First, the notion of abstract slicing is introduced. Abstract slicing extends static slicing with predicates and constraints by using as the program model an abstract state graph, which is obtained by applying predicate abstraction to a program, rather than a flow graph. This leads to a program slice that is more precise and smaller than its static counterpart. Second, a method for performing abstract slicing is developed. It is shown that abstract slicing can be reduced to a least fixpoint computation over formulas in the branching time temporal logic CTL. This enables one to use symbolic model checkers for CTL as an efficient computation engine for abstract slicing. A prototype implementation and experimental results are reported demonstrating the feasibility of the approach.

1 Introduction

Program slicing is a program analysis technique that has been proven to be useful in a variety of software engineering applications such as program debugging, testing, understanding, maintenance, metrics, and reuse. For detailed surveys, we refer the interested readers to [4, 12, 33]. Recently, program slicing has also been applied to state space reduction for formal verification[19] and test generation[5]. The original approach to program slicing was introduced by Weiser[35]. The program slice with respect to a program point, called slicing criterion, is defined as a reduced, executable program whose behavior is equivalent to that of the original program with respect to the program point. A set of data flow equations over a flow graph is solved to produce a program slice. Ottenstein and Ottenstein[29] introduced an alternative approach. The program slice with respect to a program point is defined as the parts of a program that directly or indirectly affect the program point. The direct-affect relation is computed using a flow graph and the indirect-affect relation is computed using a program dependence graph. These approaches are called static slicing since they employ information statically available from the flow graph of a program.

This paper addresses two shortcomings of static slicing. First, static slicing uses a flow graph as the program model. It is often the case that a flow graph is a very coarse abstraction since it preserves only the control flow and does not respect the values of variables. This results in an imprecise program slice that only determines, for every program point, whether the program point affects the slicing criterion or not. It is desirable to answer a more complicated question: *for every program point, under which variable values does the program point affect the slicing criterion?* Second, static slicing produces a program slice by considering all possible executions of a program and hence each program point may affect many other program points. This results in a large program slice whose size of a program slice is not significantly small in comparison to the size of the original program. It is desirable to answer: *for every program point, does the program point affect the slicing criterion if we are only interested in certain executions of a program rather than all possible ones?*

To remedy these shortcomings, this paper proposes a new approach to program slicing based on abstract interpretation[10] and model checking[9]. Our approach is a specialization of Schmidt and Steffen's framework for program analysis[32] in which an abstraction of a program is used as the program model and model checking is performed against the abstraction. The main contributions of the paper are twofold. First, we introduce the notion of abstract slicing. We illustrate the main ideas of abstract slicing through a simple C program shown in Figure 1.(a). Static slicing shows that the program points v_1, v_2, v_3, v_4 affect v_4



(a) example program

(b) static slicing on v_4

(c) abstract slicing on v_4 with predicate y > x

(d) abstract slicing on v_4 with predicate y > x and constraint (v_1, y > x)

(Figure 1.(b)). Abstract slicing shows the values of predicates under which v_1, v_2, v_3, v_4 affect v_4 (Figure 1.(c)). We observe that v_1 affects v_4 when the predicate $y > x$ is not satisfied, while v_2 and v_3 affect v_4 when $y > x$ is satisfied. Abstract slicing also supports the use of constraints (Figure 1.(d)). Suppose that we are only interested in the executions satisfying the constraint $y > x$ at v_1. In this case, abstract slicing shows that v_1 cannot affect v_4. Put together, abstract slicing extends static slicing with predicates and constraints. Predicates are used to respect the values of variables and constraints are used to limit the scope of analysis. Static slicing can be regarded as a special case of abstract slicing in which no predicates and constraints are used.

To extend static slicing with predicates and constraints, we incorporate predicate abstraction into static slicing. Predicate abstraction[14] is a special form of abstract interpretation in which a set of predicates over the program's variables is used to construct a finite and sound abstraction of the program. During the last years, several tools such as SLAM[2], BLAST[20], and MAGIC[7] have demonstrated that predicate abstraction can be effectively used for program verification. In this paper, we show that program slicing can also benefit from predicate abstraction. We use as the program model an abstract state graph, which is obtained by applying predicate abstraction to a program with predicates and constraints, rather than a flow graph. This leads to a program slice that is more precise and smaller than its static counterpart, answering the two questions: for every program point, under which predicate values does the program point affect the slicing criterion? and does a program point affect the slicing criterion if we are only interested in the constrained executions?

Second, we develop a method for performing abstract slicing. Conventional static slicing methods based on data flow equations[35] and program dependence graphs[29] can be extended for abstract slicing. These methods, however, have problems with scalability due to the state explosion problem, that is, the size of an abstract state graph grows exponentially in the number of predicates. Rather, we formulate abstract slicing in terms of symbolic model checking[28] that has been shown to be effective for controlling the state explosion problem. We show that abstract slicing can be reduced to a least fixpoint computation over formulas in the branching time temporal logic CTL[13]. This enables one to use symbolic model checkers for CTL such as SMV[28] and NuSMV[8] as an efficient computation engine for abstract slicing.

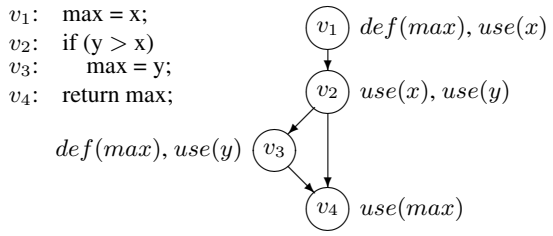
The main advantages of establishing a connection between abstract slicing and model checking may be summarized as follows. First, we need to focus on only high-level specifications of abstract slicing written in temporal logic. All the details about the implementation of least fixpoint computations are hidden in model checkers. Second, we can apply abstract slicing in a language-independent manner in the sense that the temporal logic formulas employed in our method are applicable with non-essential modifications to various programming languages. Third, we can apply abstract slicing to programs whose size and complexity are limited by the capabilities of current model checkers. More importantly, we can enjoy the continuing and rapid advances in the model checking community.

The remainder of the paper is organized as follows. Section 2 recalls the basics of static slicing, predicate abstraction, and symbolic model checking. Section 3 and Section 4 present abstract slicing and a method for performing abstract slicing, respectively. Section 5 reports a prototype implementation and experimental results. Section 6 compares our approach to previous work on program slicing. Finally, Section 7 concludes the paper with a discussion of future work.

2 Background

For the remainder of the paper, we use V to denote the set of program points of a program and partition V into two disjoint subsets V_{stmt} and V_{branch}. A program point in V_{stmt} is a simple statement such as assignment, read, or write. A program point in V_{branch} is the branch condition of a conditional or repetitive statement. There are two distinguished program points $v_s \in V$ and $v_f \in V$ that are the start point and final point of the program, respectively.

The *flow graph* G of a program is a directed graph whose nodes correspond to program points and arcs correspond to possible flow of control between program points. A simple statement has only one successor, while a branch condition has two successors. We label a program point v with definitions and uses of variables: $\{def(x) \mid x \text{ is defined at } v\} \cup \{use(x) \mid x \text{ is used at } v\}$. Figure 2 shows the flow graph of the example program in Figure 1.



Let v and v' be program points. We say that v *directly data-affects* v' (or equivalently, v' is *directly data-dependent* on v) if there is a variable x such that x is defined at v, x is used at v', and there is a path $v, v_1, ..., v_n, v'$ such that for every $1 \leq i \leq n$, x is not defined at v_i. In Figure 2, v_1 directly data-affects v_4 and v_3 directly data-affects v_4. We say that v' *postdominates* v if every path from v to v_f contains v' and that v *directly control-affects* v' (or equivalently, v' is *directly control-dependent* on v) if v has two successors v_1 and v_2 such that v' postdominates v_1 but v' does not postdominate v_2. In Figure 2, v_2 directly control-affect v_3 since v_3 postdominates itself and does not postdominate v_4. The *affect* relation between program points is the transitive closure of the union of the direct data-affect and direct control-affect relation.

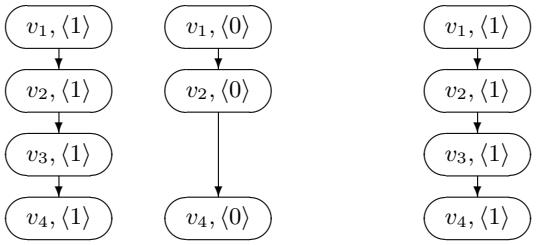
Static slicing is defined with respect to the flow graph of a program. We consider three forms of static slicing: backward slicing, forward slicing, and chopping[22]. A slicing criterion for backward slicing and forward slicing is a program point v[1]. The *backward slice* with respect to v consists of the program points that affect v. Forward slicing is the dual of backward slicing. The *forward slice* with respect to v consists of the program points that are affected by v. Chopping is a combination of backward slicing and forward slicing. A slicing criterion for chopping is a pair (v, v') of program points. The *chop* with respect to (v, v') consists of the program points that are affected by v and affect v'.

[1]For the sake of presentation, we use a program point as a slicing criterion. The results of this paper can be immediately applied to more elaborated slicing criteria, e.g., a pair (v, X) consisting of a program point v and a set X of variables[35].

Given a program and a set of predicates over the program's variables, predicate abstraction constructs a finite and sound abstraction of the program. Intuitively, an abstraction of a program is sound if every execution of the program has a corresponding execution in the abstraction.

We first adopt the following definitions. Let $\{p_1, ..., p_n\}$ be a set of predicates over the program's variables and $\{b_1, ..., b_n\}$ be a set of boolean variables such that for every $1 \leq i \leq n$, b_i represents p_i. A valuation σ over $\{b_1, ..., b_n\}$ represents the predicate $p'_1 \wedge ... \wedge p'_n$ where for every $1 \leq i \leq n$, $p'_i = p_i$ if $\sigma(b_i) = 1$ and $p'_i = \neg p_i$ if $\sigma(b_i) = 0$. For notational convenience, we identify a valuation σ with the boolean vector $\langle \sigma(b_1), ..., \sigma(b_n) \rangle$. For example, in Figure 3.(a), $\langle 1 \rangle$ and $\langle 0 \rangle$ represent the predicates $y > x$ and $\neg(y > x)$, respectively. For a valuation σ_c over the program's variables, σ is an abstraction of σ_c if σ_c satisfies $p'_1 \wedge ... \wedge p'_n$. For example, define $\sigma_{x,y}$ as a valuation over $\{x, y\}$ that maps x to 0 and y to 1. $\langle 1 \rangle$ is an abstraction of $\sigma_{x,y}$. For a predicate $cond$ over the program's varibles, σ satisfies $cond$ if there is a valuation σ_c such that σ is an abstraction of σ_c and σ_c satisfies $cond$. For example, $\langle 1 \rangle$ satisfies $y > x$.

We define the *abstract state graph* $G_{\{p_1,...,p_n\}}$ of a program with predicates $\{p_1, ..., p_n\}$ as follows. A node of $G_{\{p_1,...,p_n\}}$ is a pair (v, σ) where the control part is a program point v and the data part is a valuation σ over $\{b_1, ..., b_n\}$. A node (v, σ) is a start node if $v = v_s$ and is a final node if $v = v_f$. Figure 3.(a) shows $G_{\{y>x\}}$ of the example program in Figure 2.



(a) the abstract state graph with predicate y > x

(b) the abstract state graph with predicate y > x and constraint (v_1, y > x)

When determining whether there is an arc from (v, σ) to (v', σ'), we consider two cases. The first case is that v is a simple statement. There is an arc from (v, σ) to (v', σ') if there are two valuations σ_c and σ'_c over the program's variables such that (i) σ is an abstraction of σ_c, (ii) σ' is an abstraction of σ'_c, and (iii) the execution of v with σ_c leads to v' and σ'_c. For example, in Figure 3.(a), using $\sigma_c = \sigma'_c$

= $\sigma_{x,y}$, we can show that there is an arc from $(v_1, \langle 1 \rangle)$ to $(v_2, \langle 1 \rangle)$. The second case is that v is a branch condition $cond$. Let v_1 be the 'then' successor and v_2 be the 'else' successor of v. There is an arc from (v, σ) to (v_1, σ) if σ satisfies $cond$ and there is an arc from (v, σ) to (v_2, σ) if σ satisfies $\neg cond$. For example, consider v_2 whose true successor is v_3 and false sucessor is v_4. There is an arc from $(v_2, \langle 1 \rangle)$ to $(v_3, \langle 1 \rangle)$ since $\langle 1 \rangle$ satisfies the branch condition $y > x$, while there is no arc from $(v_2, \langle 1 \rangle)$ to $(v_4, \langle 1 \rangle)$.

We define the *abstract state graph* $G^{\{c_1,...,c_m\}}_{\{p_1,...,p_n\}}$ of a program with predicates $\{p_1, ..., p_n\}$ and constraints $\{c_1, ..., c_m\}$ as follows. A constraint is a precondition associated with a program point. The set of nodes of $G^{\{c_1,...,c_m\}}_{\{p_1,...,p_n\}}$ is a subset of nodes (v, σ) of $G_{\{p_1,...,p_n\}}$ such that if c_i is a constraint associated with v, then σ satisfies c_i. The set of arcs of $G^{\{c_1,...,c_m\}}_{\{p_1,...,p_n\}}$ is obtained by restricting the domain and range of the set of arcs of $G_{\{p_1,...,p_n\}}$ to the nodes of $G^{\{c_1,...,c_m\}}_{\{p_1,...,p_n\}}$. Figure 3.(b) shows $G^{\{y>x\}}_{\{y>x\}}$ of the example program. Due to the constraint $y > x$ at v_1, we have that $(v_1, \langle 1 \rangle)$ is a node while $(v_1, \langle 0 \rangle)$ is not.

CTL is a branching time temporal logic that has been extensively used in symbolic model checking. We give a brief introduction to CTL and refer to [13] for the formal syntax and semantics for CTL. Formulas in (future) CTL are built from a set AP of atomic propositions, standard boolean operators, path quantifiers **E** (for some path) and **A** (for all paths), and modal operators **X** (nexttime) and **U** (until).

The semantics of CTL is defined with respect to a Kripke structure (Q, Q_0, L, R) where Q is a set of states, $Q_0 \subseteq Q$ is the set of initial states, L: $Q \rightarrow 2^{AP}$ labels each state with atomic propositions, and $R \subseteq Q \times Q$ is the total transition relation. For an infinite path $\pi = q_0, q_1, ...$ of a Kripke structure, we use $\pi(i)$ to denote the i-th element of π. For a state q of Kripke structure M and a formula f, we write $q \models f$ to mean that q satisfies f. We will make use of formulas of the form $\mathbf{EX}f$ and $\mathbf{E}[f\mathbf{U}g]$ whose semantics is defined below.

- $q \models \mathbf{EX}f$ iff for some path π such that $\pi(i) = q$ for some $i \geq 0$, we have that $\pi(i+1) \models f$.
- $q \models \mathbf{E}[f\mathbf{U}g]$ iff for some path π such that $\pi(i) = q$ for some $i \geq 0$, we have that there is $j \geq i$ such that $\pi(j) \models g$ and $\pi(k) \models f$ for every $i \leq k < j$.

We will also make use of past modal operators $\mathbf{X}^-$ (yesterday) and $\mathbf{U}^-$ (since) that are the duals of **X** and **U**, respectively[13].

- $q \models \mathbf{EX}^- f$ iff for some path π such that $\pi(i) = q$ for some $i > 0$, we have that $\pi(i-1) \models f$.
- $q \models \mathbf{E}[f\mathbf{U}^- g]$ iff for some path π such that $\pi(i) = q$ for some $i \geq 0$, we have that there is $0 \leq j \leq i$ such that $\pi(j) \models g$ and $\pi(k) \models f$ for every $j < k \leq i$.

For a Kripke structure M and a formula f, the model checking problem is to find the set of states satisfying f: $\{q \mid q \models f\}$. Symbolic model checking finds the set of states satisfying a formula by computing the least (or greatest) fixpoint of a predicate transformer. For example, consider $\mathbf{EF}v$ which is the abbreviation of $\mathbf{E}[true\mathbf{U}v]$ expressing that v is reachable. The set of states satisfying $\mathbf{EF}v$ is equivalent to the least fixpoint of $\tau(Z) = v \vee \mathbf{EX}Z$. The fixpoint computation requires standard boolean operations, quantification over variables, and substitution of variables which can all be performed efficiently on binary decision diagrams (BDDs).

3 Abstract Slicing

Abstract slicing uses as the program model an abstract state graph rather than a flow graph. We first describe how we lift the affect relation from a flow graph to an abstract state graph. We then introduce three forms of abstract slicing.

It is straightforward to lift the direct data-affect relation from a flow graph to an abstract state graph. For two nodes (v, σ) and (v', σ') of an abstract state graph, we say that (v, σ) *directly data-affects* (v', σ') if there is a variable x such that x is defined at v, x is used at v', and there is a path (v, σ), (v_1, σ_1), ..., (v_n, σ_n), (v', σ') such that for every $1 \leq i \leq n$, x is not defined at v_i. In Figure 3.(a), we observe that $(v_1, \langle 1 \rangle)$ does not directly data-affect $(v_4, \langle 1 \rangle)$ since they are intervened by $(v_3, \langle 1 \rangle)$. We also observe that $(v_1, \langle 0 \rangle)$ directly data-affects $(v_4, \langle 0 \rangle)$. These two observations enable us to infer that v_1 directly data-affects v_4 only when the predicate $y > x$ is not satisfied.

In contrast to the direct data-affect relation, it is impossible to lift the direct control-affect relation. Recall that v directly control-affects v' if v' postdominates one of v's successors but v' does not postdominate the other successor. This definition cannot be lifted to an abstract state graph because a node (v, σ) of an abstract state graph does not necessarily have two successors depending on the valuation σ. Rather, we adopt the following definitions introduced by Podgurski and Clarke[30]: v' is a *postdominator* of v if v' postdominates v. In addition, if $v' \neq v$, v' is a *proper postdominator* of v. The *immediate postdominator* of v, denoted by $ipd(v)$, is the first proper postdominator

of v that occurs on every path from v to v_f. v *control-affects* v' (or equivalently, v' is *control-dependent* on v) if there is a path from v to v' not containing $ipd(v)$. In Figure 2, v_2 control-affects v_3 since the path v_2v_3 does not contain $ipd(v_2) = v_4$. In [30], it is shown that the control-affect relation is the transitive closure of the direct control-affect relation. We lift the control-affect relation from a flow graph to an abstract state graph. We say that (v, σ) *control-affects* (v', σ') if there is a path $(v, \sigma), (v_1, \sigma_1), ..., (v_n, \sigma_n), (v', \sigma')$ such that $v \neq ipd(v)$, $v' \neq ipd(v)$, and for every $1 \leq i \leq n$, $v_i \neq ipd(v)$. In Figure 3.(a), $(v_2, \langle 1 \rangle)$ control-affects $(v_3, \langle 1 \rangle)$.

We define the affect relation for an abstract state graph as the transitive closure of the union of the direct data-affect relation and control-affect relation. We overload the affect relation for program points. v affects (v', σ') if there is a node (v, σ) such that (v, σ) affects (v', σ'). (v, σ) affects v' if there is a node (v', σ') such that (v, σ) affects (v', σ'). v affects v' if there are two nodes (v, σ) and (v', σ') such that (v, σ) affects (v', σ').

We define three forms of abstract slicing: abstract backward slicing, abstract forward slicing, and abstract chopping that extend their static counterparts with predicates $\{p_1, ..., p_n\}$ and constraints $\{c_1, ..., c_m\}$. A slicing criterion for abstract backward slicing and abstract forward slicing is a triple $(v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$. The *abstract backward slice* with respect to $(v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$ consists of the reachable nodes of $G^{\{c_1,...,c_m\}}_{\{p_1,...,p_n\}}$ that affect v. The *abstract forward slice* with respect to $(v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$ consists of the reachable nodes of $G^{\{c_1,...,c_m\}}_{\{p_1,...,p_n\}}$ that are affected by v. A slicing criterion for abstract chopping is a quadruple $(v, v', \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$. The *abstract chop* with respect to $(v, v', \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$ consists of the reachable nodes of $G^{\{c_1,...,c_m\}}_{\{p_1,...,p_n\}}$ that are affected by v and affect v'.

As with other program analysis techniques based on abstractions, abstract slicing is sound (but not complete) in the sense that if a program has an execution along which an affection occurs, then its abstract state graph has a corresponding execution along which the affection occurs. Formally, if a program has an execution $(v_1, \sigma'_1), ..., (v_n, \sigma'_n)$ such that (v_1, σ'_1) affects (v_n, σ'_n), then its abstract state graph has an execution $(v_1, \sigma_1), ..., (v_n, \sigma_n)$ such that (v_1, σ_1) affects (v_n, σ_n) and for every $1 \leq i \leq n$, σ_i is an abstraction of σ'_i.

Let $V = \{v_1, ..., v_l\}$ be the set of program points. We partition the abstract backward slice ABS into the tuple $(ABS_1, ..., ABS_l)$ such that for every $1 \leq i \leq l$, $ABS_i \subseteq ABS$ is the set of nodes whose program point is v_i. Each ABS_i shows the values of $p_1, ..., p_n$ under which v_i affects v through the executions constrained by $c_1, ..., c_m$. For example, in Figure 3.(a),

$ABS_1 = \{(v_1, \langle 0 \rangle)\}$,
$ABS_2 = \{(v_2, \langle 1 \rangle)\}$,
$ABS_3 = \{(v_3, \langle 1 \rangle)\}$,
$ABS_4 = \{(v_4, \langle 0 \rangle), (v_4, \langle 1 \rangle)\}$.

In Figure 3.(b),

$ABS_1 = \emptyset$,
$ABS_2 = \{(v_2, \langle 1 \rangle)\}$,
$ABS_3 = \{(v_3, \langle 1 \rangle)\}$,
$ABS_4 = \{(v_4, \langle 1 \rangle)\}$.

Abstract forward slices and abstract chops can also be partitioned in the same way.

More refined slicing criteria may be used, e.g., $(v, pn + 1, \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$ where p_{n+1} is a predicate over the program's variables. The abstract backward slice then consists of the reachable nodes that affect (v, σ) such that σ satisfies p_{n+1}. We note that the abstract backward slice with respect to $(v, p_{n+1}, \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$ is equivalent to that with respect to $(v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\} \cup \{(v, p_{n+1})\})$ in which we have a new constraint (v, p_{n+1}) that imposes the precondition p_{n+1} on v.

4 Abstract Slicing as Symbolic Model Checking

A widely-used method for perfoming static slicing is to use a program dependence graph[29] whose nodes correspond to program points and arcs correspond to the union of the direct data-affect relation and direct control-affect relation. Analogously, abstract slicing may also be performed using a program dependence graph. In this case, nodes of a program dependence graph correspond to nodes of an abstract state graph and arcs correspond to the union of the direct data-affect relation and control-affect relation. However, the explicit construction of a program dependence graph for abstract slicing is impractical for large and complex programs due to the state explosion problem, that is, the size of an abstract state graph grows exponentially in the number of predicates. To control the state explosion problem, we reduce abstract slicing to a least fixpoint computation over CTL formulas[2] so that symbolic model checkers for CTL can be used as an efficient computation engine for abstract slicing.

[2]Or equivalently, abstract slicing can be reduced to the model checking problem of μ-calculus[25] since a least fixpoint over CTL formulas is a μ-calculus formula. In this paper, we use CTL for the sake of presentation and implementation.

We reduce abstract backward slicing to a least fixpoint computation over future CTL formulas. Let Z be a set of nodes of an abstract state graph $G^{\{c_1,...,c_m\}}_{\{p_1,...,p_n\}}$. Define predicate transformers **dda** and **ca** by

$$\mathbf{dda}(Z) = \bigvee_{x \in X} def(x) \wedge \mathbf{EXE}[\neg def(x) \mathbf{U} (use(x) \wedge Z)]$$

where X is the set of variables of the program,

$$\mathbf{ca}(Z) = \bigvee_{v \in V} v \wedge \mathbf{E}[\neg ipd(v) \mathbf{U} (\neg ipd(v) \wedge Z)].$$

For example, in Figure 3.(a),

$\mathbf{dda}(\{(v_4, \langle 1 \rangle)\})$ = $\{(v_3, \langle 1 \rangle)\}$ and
$\mathbf{ca}(\{(v_3, \langle 1 \rangle)\})$ = $\{(v_2, \langle 1 \rangle)\}$.

It is not hard to see that a node is in $\mathbf{dda}(Z)$ (resp. $\mathbf{ca}(Z)$) if and only if the node directly data-affects (resp. control-affects) some node in Z.

Define a predicate transformer $\mathbf{affect}[v]$ by

$$\mathbf{affect}[v](Z) = v \vee \mathbf{dda}(Z) \vee \mathbf{ca}(Z).$$

It is not hard to see that the least fixpoint of $\mathbf{affect}[v]$ is the set of nodes that affect v. By intersecting the least fixpoint and the set of reachable states, we obtain the abstract backward slice with respect to $(v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$.

The size of $\mathbf{dda}(Z)$ is linear in the number of variables and can be reduced using the simple observation that it is only necessary to consider the set X_{DU} of variables that are both defined and used by the program:

$$\mathbf{dda}(Z) = \bigvee_{x \in X_{DU}} def(x) \wedge \mathbf{EXE}[\neg def(x) \mathbf{U} (use(x) \wedge Z)].$$

The size of $\mathbf{ca}(Z)$ is linear in the number of program points and can be reduced using the simple observation that only a branch condition may control-affect a node:

$$\mathbf{ca}(Z) = \bigvee_{v \in V_{branch}} v \wedge \mathbf{E}[\neg ipd(v) \mathbf{U} (\neg ipd(v) \wedge Z)].$$

We reduce abstract forward slicing to a least fixpoint computation over past CTL formulas. Define $\mathbf{dda}^-$, $\mathbf{ca}^-$, and $\mathbf{affect}^-[v]$ by

$$\mathbf{dda}^-(Z) = \bigvee_{x \in X_{DU}} use(x) \wedge \mathbf{EX}^- \mathbf{E}[\neg def(x) \mathbf{U}^- (def(x) \wedge Z)],$$

$$\mathbf{ca}^-(Z) = \bigvee_{v \in V_{branch}} \mathbf{E}[\neg ipd(v) \mathbf{U}^- (\neg ipd(v) \wedge v \wedge Z)],$$

$$\mathbf{affect}^-[v](Z) = v \vee \mathbf{dda}^-(Z) \vee \mathbf{ca}^-(Z).$$

A node is in $\mathbf{dda}^-(Z)$ (resp. $\mathbf{ca}^-(Z)$) if and only if the node is directly data-affected (resp. are control-affected) by some node in Z. The least fixpoint of $\mathbf{affect}^-[v]$ is the set of nodes that are affected by v. By intersecting the least fixpoint and the set of reachable states, we obtain the abstract forward slice with respect to $(v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$.

Abstract chopping is a combination of abstract backward slicing and abstract forward slicing. The intersection of the least fixpoints of $\mathbf{affect}^-[v]$ and $\mathbf{affect}[v']$ is the set of nodes that are affected by v and affect v'. By intersecting the two least fixpoints of $\mathbf{affect}^-[v]$ and $\mathbf{affect}[v']$ and the set of reachable states, we obtain the chop with respect to $(v, v', \{p_1, ..., p_n\}, \{c_1, ..., c_m\})$.

We say that the above fixpoint computation is monolithic in the sense that the set Z is manipulated as a single entity. An alternative to the monolithic fixpoint computation is to partition Z into $(Z_1, ..., Z_l)$ according to the program points $V = \{v_1, ..., v_l\}$ and manipulate each Z_i separately during the fixpoint computation. This partition leads to the new definition of $\mathbf{affect}[v]$:

$$\begin{array}{l}\mathbf{affect}[v](Z_1, ..., Z_l) = \\ \quad (v_1 \Leftrightarrow v, ..., v_n \Leftrightarrow v) \vee \\ \quad (\mathbf{dda}_1(Z_1, ..., Z_l), ..., \mathbf{dda}_l(Z_1, ..., Z_l)) \vee \\ \quad (\mathbf{ca}_1(Z_1, ..., Z_l), ..., \mathbf{ca}_l(Z_1, ..., Z_l))\end{array}$$

where for every $1 \leq i \leq l$,

- $\mathbf{dda}_i(Z_1, ..., Z_l) = \bigvee_{1 \leq j \leq l} \mathbf{dda}_{ij}(Z_j)$
  where for every $1 \leq j \leq l$,
  $\mathbf{dda}_{ij}(Z_j) = v_i \wedge$
  $$\bigvee_{x \in DEF(v_i)} def(x) \wedge \mathbf{EXE}[\neg def(x) \mathbf{U} (use(x) \wedge Z_j)]$$
  where *DEF*(v_i) is the set of variables defined at v_i.
- $\mathbf{ca}_i(Z_1, ..., Z_l) = \bigvee_{1 \leq j \leq l} \mathbf{ca}_{ij}(Z_j)$
  where for every $1 \leq j \leq l$,
  $$\mathbf{ca}_{ij}(Z_j) = v_i \wedge \mathbf{E}[\neg ipd(v_i) \mathbf{U} (\neg ipd(v_i) \wedge Z_j)]$$

For example, in Figure 3.(a),

$\mathbf{dda}_3(\emptyset, \emptyset, \emptyset, \{(v_4, \langle 1 \rangle)\}) = \{(v_3, \langle 1 \rangle)\}$ and
$\mathbf{ca}_2(\emptyset, \emptyset, \{(v_3, \langle 1 \rangle)\}, \emptyset) = \{(v_2, \langle 1 \rangle)\}$.

A node is in $\mathbf{dda}_{ij}(Z_j)$ (resp. $\mathbf{ca}_{ij}(Z_j)$) if and only if the node has v_i as its program point and directly data-affects (resp. control-affects) some node in Z_j. It follows that a node is in $\mathbf{dda}_i(Z_1, ..., Z_l)$ (resp. $\mathbf{ca}_i(Z_1, ..., Z_l)$) if and only if the node has v_i as its program point and directly data-affects (resp. control-affects) some node in $Z_1 \vee ... \vee Z_l$. The least fixpoint of $\mathbf{affect}[v]$ is the tuple $(Z_1^f, ..., Z_l^f)$ such that for every $1 \leq i \leq l$, Z_i^f is the set of nodes that have v_i as the program point and affect v.

Since $\mathbf{dda}_{ij}$ and $\mathbf{ca}_{ij}$ are only concerned with nodes whose program point is either v_i or v_j, they can be optimized using the following observations: If v_i does not directly data-affect v_j, then (v_i, σ) does not directly data-affect (v_j, σ'). Similarly, if v_i does not control-affect v_j, then (v_i, σ) does not control-affect (v_j, σ'). The information of whether v directly data-affects (or control-affects) v' is statically available from the flow graph of a program. These observations lead to the new definition of $\mathbf{dda}_{ij}$ and $\mathbf{ca}_{ij}$:

- if v_i directly data-affects v_j,
  $\mathbf{dda}_{ij}(Z_j) = v_i \wedge$
  $$\bigvee_{x \in DEF(v_i)} def(x) \wedge \mathbf{EXE}[\neg def(x) \mathbf{U} (use(x) \wedge Z_j)]$$
  otherwise, $\mathbf{dda}_{ij}(Z_j) = false$.
- if v_i control-affects v_j,
  $$\mathbf{ca}_{ij}(Z_j) = v_i \wedge \mathbf{E}[\neg ipd(v_i) \mathbf{U} (\neg ipd(v_i) \wedge Z_j)]$$
  otherwise, $\mathbf{ca}_{ij}(Z_j) = false$.

5 Implementation and Experimentation

Our prototype implementation of abstract slicing consists of the following three tools.

- MAGIC: For a program written in C and a set of predicates over the program's variables, we constructed an abstract state graph using the predicate abstraction capability of MAGIC[7]. Since we used MAGIC, which does not accept constraints as input, as a black box, we considered programs with predicates only. Constraints are taken into account when translating an abstract state graph into input to the symbolic model checker NuSMV[8].
- MAGIC2SMV: We implemented the tool MAGIC2SMV that translates an abstract state graph constructed by MAGIC together with a set of constraints into input to NuSMV.
- NuSMV: For abstract backward slicing, we implemented both the monolithic fixpoint computation and partitioned fixpoint computation of $\mathbf{affect}[v]$ in NuSMV using the algorithms shown in Figure 4.(b) and Figure 4.(c), respectively. These algorithms are extensions of the fixpoint computation algorithm for the CTL formula $\mathbf{EF}v$ shown in Figure 4.(a). In order to compute $\mathbf{dda}(old)$, $\mathbf{ca}(old)$, $\mathbf{dda}_{ij}(old[j])$, and $\mathbf{ca}_{ij}(old[j])$, the algorithms use the capability of NuSMV to find the set of states satisfying CTL formulas of the form $\mathbf{EX}f$ or $\mathbf{E}[f \mathbf{U} g]$. For abstract forward slicing and abstract chopping, we do not have an implementation. To compute the least fixpoint of $\mathbf{affect}^-[v]$, we should be able to find the set of states satisfying past CTL formulas of the form $\mathbf{EX}^-f$ or $\mathbf{E}[f \mathbf{U}^- g]$, which is beyond the capability of current symbolic model checkers.

The goals of our experimentation are twofold. We wished to evaluate the feasibility of our approach. In addition, we wished to evaluate the relative performances of the three fixpoint computations shown in Figure 4. The experimentation was carried out on a Linux machine with 1 Ghz Pentium III processor and 1.5 Gbyte memory using program modules selected from the Collected Algorithms by the ACM (`http://www.acm.org/calgo`). Table 1 shows the information of the program modules used in the experimentation.

program	loc	stmt	branch	var	def	use
numforw	14	10	4	5	5	8
adjust	27	20	7	7	17	29
integrate	57	39	18	14	35	41
revolve	103	71	32	22	53	104

Table 2 summarizes the experimental results. The first column shows program modules. The second column shows the number of predicates for each program module. The third column shows the time usage of MAGIC to construct abstract state graphs from the program modules with predicates. The fourth column shows the number of reachable states of the resulting abstract state graphs. The remaining columns show the time and memory usage of NuSMV to perform the three fixpoint computations.

We first used NuSMV to perform abstract slicing with no predicates. As mentioned earlier, this corresponds to static slicing. The four rows with 0 predicates in Table 2 show the results. As expected, only a slight amount of time and memory was necessary for NuSMV to produce program slices.

We then used NuSMV to perform abstract slicing with a significant number of predicates. The remaining rows in

$old := false$;
$new := v$;
while $(old \neq new)$ do begin
  $old := new$;
  $new := old \cup \mathbf{EX}old$;
end while
return new;

(a) $\mathbf{EF}v$

$old := false$;
$new := v$;
while $(old \neq new)$ do begin
  $old := new$;
  $new := old \cup \mathbf{dda}(old) \cup \mathbf{ca}(old)$;
end while
return new;

(b) **affect**$[v]$, monolithic

for i = 0 to l do begin
  $old[i] := false$;
  if $(v = v_i)$ then $new[i] := true$;
  else $new[i] := false$;
end for
while $(old \neq new)$ do begin
  $old := new$;
  for $i = 0$ to l do
    for $j = 0$ to l do
      $new[i] := old[i] \cup \mathbf{dda}_{ij}(old[j]) \cup \mathbf{ca}_{ij}(old[j])$;
end while
return new;

(c) **affect**$[v]$, partitioned

program	predicates	abstract state graph		$\mathbf{EF}v$		**affect**$[v]$, monolithic		**affect**$[v]$, partitioned	
		user time (sec)	reachable states	user time (sec)	BDD nodes allocated	user time (sec)	BDD nodes allocated	user time (sec)	BDD nodes allocated
numforw	0	0.02	14	0.010	282	0.013	351	0.011	331
adjust	0	0.04	27	0.005	596	0.018	1386	0.025	788
integrate	0	0.10	57	0.012	1362	0.026	1830	0.035	1423
revolve	0	0.21	103	0.019	2509	0.081	6033	0.132	3592
numforw	16	0.28	$2^{16.9}$	0.051	17.6 (K)	0.06	17.1 (K)	0.06	17.1 (K)
adjust	34	0.81	$2^{35.9}$	0.432	143.7 (K)	0.44	141.6 (K)	0.45	141.7 (K)
integrate	30	3.88	$2^{30.3}$	13.627	1911.8 (K)	13.67	1911.0 (K)	13.76	1908.3 (K)
integrate	35	8.94	$2^{34.5}$	60.010	5756.4 (K)	60.94	5739.2 (K)	61.10	5735.3 (K)
revolve	48	37.83	$2^{48.9}$	7.042	1464.4 (K)	7.43	1415.8 (K)	9.34	1450.2 (K)
revolve	62	41.89	$2^{60.9}$	246.028	14359.5 (K)	251.26	14102.7 (K)	291.09	14461.6 (K)

Table 2 show the results. There are two points of using such a significant number of predicates. First, we wished to consider the worst case and evaluate the performance of abstract slicing in the presence of state explosion, although we do not believe that programmers want to use so many predicates when performing abstract slicing. Second, even though programmers are only interested in a small number of predicates, additional predicates are helpful and often mandatory since as the number of predicates increases, the precision of the resulting abstract state graph also increases, making abstract slicing more informative.

Abstract slicing with a smaller number of predicates does not always guarantee a better performance. For example, consider the abstract state graphs of `integrate` with 35 predicates and `revolve` with 48 predicates. Although the former is much smaller than the latter in terms of the number of reachable states, abstract slicing of the former required more time in one order of magnitude. This is consistent with the general principle of symbolic model checking that it is not the sheer number of reachable states but the complexity of a system being analyzed that determines the performance of symbolic model checking. In our case, we believe that the complexity of a program is strongly dependent on the number of conditional and repetative statements as well as the nested structure of such statements.

Finally we discuss the relative performances of the three fixpoint computations. Before carrying out the experimentation, we hypothesized that the monolithic and partitioned fixpoint computations of **affect**$[v]$ are much more demanding than that of $\mathbf{EF}v$ because the updates of *new* in Figure 4.(b) and Figure 4.(c) use very long CTL formulas, whereas the update of *new* in Figure 4.(a) uses only one CTL formula $\mathbf{EX}old$. For example, consider the program module `revolve`. There are 53 definitions, 104 uses, and 261 pairs of definitions and uses of the same variable and hence **dda**(*old*) is the disjunction of 261 formulas of the form $def(x) \wedge \mathbf{EXE}[\neg def(x)\mathbf{U}(use(x) \wedge old)]$. There are 32 branch conditions and hence **ca**(*old*) is the disjunction of 32 formulas of the form $v \wedge \mathbf{E}[\neg ipd(v)\mathbf{U}(\neg ipd(v) \wedge old)]$. Our experimentation, however, showed that there were no big differences among the relative performances of the three fixpoint computations. To understand this phenomenon, we collected the number of iterations of the updates of *new* as well as the time and memory usage of NuSMV to perform the update of *new* at each iteration. We found out that the monolithic and partitioned fixpoint computations

for **affect**[v] are short and thick since **dda**(*old*) and **ca**(*old*) introduce many program points into *new* at each iteration, while the fixpoint computation for **EF**v is long and thin since **EX***old* introduces program points, only in one step backward from some program point in *old*, into *new*.

6 Related Work

A number of different approaches to program slicing have been proposed in order to remedy the shortcomings of static slicing. Included are dynamic slicing[1, 24], simultaneous dynamic slicing[16], hybrid slicing[15], quasi static slicing[34], conditioned slicing[6], backward conditioned slicing[11], and pre/post conditioned slicing[18]. The main purpose of these approaches is to reduce the size of a program slice by limiting the scope of analysis to a single execution or a set of executions rather than all possible ones.

Dynamic slicing[1, 24] limits the scope of analysis using a valuation over input variables, which maps every input variable to its value. A program slice is produced with respect to the execution induced by a valuation. Simultaneous dynamic slicing[16] uses a set of valuations over input variables rather than one. Hybrid slicing[15] integrates information obtained by dynamic slicing into static slicing. Since dynamic slicing exploits run-time information obtained during the execution of a program, it produces a program slice that is significantly more precise and smaller than its static counterpart and is well-suited for program debugging with complex data structures such as arrays and pointers. However, the analysis result is confined to a single execution.

The approaches in [34, 6, 11, 18] lie between static slicing and dynamic slicing. Quasi static slicing[34] limits the scope of analysis using a partial valuation over input variables, which maps some input variables to their values while leaving the others unconstrained. A program slice is defined with respect to the set of executions induced by a partial valuation. Conditioned slicing[6] (resp. backward conditioned slicing[11]) limits the scope of analysis using a condition and performs forward symbolic execution (resp. backward symbolic execution) to identify the set of executions induced by the condition. Pre/post conditioned slicing[18] is a combination of conditioned slicing and backward conditioned slicing. These approaches have two limitations. First, the question of "under which variable values does a program point affect another?" has remained unanswered. Second, the question of "does a program point affect another if we are only interested in the constrained executions?" is answered using symbolic execution. To identify the set of executions induced by a condition, symbolic execution should propagate the condition through all possible executions of the program. The number of possible executions is often very large or infinite, which makes it necessary to have a bound on the number of iterations of loops or the size of the input domain[23, 27]. This bound in turn may lead to a program slice which is not sound in that some affections between program points are missing.

The approach in [26] addresses the question of "for two given program points, under which variable values does a program point affect another?" by performing symbolic execution through all possible executions between the two program points. Since this approach is also based on symbolic execution, it may miss some values of variables under which a program point affects another. Moreover, the approach is only applicable to two given program points. It is not clear how the approach can be generalized for backward slicing and forward slicing where we are given a slicing criterion and want to find, for every program point, the values of variables under which the program point affects or is affected by the slicing criterion.

7 Conclusions and Future Work

We have presented an approach to program slicing based on predicate abstraction and symbolic model checking. We described the notion of abstraction slicing that extends static slicing with predicates and constraints. We also described a method for performing abstract slicing that reduces abstract slicing to a least fixpoint computation over CTL formulas. The method was implemented in NuSMV and was applied to programs with a significant number of predicates, demonstrating the feasibility of our approach.

We have two main directions for future research. First, we plan to apply abstract slicing to interprocedural programs and object-oriented programs. Program slicing of such programs is more complicated due to global and local variables, reference parameters, procedure call/return, and recursion. We are investigating how conventional static slicing methods for interprocedural programs such as [21, 31] can be combined with predicate abstraction and symbolic model checking. Second, we plan to apply abstract slicing to requirements specifications written in statecharts[17] or SDL[3] whose underlying model is extended finite state machine, that is, finite state machine with data variables. We are investigating how abstract slicing can be extended to cope with the language constructs of statecharts and SDL for modeling hierarchy, concurrency, and communications.

References

[1] H. Agrawal and J.R. Horgan, "Dynamic Program Slicing," *Proc. of the ACM Conference on Programming Language Design and Implementation*, pp. 246-256, 1990.

[2] T. Ball and S.K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *SPIN '01*, Vol. 2057 of LNCS, pp. 103-122, Springer, 2001.

[3] F. Belina and D. Hogrefe, "The CCITT-Specification and Description Language SDL," *Computer Networks and ISDN Systems*, 16(3):311-341, Mar. 1989.

[4] D.W. Binkley and K.B. Gallagher, "Program Slicing," *Advances in Computing*, 43:1-50, Academic Press, 1996.

[5] M. Bozga, J-.C. Fernandez, and L. Ghirvu, "Using Static Analysis to Improve Automatic Test Generation," *International Journal on Software Tools for Technology Transfer*, 4(2):142-152, Feb. 2003.

[6] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned Program Slicing," *Information and Software Technology*, 40(11/12):595-607, Dec. 1998.

[7] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C," *IEEE Transactions on Software Engineering*, 30(6):388-402, June 2004.

[8] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," *CAV '02*, Vol. 2404 of LNCS, pp. 359-364, Springer, 2002.

[9] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, The MIT Press, 1999.

[10] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proc. of the ACM Symposium on Principles of Programming Languages*, pp. 238-252, 1977.

[11] S. Danicic, C. Fox, M. Harman, and R.M. Hierons, "Backward Conditioning: A New Program Specialisation Technique and its Application to Program Comprehension," *Proc. of the IEEE Workshop on Program Comprehension*, pp. 89-97, 2001.

[12] A. De Lucia, "Program Slicing: Methods and Applications," *Proc. of the IEEE Workshop on Source Code Analysis and Manipulation*, pp. 142-149, 2001.

[13] E.A. Emerson, "Temporal and Modal Logic," *Handbook of Theoretical Computer Science*, Vol. B, Ch. 16, pp. 995-1072, MIT Press, 1990.

[14] S. Graf and H. Saidi, "Construction of Abstract State Graphs with PVS," *CAV '97*, Vol. 1254 of LNCS, pp. 72-83, Springer, 1997.

[15] R. Gupta, M.L. Soffa, and J. Howard, "Hybrid Slicing: Integrating Dynamic Information with Static Analysis," *ACM Transactions on Software Engineering and Methodology*, 6(4):370-397, Oct. 1997.

[16] R.J. Hall, "Automatic Extraction of Executable Program Subsets by Simultaneous Program Slicing," *Journal of Automated Software Engineering*, 2(1):33-53, 1995.

[17] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, 8(3):231-274, June 1987.

[18] M. Harman, R.M. Hierons, C. Fox, S. Danicic, and J. Howroyd, "Pre/Post Conditioned Slicing," *Proc. of the IEEE International Conference on Software Maintenance*, pp. 138-147, 2001.

[19] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, H. Zheng, "A Formal Study of Slicing for Multi-Threaded Programs with JVM Concurrency Primitives," *SAS '99*, Vol. 1694 of LNCS, pp. 1-18, Springer, 1999.

[20] T.A. Henzinger, R. Jhala, R. Majumdar and G. Sutre "Lazy Abstraction," *Proc. of the ACM Symposium on Principles of Programming Languages*, pp. 58-70, 2002.

[21] S. Horowitz, T. Reps, and D.W. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, 12(1):26-60, Jan. 1990.

[22] D. Jackson and E.J. Rollins, "A New Model of Program Dependences for Reverse Engineering," *Proc. of the ACM Symposium on the Foundations of Software Engineering*, pp. 2-10, 1994.

[23] D. Jackson and M. Vaziri, "Finding Bugs with a Constraint Solver," *Proc. of the ACM International Symposium on Software Testing and Analysis*, pp. 14-25, 2000.

[24] B. Korel and J. Laski, "Dynamic Program Sling," *Information Processing Letters*, 29(3):155-163, Oct. 1988.

[25] D. Kozen, "Results on the Propositional Mu-calculus," *Theoretical Computer Science*, 27(3):333-354, 1983.

[26] J. Krinke and G. Snelting, "Validation of Measurement Software as An Application of Slicing and Constraint Solving," *Information and Software Technology*, 40(11/12):661-676, Dec. 1998.

[27] S. Khursid, C. Pasareanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," *TACAS '03*, Vol. 2619 of LNCS, pp. 553-568, Springer, 2003.

[28] K.L. McMillan, *Symbolic Model Checking – an Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.

[29] K. Ottenstein and L. Ottenstein, "The Program Dependence Graph in A Software Development Environment," *Proc. of the ACM Symposium on Practical Software Development Environments*, pp. 177-184, 1984.

[30] 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, Sept. 1990.

[31] T. Reps and G. Rosay, "Precise Interprocedural Chopping," *Proc. of the ACM Symposium on Foundations of Software Engineering*, pp. 41-52, 1995.

[32] D.A. Schmidt and B. Steffen, "Program Analysis as Model Checking of Abstract Interpretations," *SAS '98*, Vol. 1503 of LNCS, pp. 351-380, Springer, 1998.

[33] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, 3(3):121-189, Sept. 1995.

[34] G.A. Venkatesh, "The Semantic Approach to Program Slicing," *ACM SIGPLAN Notices*, 26(6):107-119, 1991.

[35] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984.