October 2005

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

Hyoung Seok Hong
Concordia University, Canada

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Recommended Citation

Copyright 2005 IEEE. Reprinted from Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05), pages 25-34

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/229
For more information, please contact libraryrepository@pobox.upenn.edu.
Abstract Slicing: A New Approach to Program Slicing Based on Abstract Interpretation and Model Checking

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.

Comments
Copyright 2005 IEEE. Reprinted from Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05), pages 25-34

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.
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 $v_4$
Abstract slicing shows the values of predicates under which \( v_1, v_2, v_3, v_4 \) affect \( v_4 \) (Figure 1.(b)). 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 \( v_4 \). 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 \( v_f \in V \) and \( v_j \in V \) 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: $\{\text{def}(x) \mid x \text{ is defined at } v\} \cup \{\text{use}(x) \mid x \text{ is used at } v\}$. Figure 2 shows the flow graph of the example program in Figure 1.

\[
\begin{align*}
v_1: & \quad \text{max} = x; \\
v_2: & \quad \text{if}(y > x); \\
v_3: & \quad \text{max} = y; \\
v_4: & \quad \text{return max}; \\
\end{align*}
\]

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 $v$.

**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 $v'$.**

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 $\sigma$ over $\{b_1, ..., b_n\}$ represents the predicate $p_1 \land ... \land p_n$ where for every $1 \leq i \leq n$, $p_i = b_i$ if $\sigma(b_i) = 1$ and $p_i = \neg b_i$ if $\sigma(b_i) = 0$. For notational convenience, we identify a valuation $\sigma$ with the boolean vector $(\sigma(b_1), ..., \sigma(b_n))$. For example, in Figure 3.(a), $\{1\}$ and $\{0\}$ represent the predicates $y > x$ and $\neg(y > x)$, respectively. For a valuation $\sigma$ over the program’s variables, $\sigma$ is an abstraction of $\sigma'$ if $\sigma'$ satisfies $p_1 \land ... \land p_n$. For example, define $\sigma_{x,y}$ as a valuation over $\{x, y\}$ that maps $x$ to 0 and $y$ to 1. $\{1\}$ is an abstraction of $\sigma_{x,y}$. For a predicate $\text{cond}$ over the program’s variables, $\sigma$ satisfies $\text{cond}$ if there is a valuation $\sigma_c$ such that $\sigma$ is an abstraction of $\sigma_c$ and $\sigma_c$ satisfies $\text{cond}$. For example, $\{1\}$ 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 $(v, \sigma)$ where the control part is a program point $v$ and the data part is a valuation $\sigma$ over $\{b_1, ..., b_n\}$. A node $(v, \sigma)$ 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.

When determining whether there is an arc from $(v, \sigma)$ to $(v', \sigma')$, we consider two cases. The first case is that $v$ is a simple statement. There is an arc from $(v, \sigma)$ to $(v', \sigma')$ if there are two valuations $\sigma_c$ and $\sigma_c'$ over the program’s variables such that (i) $\sigma$ is an abstraction of $\sigma_c$, (ii) $\sigma'$ is an abstraction of $\sigma_c'$, and (iii) the execution of $v$ with $\sigma_c$ leads to $v'$ and $\sigma_c'$. For example, in Figure 3.(a), using $\sigma_c = \sigma_c'$.
\( \sigma_{x,y} \), we can show that there is an arc from \((v_1, 1)\) to \((v_2, 1)\). 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 \((v, \sigma)\) to \((v_1, \sigma)\) if \(\sigma\) satisfies \(cond\) and there is an arc from \((v, \sigma)\) to \((v_2, \sigma)\) if \(\sigma\) satisfies \(\neg cond\). For example, consider \(v_2\) whose true successor is \(v_3\) and false successor is \(v_4\). There is an arc from \((v_2, (1))\) to \((v_3, (1))\) since \(1\) satisfies the branch condition \(y > x\), while there is no arc from \((v_2, (1))\) to \((v_4, (1))\).

We define the abstract state graph \(G(c_1, \ldots, c_m)\) of a program with predicates \(\{p_1, \ldots, p_n\}\) and constraints \(\{c_1, \ldots, c_m\}\) as follows. A constraint is a precondition associated with a program point. The set of nodes of \(G(c_1, \ldots, c_m)\) is a subset of nodes \((v, \sigma)\) of \(G(p_1, \ldots, p_n)\) such that if \(c_i\) is a constraint associated with \(v\), then \(\sigma\) satisfies \(c_i\). The set of arcs of \(G(c_1, \ldots, c_m)\) is obtained by restricting the domain and range of the set of arcs of \(G(p_1, \ldots, p_n)\) to the nodes of \(G(c_1, \ldots, c_m)\). Figure 3.(b) shows \(G(\sigma_{2, \ldots}, \sigma_{3, \ldots})\) of the example program. Due to the constraint \(y > x\) at \(v_1\), we have that \((v_1, (1))\) is a node while \((v_1, (0))\) 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 \(\pi = q_0, q_1, \ldots\) of a Kripke structure, we use \(i(\pi)\) to denote the \(i\)-th element of \(\pi\). 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 \(EXf\) and \(E[Ug]\) whose semantics is defined below.

- \(q \models EXf\) iff for some path \(\pi\) such that \(\pi(i) = q\) for some \(i \geq 0\), we have that \(\pi(i + 1) \models f\).

- \(q \models E[Ug]\) iff for some path \(\pi\) 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 \(X^{-}\) (yesterday) and \(U^{-}\) (since) that are the duals of \(X\) and \(U\), respectively [13].

- \(q \models EX^{-}f\) iff for some path \(\pi\) such that \(\pi(i) = q\) for some \(i > 0\), we have that \(\pi(i - 1) \models f\).

- \(q \models E[U^{-}g]\) iff for some path \(\pi\) 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 \(EF\) which is the abbreviation of \(E[trueUv]\) expressing that \(v\) is reachable. The set of states satisfying \(EF\) is equivalent to the least fixpoint of \(\tau(Z) = v \lor EXZ\). 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 \((v, \sigma)\) and \((v', \sigma')\) of an abstract state graph, we say that \((v, \sigma)\) directly data-affects \((v', \sigma')\) if there is a variable \(x\) such that \(x\) is defined at \(v\), \(x\) is used at \(v'\), and there is a path \((v, \sigma), (v_1, \sigma_1), \ldots, (v_n, \sigma_n), (v', \sigma')\) such that for every \(1 \leq i \leq n\), \(x\) is not defined at \(v_i\). In Figure 3.(a), we observe that \((v_1, (1))\) does not directly data-affect \((v_3, (1))\) since they are intervened by \((v_2, (1))\). We also observe that \((v_1, (0))\) directly data-affects \((v_3, (0))\). These two observations enable us to infer that \(v_3\) 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 \((v, \sigma)\) of an abstract state graph does not necessarily have two successors depending on the valuation \(\sigma\). Rather, we adopt the following definitions introduced by Podgurski and Clarke[30]: \(v'\) is a postdominat 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 postdominator
of \( v \) that occurs on every path from \( v \) to \( v' \). \( v \) 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 \( \langle v, \sigma \rangle \) control-affects \( \langle v', \sigma' \rangle \) if there is a path \( \langle v, \sigma \rangle, \langle v_1, \sigma_1 \rangle, ..., \langle v_n, \sigma_n \rangle, \langle v', \sigma' \rangle \) such that \( \sigma \neq ipd(v), v' \neq ipd(v) \), and for every \( 1 \leq i \leq n, v_i \neq ipd(v) \). In Figure 3(a), \( \langle v_2, \langle 1 \rangle \rangle \) control-affects \( \langle v_3, \langle 1 \rangle \rangle \).

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 \( \langle v', \sigma' \rangle \) if there is a node \( \langle v, \sigma \rangle \) such that \( \langle v, \sigma \rangle \) affects \( \langle v', \sigma' \rangle \). \( \langle v, \sigma \rangle \) affects \( \langle v', \sigma' \rangle \) if there is a node \( \langle v', \sigma' \rangle \) such that \( \langle v, \sigma \rangle \) affects \( \langle v', \sigma' \rangle \). \( v \) affects \( v' \) if there are two nodes \( \langle v, \sigma \rangle \) and \( \langle v', \sigma' \rangle \) such that \( \langle v, \sigma \rangle \) affects \( \langle v', \sigma' \rangle \).

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 \( \langle v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\} \rangle \). The abstract backward slice with respect to \( \langle v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\} \rangle \) consists of the reachable nodes of \( G \{p_1, ..., p_n\} \{c_1, ..., c_m\} \) that affect \( v \). The abstract forward slice with respect to \( \langle v, \{p_1, ..., p_n\}, \{c_1, ..., c_m\} \rangle \) consists of the reachable nodes of \( G \{p_1, ..., p_n\} \{c_1, ..., c_m\} \) that are affected by \( v \). A slicing criterion for abstract chopping is a quadruple \( \langle v, v', \{p_1, ..., p_n\}, \{c_1, ..., c_m\} \rangle \). The abstract chop with respect to \( \langle v, v', \{p_1, ..., p_n\}, \{c_1, ..., c_m\} \rangle \) consists of the reachable nodes of \( G \{p_1, ..., p_n\} \{c_1, ..., c_m\} \) that are affected by \( v \) and \( v' \).

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 \( \langle v_1, \sigma_1 \rangle, ..., \langle v_n, \sigma_n \rangle \) such that \( \langle v_i, \sigma'_i \rangle \rangle \) affects \( \langle v_n, \sigma_n \rangle \rangle \), then its abstract state graph has an execution \( \langle v_1, \sigma_1 \rangle, ..., \langle v_n, \sigma_n \rangle \rangle \) such that \( \langle v_1, \sigma_1 \rangle \rangle \) affects \( \langle v_n, \sigma_n \rangle \rangle \) for every \( 1 \leq i \leq n, \sigma_1 \rangle \rangle \) is an abstraction of \( \sigma'_i \rangle \rangle . \)

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),

\[
\begin{align*}
ABS_1 & = \{ (v_1, \langle 0 \rangle) \} \\
ABS_2 & = \{ (v_2, \langle 1 \rangle) \} \\
ABS_3 & = \{ (v_3, \langle 1 \rangle) \} \\
ABS_4 & = \{ (v_4, \langle 1 \rangle) \} \\
\end{align*}
\]

In Figure 3(b),

\[
\begin{align*}
ABS_1 & = \emptyset \\
ABS_2 & = \{ (v_2, \langle 1 \rangle) \} \\
ABS_3 & = \{ (v_3, \langle 1 \rangle) \} \\
ABS_4 & = \{ (v_4, \langle 1 \rangle) \} \\
\end{align*}
\]

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

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

4 Abstract Slicing as Symbolic Model Checking

A widely-used method for performing 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 \( \mu \)-calculus[25] since a least fixpoint over CTL formulas is a \( \mu \)-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 \( G(x_1, \ldots, x_n) \). Define predicate transformers \( \text{dda} \) and \( \text{ca} \) by

\[
\text{dda}(Z) = \bigvee_{x \in X} \text{def}(x) \land \text{EXE}[^{-}\text{def}(x) \cup \text{use}(x) \land Z],
\]

where \( X \) is the set of variables of the program,

\[
\text{ca}(Z) = \bigvee_{v \in V} v \land \text{E}[^{-}\text{ipd}(v) \cup (-\text{ipd}(v) \land Z)].
\]

For example, in Figure 3.(a),

\[
\text{dda}((v_4, (1))) = \{(v_3, (1))\} \quad \text{and} \quad \text{ca}((v_3, (1))) = \{(v_2, (1))\}.
\]

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

Define a predicate transformer \( \text{affect}[v] \) by

\[
\text{affect}[v](Z) = v \lor \text{dda}(Z) \lor \text{ca}(Z).
\]

It is not hard to see that the least fixpoint of \( \text{affect}[v] \) 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 \( \{v, \{p_1, \ldots, p_n\}, \{c_1, \ldots, c_m\}\} \).

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 \( Z_1, \ldots, Z_l \) according to the program points \( V = \{v_1, \ldots, v_l\} \) and manipulate each \( Z_i \) separately during the fixpoint computation. This partition leads to the new definition of \( \text{affect}[v] \):

\[
\text{affect}[v](Z_1, \ldots, Z_l) = (v_1 ~ v \ldots v_n ~ v) \lor \bigvee (\text{dda}(Z_1, \ldots, Z_l), \ldots, \text{dda}(Z_1, \ldots, Z_l) \lor (\text{ca}(Z_1, \ldots, Z_l), \ldots, \text{ca}(Z_1, \ldots, Z_l))
\]

where for every \( 1 \leq i \leq l \),

- \( \text{dda}_i(Z_1, \ldots, Z_l) = \bigvee_{1 \leq j \leq l} \text{dda}_{ij}(Z_j) \)
  where for every \( 1 \leq j \leq l \),
  \( \text{dda}_{ij}(Z_j) = v_i \land \bigvee_{x \in \text{DEF}(v_i)} \text{def}(x) \land \text{EXE}[^{-}\text{def}(x) \cup \text{use}(x) \land Z_j] \)
  where \( \text{DEF}(v_i) \) is the set of variables defined at \( v_i \).
- \( \text{ca}_i(Z_1, \ldots, Z_l) = \bigvee_{1 \leq j \leq l} \text{ca}_{ij}(Z_j) \)
  where for every \( 1 \leq j \leq l \),
  \( \text{ca}_{ij}(Z_j) = v_i \land \text{E}[^{-}\text{ipd}(v_i) \cup (-\text{ipd}(v_i) \land v \land Z_j)] \)

For example, in Figure 3.(a),
\[ \text{dda}_3(\emptyset, \emptyset, \{(v_4, (1))\}) = \{(v_3, (1))\} \]
\[ \text{ca}_2(\emptyset, \emptyset, \{(v_5, (1))\}) = \{(v_2, (1))\} \]

A node is in \( \text{dda}_i(Z_j) \) (resp. \( \text{ca}_i(Z_j) \)) 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 \( \text{dda}_i(Z_{1}, ..., Z_l) \) (resp. \( \text{ca}_i(Z_{1}, ..., Z_l) \)) if and only if the node has \( v_i \) as its program point and directly data-affects (resp. control-affects) some node in \( Z_{1} \cup ... \cup Z_l \). The least fixpoint of \( \text{affect}^-[v] \) is the tuple \( (Z'_1, ..., Z'_l) \) such that for every \( 1 \leq i \leq l \), \( Z'_i \) is the set of nodes that have \( v_i \) as the program point and affect \( v \).

Since \( \text{dda}_i \) and \( \text{ca}_ij \) 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 \( (v_i, \sigma) \) does not directly data-affect \( (v_j, \sigma') \). Similarly, if \( v_i \) does not control-affect \( v_j \), then \( (v_i, \sigma) \) does not control-affect \( (v_j, \sigma') \). 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 \( \text{dda}_ij \) and \( \text{ca}_ij \):

- If \( v_i \) directly data-affects \( v_j \),
  \[ \text{dda}_ij(Z_j) = v_i \land \bigvee_{x \in \text{DEF}(v_i)} \text{def}(x) \land \text{EX}[-\text{def}(x) \lor \text{U}(\text{use}(x) \land Z_j)] \]
  otherwise, \( \text{dda}_ij(Z_j) = \text{false} \).
- If \( v_i \) control-affects \( v_j \),
  \[ \text{ca}_ij(Z_j) = v_i \land \text{E}[\neg \text{ipd}(v_i) \lor (\text{ipd}(v_i) \land Z_j)] \]
  otherwise, \( \text{ca}_ij(Z_j) = \text{false} \).

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 \( \text{affect}^-[v] \) 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 \( \text{EF}v \) in Figure 4.(a). In order to compute \( \text{dda}(\text{old}) \), \( \text{ca}(\text{old}) \), \( \text{dda}_i(\text{old}[j]) \), and \( \text{ca}_ij(\text{old}[j]) \), the algorithms use the capability of NuSMV to find the set of states satisfying CTL formulas of the form \( \text{EX}f \) or \( \text{E}[f \lor g] \). For abstract forward slicing and abstract chopping, we do not have an implementation. To compute the least fixpoint of \( \text{affect}^-[v] \), we should be able to find the set of states satisfying past CTL formulas of the form \( \text{EX}^-f \) or \( \text{E}[f \lor g] \), 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.

```
<table>
<thead>
<tr>
<th>program</th>
<th>loc</th>
<th>stmt</th>
<th>branch</th>
<th>var</th>
<th>def</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>numforw</td>
<td>14</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>adjust</td>
<td>27</td>
<td>20</td>
<td>7</td>
<td>7</td>
<td>17</td>
<td>29</td>
</tr>
<tr>
<td>integrate</td>
<td>57</td>
<td>39</td>
<td>18</td>
<td>14</td>
<td>35</td>
<td>41</td>
</tr>
<tr>
<td>revolve</td>
<td>103</td>
<td>71</td>
<td>32</td>
<td>22</td>
<td>53</td>
<td>104</td>
</tr>
</tbody>
</table>
```

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
Table 2 shows 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 repetitive 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 $\text{affect}[v]$ are much more demanding than that of $\text{EF}v$ because the updates of $\text{new}$ in Figure 4.(b) and Figure 4.(c) use very long CTL formulas, whereas the update of $\text{new}$ in Figure 4.(a) uses only one CTL formula $\text{EXold}$. 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 $\text{dda}(\text{old})$ is the disjunction of 261 formulas of the form $\text{def}(x) \land \text{EXE}[\neg \text{def}(x) \cup \text{use}(x) \land \text{old})].$ There are 32 branch conditions and hence $\text{ca}(\text{old})$ is the disjunction of 32 formulas of the form $v \land \text{E}[\neg \text{ipd}(v) \cup \text{E}[\neg \text{ipd}(v) \land \text{old})].$ 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 $\text{new}$ as well as the time and memory usage of NuSMV to perform the update of $\text{new}$ at each iteration. We found out that the monolithic and partitioned fixpoint computations

\[
\begin{align*}
\text{old} & := \text{false}; \\
\text{new} & := v; \\
\text{while} & \ (\text{old} \neq \text{new}) \ \text{do} \left( \\
\text{old} & := \text{new}; \\
\text{new} & := \text{old} \cup \text{EXold}; \\
\text{end} \right) \left( \\
\text{return} & \text{new}; \\
\right) \left( \\
\text{(a) } \text{EF}v \right)
\end{align*}
\]

\[
\begin{align*}
\text{old} & := \text{false}; \\
\text{new} & := v; \\
\text{while} & \ (\text{old} \neq \text{new}) \ \text{do} \left( \\
\text{old} & := \text{new}; \\
\text{new} & := \text{old} \cup \text{dda}(\text{old}) \cup \text{ca}(\text{old}); \\
\text{end} \right) \left( \\
\text{return} & \text{new}; \\
\right) \left( \\
\text{(b) } \text{affect}[v], \text{monolithic} \right)
\end{align*}
\]

\[
\begin{align*}
\text{old} & := \text{false}; \\
\text{new} & := v; \\
\text{while} & \ (\text{old} \neq \text{new}) \ \text{do} \left( \\
\text{old} & := \text{new}; \\
\text{new} & := \text{old} \cup \text{dda}(\text{old}) \cup \text{ca}(\text{old}); \\
\text{end} \right) \left( \\
\text{return} & \text{new}; \\
\right) \left( \\
\text{(c) } \text{affect}[v], \text{partitioned} \right)
\end{align*}
\]
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 EFv is long and thin since EXold 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


