Dynamic Path Reduction for Software Model Checking

Zijiang Yang[1], Bashar Al-Rawi[2], Karem Sakallah[2], Xiaowan Huang[3], Scott Smolka[3], and Radu Grosu[3]

[1] Western Michigan University, Kalamazoo, MI, USA
[2] University of Michigan, Ann Arbor, MI, USA
[3] Stony Brook University, Stony Book, NY, USA

Abstract. We present the new technique of *dynamic path reduction* (DPR), which allows one to prune redundant paths from the state space of a program under verification. DPR is a very general technique which we consider here in the context of the bounded model checking of sequential programs with nondeterministic conditionals. The DPR approach is based on the symbolic analysis of concrete executions. For each explored execution path π that does not reach an `abort` statement, we repeatedly apply a weakest-precondition computation to accumulate the constraints associated with an infeasible sub-path derived from π by taking the alternative branch to an if-then-else statement. We then use an SMT solver to learn the minimally unsatisfiable core of these constraints. By further learning the statements in π that are critical to the sub-path's infeasibility as well as the control-flow decisions that must be taken to execute these statements, unexplored paths containing the same unsatisfiable core can be efficiently and dynamically pruned. Our preliminary experimental results show that DPR can prune a significant percentage of execution paths, a percentage that grows with the size of the instance of the problem being considered.

1 Introduction

There are two approaches to software model checking. The first, as typified by [2], applies traditional model-checking techniques [12, 10] to a finite-state model automatically extracted from the software system in question. The use of abstraction techniques [4, 9] leads to a model with more behaviors than the original program and consequently an analysis that is conservative in nature. This form of software model checking allows one to prove the absence of certain types of errors without actually executing the program. Its success hinges on recent advances in symbolic techniques. Performance can be further improved by exploiting software-specific features [16, 18].

The second approach is based on the dynamic execution of the actual program; see, for example, [7]. It differs from testing in that it explores exhaustively the program's state space. This approach allows for the detection of subtle implementation errors that are usually missed by abstraction-based software model

checkers. In the case of concurrent systems, partial-order reduction [15, 13, 6] can be used to reduce the size of the state space by exploiting the commutativity of concurrently executed transitions that result in the same state when executed in different orders.

In this paper, we present the new technique of *dynamic path reduction* (DPR), which allows one to prune redundant paths from the state space of a program under verification. DPR is a very general technique which we consider here in the context of the bounded model checking of sequential programs with nondeterministic conditionals. Such programs arise naturally as a byproduct of abstraction during verification as well as being inherent in nondeterministic programming languages. Nondeterministic choice also arises in the modeling of randomized algorithms. The key idea behind DPR is to *learn* from each explored path so that unexplored paths exhibiting the same behavior can be avoided (pruned).

To illustrate the DPR approach to model checking, consider the C program of Figure 1(a). Its first two conditional statements are nondeterministic, denoted by placing an asterisk in the condition position. The property we would like to check is whether the program can reach the `abort` statement. The initial values of variables `x, y, z` are 5, 8, 20, respectively. Suppose the first executed path is $\pi = \langle 0, 1, 2, 5, 6, 7, 10, 13 \rangle$. Executing the program along this path avoids the `abort` statement and ends with the `halt` statement. After executing this path, most existing model checkers will backtrack to Line 6 and explore the else-branch in the next execution. Since there are two nondeterministic choices in the program, four executions are required to prove that it cannot be aborted.

This is where DPR comes into play. Analyzing the execution trace π allows us to learn that the assignments `x=5` and `x=2*x` falsify the predicate `x>10` which forces the third conditional to choose its else-branch. We also learn that none of the assignments within the branches of the nondeterministic conditionals can make the predicate true. This allows us to prune all the remaining paths from the search space. A DPR-based model checker would therefore stop after the first execution and report that `abort` is not reachable.

The rest of the paper is organized as follows. Section 2 presents our execution-based, bounded model-checking algorithm with dynamic path reduction. Section 3 discusses our space-efficient, symbolic representation of execution paths. Section 4 contains our experimental results, while Section 5 considers related work. Section 6 offers our concluding remarks and directions for future work.

2 DPR-Based Model Checking Algorithm

In this section, we present `DPR-MC`, our bounded model-checking algorithm with dynamic path reduction. Our presentation is carried out in stages, starting with a simplified but transparent version of the algorithm, and with each stage incrementally improving the algorithm's performance. The model-checking algorithm we propose is tunable to run either as a randomized Las Vegas algorithm or as a guided-search algorithm.

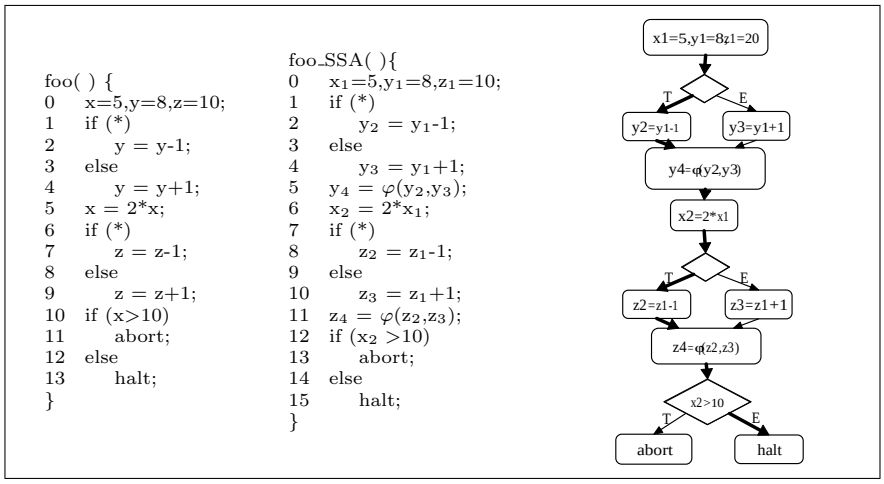

Fig. 1. A sample C program (left), its SSA form (middle), and SSA graph representation (right).

As defined formally below, a k-oracle is a bit string of length k representing a sequence of nondeterministic choices a program might make during execution. Suppose we want to perform bounded model checking on a program up to search depth D, such that within this D-bounded search space, each execution path contains at most k nondeterministic choices. In this case, the `DPR-MC` algorithm repeats the following three steps until the entire D-bounded search space has been explored: (1) Ask the constraint (SAT) solver to provide a k-oracle. (2) Execute the program on that oracle; stop if an `abort` statement is reached. (3) Use the constraint solver again to prune from the search space all paths that are equivalent to the one just executed.

2.1 Global Search Algorithm

The core language we use for analysis is a subset of C, extended with one statement type not present in C: nondeterministic conditionals. To simplify the analysis undertaken by `DPR-MC`, we use the *static single assignment* (SSA) representation of programs. For example, the SSA representation of the C program of Figure 1 (left) is shown in Figure 1 (middle). By indexing (versioning) variables and introducing the so-called φ function at join points, this intermediate representation ensures that each variable is statically assigned only once. We leverage the SSA representation to interface with the satisfiability modulo theory (SMT) solver Yices [5]. In this context, every statement (excepting statements within loops) can be conveniently represented as a predicate. Looping statements are handled by unfolding them up so that every execution path has at most k nondeterministic choices; i.e., a k-oracle is used to resolve the choices. We refer to the SSA representation obtained after such a k-unfolding as the *dynamic single assignment* (DSA) representation.

Suppose the program C to be analyzed has at most k nondeterministic conditionals on every execution path. We call a resolution of these k conditionals a *k-oracle*. Obviously, each k-oracle uniquely determines a finite concrete execution path of C. Let $\mathcal{R}$ be the set of all k-oracles (resolvents) of C. $\mathcal{R}$ can be organized as a decision tree whose paths are k-oracles.

Algorithm 1 DPR-MC(PROGRAM C, INT k)

1: $\mathcal{R}$ = all k-oracles in C;
2: **while** $\mathcal{R} \neq \emptyset$ **do**
3: Remove an oracle $R = \langle r_1 r_2 \dots r_k \rangle$ from $\mathcal{R}$;
4: ExecuteFollowOracle($R, \mathcal{R}, k$);
5: **end while**
6: exit("No bug found up to oracle-depth k");

Algorithm 1 is the main loop of our `DPR-MC` algorithm. It repeatedly removes a k-oracle R from $\mathcal{R}$ and executes C as guided by R. The algorithm terminates if: (1) execution reaches `abort` within `ExecuteFollowOracle`, indicating that a bug is found; or (2) $\mathcal{R}$ becomes empty after all oracles have been explored, in which case the program is bug-free to oracle-depth k.

Note that Algorithm 1 employs a *global* search strategy. If the oracle removal is random, it corresponds to a randomized Las Vegas algorithm. If the oracle removal is heuristic, it corresponds to a guided-search algorithm. Obviously, the number of oracles is exponential in the depth k of the decision tree $\mathcal{R}$. Hence, the algorithm is unlikely to work for nontrivial programs. We subsequently shall show how to efficiently store the decision tree and how to prune oracles by learning from previous executions.

2.2 Weakest Precondition Computation

An execution path $\pi = \langle s_1, s_2, \dots, s_n \rangle$ is a sequence of program statements, where each s_i is either an assignment or a conditional. We write c_T and c_E for the `then` and `else` branches respectively of a conditional statement c. For brevity, we sometimes refer to an execution path simply as a "path".

Definition 1. *Let* `x` *be a variable,* `e` *an expression,* `c` *a Boolean expression, P a predicate, and P[`e`/`x`] the simultaneous substitution of* `x` *with* `e` *in P. The* weakest precondition `wp(`π`,`P`)` *of π with respect to P is defined inductively as follows:*

Assignment: `wp(x=e,` P`)` $= P$`[e/x]`.
Conditional: `wp(if(c)`$_T$`,` P`)` $= P \wedge$ `c`; `wp(if(c)`$_E$`,` P`)` $= P \wedge \neg$`c`.
Nondeterminism: `wp(if(*)`$_T$`,` P`)` $=$ `wp(if(*)`$_E$`,` P`)` $= P$.
Sequence: `wp(`$s_1; s_2$`,` P`)` $=$ `wp(`s_1`, wp(`s_2`,` P`))`.

Given an execution path $\pi = \langle s_1, s_2, \dots, s_n \rangle$, we use $\pi^i = s_i$ to denote the i-th statement of π, and $\pi^{i,j} = s_i, \dots, s_j$ to denote the segment of π between

i and j. Assume now that π^n, the last statement of π, is either c_T or c_E. If $\pi^n = c_T$, then it is impossible for any execution path with prefix $\pi^{1,n-1}$ to take the `else`-branch at π^n. That is, any execution path that has ρ as a prefix, where $\rho^i = \pi^i (1 \leq i < n)$ and $\rho^n \neq \pi^n$, is infeasible. Because of this, we say that ρ is an *infeasible sub-path*.

Let ρ be an infeasible sub-path of length m where ρ^m is a conditional c. We use $wp(\rho)$ to denote $wp(\rho^{1,m-1}, c)$, and $wp(\rho) = \mathit{false}$ as ρ is infeasible. According to Definition 1, assuming that ρ contains $t < m$ conditionals in addition to c, we have:

$$wp(\rho) = c' \wedge (c'_1 \wedge c'_2 \ldots \wedge c'_t) = \mathit{false}$$

where c' is ρ^n transformed through transitive variable substitutions, and similarly each c'_l is a transformed deterministic predicate in s_l: $(c_l)_{T/E}$ $(1 \leq l \leq t)$. More formally, given a formula F, we use F' to denote the formula in wp that is transformed from F. The definition is transitive in that both $F' = F(e/v)$ and $F'(e_2/v_2)$ are *transformed formulae* from F.

2.3 Learning From Infeasible Sub-paths

Upon encountering a new execution path, the `DPR-MC` algorithm collects information about infeasible sub-paths at deterministic predicates by using the weakest precondition computation presented in the previous section. We now analyze the reasons behind the infeasibility of such paths in order to provide useful information for pruning unexplored execution paths.

Since $wp(\rho)$ is unsatisfiable, there must exist an unsatisfiable subformula $wp_{us}(\rho)$ that consists of a subset of clauses $\{c', c'_1, c'_2 \ldots, c'_t\}$.

Definition 2. *A* minimally unsatisfiable subformula *of $wp(\rho)$, denoted by $mus(\rho)$, is a subformula of $wp(\rho)$ that becomes satisfiable whenever any of its clauses is removed. A* smallest cardinality MUS *of $wp(\rho)$, denoted by $smus(\rho)$, is an MUS such that for all $mus(\rho)$, $|smus(\rho)| \leq |mus(\rho)|$.*

In general, any unexplored paths that contain $mus(\rho)$ are infeasible and can be pruned. $wp(\rho)$ can have one or more MUSes; as a matter of succinctness, we keep track of $smus(\rho)$ for pruning purposes.

Next, we need to identify which statements are responsible for ρ's infeasibility and thus $smus(\rho)$.

Definition 3. *A* transforming statement *of a predicate c is an assignment statement* `s: v = e` *such that variable v appears in the transitive support of c.*

For example, the statement `s1:x = y+1` is a transforming statement of the condition $c : (x > 0)$, since $wp(s1, c)$ produces $c' : (y + 1 > 0)$. Statement `s2:y = z*10` is also a transforming statement of c, since $wp(s2, c')$ produces $(z * 10 + 1 > 0)$. During weakest precondition computations, only assignment statements can transform an existing conjunct c into a new conjunct c'. Branching statements can only add new conjuncts to the existing formulae, but cannot transform them. Given an execution path $\pi^{i,j} = s_i, \ldots, s_j$, we use $ts(\pi^{i,j}, c) \subseteq \{s_i, \ldots, s_j\}$ to denote the transforming statements for c.

Definition 4. *We define the* explanation *of the infeasibility of ρ to be the set of transforming statements* $explain(\rho) = \{s \mid s \in ts(\rho, smus(\rho))\}$.

2.4 Pruning Unexplored Paths

In this section we use examples to illustrate how to prune the path search space based on information obtained after learning.

The SSA form of the program of Figure 1 is represented graphically to its right. Assume the first explored execution π (highlighted in the figure) takes the `then`-branches at the two nondeterministic `if` statements. We would like to learn from π to prove unexplored paths. In the example, $\pi = \langle x_1 = 5, y_1 = 8, z_1 = 20, *, y_2 = y_1 - 1, y_4 = y_2, x_2 = 2x_1, *, z_2 = z_1 - 1, z_4 = z_2, \neg(x_2 > 10), halt\rangle$, which implies the infeasible sub-path $\rho = \langle x_1 = 5, y_1 = 8, z_1 = 20, *, y_2 = y_1 - 1, y_4 = y_2, x_2 = 2x_1, *, z_2 = z_1 - 1, z_4 = z_2, x_2 > 10\rangle$. According to Definition 1, we have:

$$wp(\rho) = (x_1 = 5) \wedge (y_1 = 8) \wedge (z_1 = 20) \wedge (true) \wedge (true) \wedge (2x_1 > 10) = false$$

The first three conjuncts come from the initial variable assignments and the next two ($true$) come from the nondeterministic conditionals. The last conjunct $2x_1 > 10$ is due to the deterministic conditional $x_2 > 10$ and the assignment $x_2 = 2x_1$. With a decision procedure, we can decide $smus(\rho) = (2x_1 \leq 10) \wedge (x_1 = 5)$. The explanation for ρ's infeasibility is $explain(\rho) = \{x_1 = 5, x_2 = 2 * x_1\}$. Therefore, we learned that any path containing these two assignments will not satisfy $x_2 > 10$; that is, any execution that contains $explain(\rho)$ can only take the `else`-branch to the conditional $x_2 > 10$. Since all the four possible paths contain $explain(\rho)$, none can reach the `abort` statement, which requires a path through the `then`-branch of the conditional $x_2 > 10$. Therefore, with SMT-based learning, a proof is obtained after only one execution.

A question that naturally arises from the example is what happens if a variable assigned in $explain(\rho)$ is subsequently reassigned? The answer is that if a variable is reassigned at s_i, then s_i will be included in $explain(\rho)$ if it is considered part of the explanation to ρ's infeasibility. For example, consider the program `foo2` which is the same as program `foo` of Figure 1 except for an additional assignment $x = x + 1$. The SSA form of `foo2` and its graphical representation is shown in Figure 2. Due to the new assignment $x = x + 1$ on Line 11, we need to add $x_4 = \varphi(x_2, x_3)$ on Line 12 to decide which version of x to use on Line 14. Assume the first execution, as highlighted in Figure 2, is $\pi_2 = \langle 0 : x_1 = 5, y_1 = 8, z_1 = 10, 1_T : *, 2 : y_2 = y_1 - 1, 5 : y_4 = y_2, 6 : x_2 = 2x_1, 7_T : *, 8 : z_2 = z_1 - 1, 12 : x_4 = x_2, 13 : z_4 = z_2, 14_E : \neg(x_4 > 10), 15 : halt\rangle$. From this, we can infer the infeasible execution segment $\rho_2 = \langle 0 : x_1 = 5, y_1 = 8, z_1 = 10, 1_T : *, 2 : y_2 = y_1 - 1, 5 : y_4 = y_2, 6 : x_2 = 2x_1, 7_T : *, 8 : z_2 = z_1 - 1, 12 : x_4 = x_2, 13 : z_4 = z_2, 14_T : x_4 > 10\rangle$. Based on an analysis similar to that used in the previous example, we have:

$$wp(\rho_2) = ((x_1 = 5) \wedge (y_1 = 8) \wedge (z_1 = 10) \wedge (true) \wedge (true) \wedge (2x_1 > 10)) = false$$

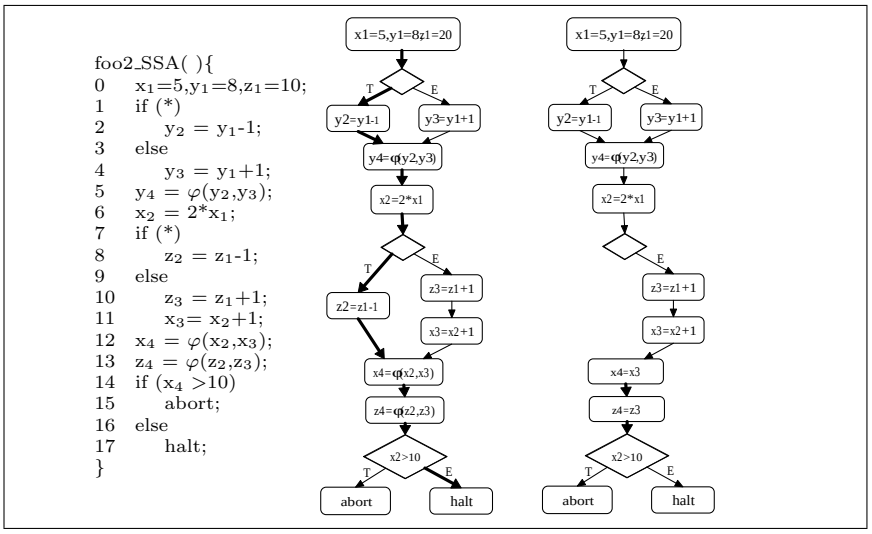

Fig. 2. A C program in SSA form (left), its graphical representation with a highlighted execution path (middle), and the remaining paths after learning from the highlighted path.

Although it results in the same $smus(\rho_2) = (2x_1 \leq 10) \wedge (x_1 = 5)$ as $smus(\rho_1)$, the explanation to $smus(\rho_2)$ is different: $explain(\rho_2) = \{x_1 = 5, x_2 = 2x_1, x_4 = x_2\}$. As a result, we can prune fewer paths than in the previous example of Figure 1. Figure 2(right) shows the remaining paths after pruning. Both of the remaining paths take the `else`-branch at the second nondeterministic `if` statement, which will go through $x_4 = x_3$. They cannot be pruned because neither path contains the statement $x_4 = x_2$ of $explain(\rho_2)$.

2.5 Path Reduction Algorithm

Algorithm 2 gives the pseudo-code that our `DPR-MC` algorithm uses in order to drive the execution of the program under verification along the path determined by a given k-oracle R. If the current statement s_i is an `abort` statement (Lines 3-4), an execution with a bug is found and the algorithm terminates. If s_i is a `halt` statement (Lines 5-6), the current execution is completed. An assignment is performed if s_i is an assignment statement (Lines 7-8). If s_i is a nondeterministic conditional (Lines 9-12), the algorithm checks if the threshold k has already been reached. If not, the algorithm follows the branch specified by oracle $R[j]$ and increase the value of j by 1; otherwise the algorithm breaks from the loop. If s_i is a deterministic conditional c (Lines 13-17), the value of c is computed and the corresponding branch is taken. Meanwhile, SMT-based learning is performed on the branch not taken, as shown in Algorithm 3, if the taken branch cannot reach the `abort` statement. Finally, the completed execution is removed from the unexplored oracle set (Line 20).

Algorithm 2 ExecuteFollowOracle(k-Oracle R, Set $\mathcal{R}$, int k)

```
1: i = j = 0;
2: while true do
3:    if s_i== abort then
4:       exit("report bug trace ⟨s_1, ..., s_i⟩");
5:    else if s_i == halt then
6:       break;
7:    else if s_i is an assignment then
8:       Perform the assignment;
9:    else if s_i is a nondeterministic conditional then
10:      if j == k break;
11:      follow oracle R[j];
12:      j++;
13:   else if s_i is deterministic conditional c with value true then
14:      LearnToPrune(⟨s_1, ..., s_{i-1}, ¬c⟩, R) if then-branch cannot reach abort;
15:   else if s_i is deterministic conditional c with value false then
16:      LearnToPrune(⟨s_1, ..., s_{i-1}, c⟩, R) if else-branch cannot reach abort;
17:   end if
18:   i++;
19: end while
20: R = R − {R};
```

The SMT-based learning procedure is given in Algorithm 3. The meaning of, and reason for, each statement, i.e., weakest-precondition computation, SMUS and transforming statements, have been explained in previous sections.

Algorithm 3 LearnToPrune(InfeasibleSubPath ρ, Set $\mathcal{R}$)

```
1: w = wp(ρ); // Perform weakest precondition computation
2: s = smus(w) // Compute smallest cardinality MUS
3: e = explain(s); // Obtain transforming statements
4: R = prune(R, e); //Remove all oracles in R that define paths containing e
```

3 Implicit Oracle Enumeration using SAT

One problem with Algorithm 1 is the need to save in $\mathcal{R}$ all k-oracles when model checking commences, the number of which can be exponential in k. In order to avoid this complexity, we show how Boolean formulae can be used to symbolically represent k-oracles.

Our discussion of the symbolic representation of k-oracles will be centered around loop-unrolled control flow graphs (CFGs), which can be viewed as directed acyclic graphs whose nodes are program statements and whose edges represent the control flow among statements. We shall assume that every loop-

unrolled CFG has a distinguished root node. The *statement depth* of a loop-unrolled CFG is the maximum number of statements along any complete path from the root. The *oracle depth* of a loop-unrolled CFG is the maximum number of nondeterministic conditional nodes along any complete path from the root.

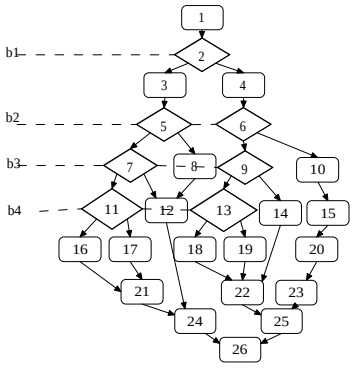

Fig. 3. An example control flow graph.

Figure 3 depicts a typical loop-unrolled CFG, where each node in the CFG has a unique index. Diamond-shaped nodes correspond to nondeterministic conditionals and rectangles are used for other statement types. The statement depth of this CFG is 10. As for its oracle depth, there are 7 nondeterministic conditionals divided into 4 levels (indicated by dotted lines); i.e., its oracle depth is 4.

To encode the choice made along a particular execution path at each level, we introduce the Boolean variables b_1, b_2, b_3 and b_4, with positive literal b_i indicating the `then`-branch and negative literal $\neg b_i$ indicating the `else`-branch. For example, path $\langle 1, 2, 4, 6, 9, 13, 19, 22, 25, 26\rangle$ is captured by $\neg b_1 \wedge b_2 \wedge b_3 \wedge \neg b_4$.

In general, a loop-unrolled CFG will have k levels of nondeterministic conditionals, and we will use k-oracles to explore its path space, with each k-oracle represented as a bit vector of the form $R = \langle b_1, b_2, \ldots, b_k\rangle$. As such, the valuation of Boolean variable b_i indicates an oracle's choice along an execution path at level i, and we call b_i an *oracle choice variable* (OCV). Such considerations lead to a symbolic implementation of the oracle space in which we use Boolean formulae over $b_i (1 \leq i \leq k)$ to encode k-oracles. For example, the Boolean formula $b_1b_2b_3b_4 + \neg b_1b_2\neg b_3$ encodes two paths through the CFG of Figure 3: $\langle 1, 2, 3, 5, 7, 11, 16, 21, 24, 26\rangle$ and $\langle 1, 2, 4, 6, 9, 14, 22, 25, 26\rangle$. In order to use modern SAT solvers, we maintain such Boolean formulae in conjunctive normal form (CNF).

Algorithm 4 presents a SAT-based implementation of Algorithm 1. It maintains a CNF $\mathcal{B}$ over k OCVs $\{b_1, b_2, \ldots, b_k\}$. Initially, $\mathcal{B}$ is a tautology; the while-loop continues until $\mathcal{B}$ becomes unsatisfiable. Inside the while-loop, we first use a SAT solver to find a k-oracle that is a solution of $\mathcal{B}$, and then perform the

program execution determined by the oracle. Algorithm 4 is essentially the same as Algorithm 1 except that: 1) oracle R and set $\mathcal{R}$ are represented symbolically by $\widehat{b}$ and $\mathcal{B}$, respectively; and 2) function calls to `LearnToPrune` (in algorithm `ExecuteFollowOracle`) are replaced by function calls to `SATLearnToPrune`, whose pseudo-code is given in Algorithm 5.

Algorithm 4 DPR-SATMC(PROGRAM C, INT k)

1: Let $b_i (1 \leq i \leq k)$ be k OCV variables, where k is C's oracle depth;
2: CNF $\mathcal{B} = true$;
3: **while** $\mathcal{B}$ is satisfiable **do**
4: Obtain a k-oracle $\widehat{b} = \langle \widehat{b_1} \widehat{b_2} \ldots \widehat{b_k} \rangle$ which is a solution of $\mathcal{B}$;
5: ExecuteFollowOracle($\widehat{b}, \mathcal{B}, k$);
6: **end while**
7: exit("No bug found up to oracle-depth k");

Let s be an assignment statement in an infeasible sub-path ρ. We define OCV_s to be the conjunction of those signed (positive or negative) OCVs within whose scope s falls. Also, given ρ's set of transforming statements $explain(\rho) = \{s_1, \ldots, s_t\}$, $OCV(explain(\rho)) = \wedge_{i=1}^{t} OCV_{s_i}$. Note that $OCV(\rho) \neq false$ as all statements in $explain(\rho)$ are along a single path. Further note that $explain(\rho)$ and $OCV(explain(\rho))$ can be simultaneously computed with one traversal of ρ: if a transforming statement s in $explain(\rho)$ is within the scope of a nondeterministic conditional, then the conditional's associated OCV variable is in OCV_s.

To illustrate these concepts, assume $explain(\rho) = \{1, 4, 22\}$ in the loop-unrolled CFG of Figure 3. Since node 1 can be reached from root node without going through any conditional branches, $OCV_1 = true$. Node 4 on the other hand is within the scope of the `else`-branch of nondeterministic conditional node 2 and thus $OCV_4 = \neg b_1$. Similarly, $OCV_{22} = \neg b_1 b_2$. Notice that the scopes of b_3 and b_4 close prior to node 22 and are therefore not included in OCV_{22}. Finally, $OCV(\rho) = OCV_1 \wedge OCV_4 \wedge OCV_{22} = \neg b_1 b_2$.

Algorithm 5 is our SAT-based implementation of Algorithm 3. $OCV(e)$ determines the set of paths containing all statements in $explain(\rho)$, and thus all paths that can be pruned. Let $OCV(e) = l_1 \wedge l_2 \wedge \ldots \wedge l_m$, where l_i is a literal denoting b_i or $\neg b_i$. Adding $\neg OCV(e) = \neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m$ to the CNF formula $\mathcal{B}$ will prevent the SAT solver from returning any solution (k-oracle) that has been pruned. We refer to $\neg OCV(e)$ as a *conflict clause*.

Note that the added conflict clause may prune multiple oracles, including the one just executed. Further note that when exploring a path by virtue of a given k-oracle, not all OCVs may be executed. For example, if the k-oracle in question is $b_1 \neg b_2 b_3 b_4$ in Figure 3, the actual execution path terminates after $\neg b_2$. In this case, the added conflict clause is $(\neg b_1 \vee b_2)$ instead of $(\neg b_1 \vee b_2 \vee \neg b_3 \vee \neg b_4)$.

To further illustrate Algorithms 4 and 5, consider once again the program of Figure 1. Suppose that the first path π_1 to be explored is the highlighted one in the figure. In this case, the infeasible sub-path ρ_1 to be considered is the

Algorithm 5 SATLearnToPrune(InfeasibleSubPath ρ, CNF $\mathcal{B}$)

1: $w = wp(\rho)$; // Perform weakest precondition computation
2: $s = smus(w)$; // Compute smallest cardinality MUS
3: $e = explain(s)$; // Obtain transforming statements
4: $b = OCV(e)$; //Obtain OCV on which e depends
5: let $b = l_1 \wedge l_2 \wedge \ldots \wedge l_m$ where l_i is a literal for b_i or $\neg b_i$;
6: $\mathcal{B} = \mathcal{B} \wedge (\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m)$;

same as π_1 except that the then-branch of the final deterministic conditional is taken leading to the `abort` statement. We then have that $smus(\rho_1) = (2x_1 \leq 10) \wedge (x_1 = 5)$ and the explanation for ρ_1's infeasibility is $explain(\rho_1) = \{x_1 = 5, x_2 = 2 * x_1\}$. Moreover, $OCV(e_1) = \mathit{true}$ as neither of the statements in $e_1 = explain(\rho)$ are in the scope of a nondeterministic conditional. The resulting conflict clause is *false* and adding (conjoining) it to $\mathcal{B}$ renders $\mathcal{B}$ unsatisfiable; i.e., all remaining paths can be pruned.

Consider next the program of Figure 2 and its highlighted execution π_2. As explained in Section 2, $smus(\rho_2) = smus(\rho_1)$, where ρ_2 is the infeasible subpath corresponding to π_2. However, the explanation for $smus(\rho_2)$, $explain(\rho_2) = \{x_1 = 5, x_2 = 2x_1, x_4 = x_2\}$, is different. Furthermore, $OCV(e_2) = b_2$, where $e_2 = explain(\rho_2)$, since the assignment $x_4 = x_2$ is within the scope of the `then`-branch of the second nondeterministic conditional. We thus add conflict clause $\neg b_2$ to $\mathcal{B}$, which results in the two remaining paths after pruning illustrated in Figure 2(right), both of which take the `else`-branch at the second nondeterministic conditional.

Theorem 1. *(Soundness and Completeness). Let C be a CFG that is loop-unrolled to statement depth D, and let ϕ be a safety property, the violation of which is represented by an* `abort` *statement in C. Then algorithm* `DPR-MC` *reports that the* `abort` *statement is reachable if and only if C violates ϕ within statement depth D.*

4 Experimental Evaluation

In order to assess the effectiveness of the DPR technique in the context of bounded model checking, we conducted several case studies involving well-known randomized algorithms. All results were obtained on a PC with a 3 GHz Intel Duo-Core processor with 4 GB of RAM running Fedora Core 7. We set a time limit of 500 seconds for each program execution.

In the first case study, we implemented a randomized algorithm for the MAX-3SAT problem. Given a 3-CNF formula (i.e., with at most 3 variables per clause), MAX-3SAT finds an assignment that satisfies the largest number of clauses. Obtaining an exact solution to MAX-3SAT is NP-hard. A randomized approximation algorithm independently sets each variable to 1 with probability 0.5 and to 0 with probability 0.5, and the number of satisfied clauses is then determined.

Table 1. Bounded model checking with DPR of Randomized MAX-3SAT.

vars	clauses	paths	explored	pruned	time w DPR(s)	time w/o DPR(s)
9	349	512	44	468	5.44	3.86
10	488	1024	264	760	13.77	7.61
11	660	2048	140	1908	9.67	15.58
12	867	4096	261	3835	14.53	30.59
13	1114	8192	1038	7154	49.61	70.10
14	1404	16384	965	15419	54.05	150.32
15	1740	32768	337	32431	25.58	300.80
16	2125	65536	2369	63167	49.32	Timeout
17	2564	131072	2024	129048	184.91	Timeout
18	3060	262144	1344	260800	175.34	Timeout
19	3615	524288	669	523619	110.14	Timeout

In our implementation, we inserted an unreachable `abort` statement; as such, all paths have to be explored to prove the absence of any reachable `abort` statement. Table 1 presents our experimental results for the randomized MAX-3SAT algorithm. Each row of the table contains the data for a randomly generated CNF instance, with Columns 1 and 2 listing the number of variables and clauses in the instance, respectively. Columns 3-5 respectively show the total number of execution paths, the number explored paths, and the number of pruned paths, with the sum of the latter two equal to the former. Finally, Columns 6-7 present the run time with DPR and the run time of executing all paths without DPR. From these results, we can observe that DPR is able to prune a significant number of the possible execution paths. Furthermore, the larger the CNF instance, the more effective dynamic path reduction is.

In our second case study, we implemented an algorithm that uses a Nondeterministic Finite Automaton (NFA) to recognize regular expressions for floating-point values of the form $[+]?[0-9]+\backslash.[0-9]+$. We encoded the accept state as an `abort` statement and verified whether it is reachable. Table 2 contains our experimental results on nine input sentences, among which five are valid floating-point expressions and four are not. Columns 1 and 2 give the length of the input

Table 2. Bounded model checking with DPR of NFA for floating-point expressions.

Benchmark			With DPR			Without DPR	
length	valid	paths	explored	pruned	time(s)	explored	time(s)
13	yes	8192	22	8166	0.707	2741	0.085
14	yes	16384	28	16356	0.845	10963	0.144
18	yes	262144	39	262105	2.312	175403	7.285
20	yes	1048576	29	1048542	4.183	350806	6.699
21	yes	2097152	26	2097097	4.202	175403	4.339
11	no	2048	15	2033	1.69	2048	10.027
13	no	4096	13	4083	0.52	4096	16.607
14	no	16384	8	16376	0.84	16384	53.358
20	no	1048576	28	1048548	3.32	-	Timeout

and whether or not it is accepted by the NFA. Column 3 lists the total number of execution paths Columns 4-6 contain the results using DPR, i.e. the number explored paths, the number of pruned paths and the run time. Columns 7 and 8 list the number of explored paths and run time without DPR. Note that in the case of a valid floating-point expression, the number of explored paths without DPR may not be the same as the number of total paths since the accept state is reached before exploring the remaining paths. As in the MAX-3SAT case study, we can again observe a very high percentage of pruned paths, a percentage that grows with the instance size.

5 Related Work

With dynamic path reduction, we perform symbolic analysis on program executions in order to learn and subsequently prune infeasible executions. Concolic testing and related approaches [8, 3, 11] also uses symbolic analysis of program executions—in conjunction with random testing—to generate new test inputs that can systematically direct program execution along alternative program paths. While these approaches can handle nondeterminism by introducing a Boolean input variable for each nondeterministic choice, they do not attempt to learn and prune infeasible paths. In fact, these testing procedures generate all possible paths and, for each such path, pass to a constraint solver the relevant constraints to determine the path's feasibility. Consequently, DPR can be beneficially used to reduce the path space these procedures explore.

We use weakest-precondition and minimally-unsatisfiable-core computations to identity interesting (transforming) statements along an execution path. Program slicing [17] also attempts to identify interesting program statements. There exists "precise" dynamic slicing algorithms that give the exact slice to any variable in an execution path [1, 19]. Here we use an example to show that dynamic slicing is less precise than DPR. Figure 4 shows a simple C program and its

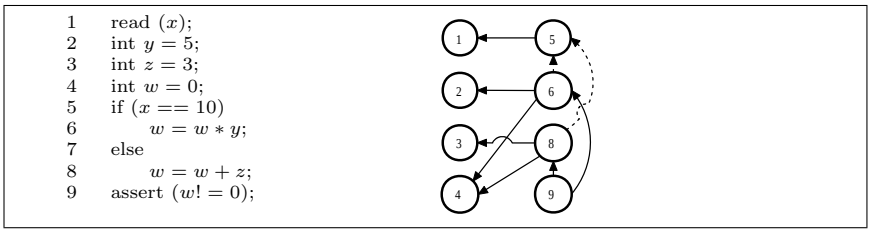

Fig. 4. A simple C program and its dependence graph.

dependence graph. The solid and dotted lines denote data and control dependencies, respectively. The static slice with respect to w on Line 9 is obtained by a forward traversal from Node 9 in the dependence graph: $\{1, 2, 3, 4, 5, 6, 8\}$. Assume the user input value to x is 10, the execution path $\pi = \langle 1, 2, 3, 4, 5, 6, 9 \rangle$

will report an assertion failure on Line 9. A dynamic analysis based on the dependence graph will give the "precise" slice that is responsible for the assertion failure: $\{1, 2, 4, 5, 6\}$. The dynamic slice is more precise than static slice because Line 8 is not in the execution and no statement is data- or control-dependent on Line 3. Finally, in our approach, we have $wp(\pi, w! = 0) \equiv (x = 10) \wedge (y = 5) \wedge (z = 3) \wedge (w = 0) \wedge (x = 10) \wedge (w * y! = 0) = \mathit{false}$. Apparently, $smus(\pi, w! = 0) \equiv (w = 0) \wedge (w * y! = 0)$, and $explain(\pi, w! = 0) = \{4, 6\}$, which is much smaller than the dynamic slice. The main reason that our approach is more precise than dynamic slicing is that dynamic slicing ignores values, and instead relies on analyzing the dependence graph. Although the dependence graph captures the dependence relation among statements, it contains no information about values and therefore cannot offer precise answer to questions involving values. In order to address this problem, we use decision procedures that can handle values. Since the pruning made possible by DPR is essential to its performance, the learned core obtained after each explored path needs to be as small as possible.

DPR achieves similar optimizations as non-chronological backtracking (NCB) [14] used in modern SAT and SMT solvers. In case of a conflict during, normal backtracking flips most recent open decision. However, such approach leads to redundant search (the same conflict will happen) if the most recent open decision does not cause the conflict. NCB avoids the redundant search by flips the most recent open decision contributing to conflict. NCB is made possible by maintaining a decision level for each variable and perform a learning when conflict happens. On the other hand, our learning is performed at the program language level. Although the objective (prune search space) is the same, the DPR algorithms have to be completely re-designed.

6 Conclusions

We have presented the new technique of dynamic path reduction (DPR) for software model checking. SMT-based learning techniques allow DPR to prune infeasible execution paths while model checking sequential software systems with nondeterministic conditionals. Our approach uses weakest-precondition and minimally-unsatisfiable-core computations to reveal the interesting (transforming) statements behind infeasible sub-paths. By determining the oracle control variables associated with these statements, unexplored paths containing the same unsatisfiable core can be efficiently and dynamically pruned. Our preliminary experimental results show that DPR can prune a significant percentage of execution paths, a percentage that grows with the instance size.

The language we currently handle is a subset of C allowing only one procedure and assignments, loops and (and possibly nondeterministic) conditional statements. There are no constraints placed on conditionals, but the constraint solver is able to handle linear constraints only. While we can analyze certain applications, future work will seek to extend the DPR technique to more general programs, including those with input statements.

References

1. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY, June 1990.
2. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.
3. C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. In *ACM conference on Computer and communications security (CCS)*, 2006.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
5. Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for dpll(t). In *International Conference on Computer Aided Verification (CAV)*, 2006.
6. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. 1996.
7. P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM.
8. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
10. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
11. R. Majumdar and K. Sen. Hybrid concolic testing. In *International Conference on Software Engineering (ICSE)*, 2007.
12. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.
13. D. Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification (CAV'93)*, pages 409–423, 1993.
14. Joao P. Marques Silva and Karem A. Sakallah. GRASP—a new search algorithm for satisfiability. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, 1996.
15. A. Valmari. Stubborn sets for reduced state generation. In *APN 90: Proceedings on Advances in Petri nets 1990*, pages 491–515, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
16. C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Disjunctive image computation for software verfication. *ACM Transactions on Design Automation of Electronic Systems*, 12(2), 2007.
17. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering (TSE)*, (4), 1982.
18. Z. Yang, C. Wang, F. Ivancic, and A. Gupta. Mixed symbolic representations for model checking software programs. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign (Memocode'06)*, 2006.
19. X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *IEEE/ACM International Conference on Software Engineering*, pages 319–329, 2003.