ExpliSAT: Guiding SAT-Based Software Verification with Explicit States

Sharon Barner[1], Cindy Eisner[1], Ziv Glazberg[1], Daniel Kroening[2], and Ishai Rabinovitz[3,⋆]

[1] IBM Haifa Research Lab
{sharon,eisner,glazberg}@il.ibm.com
[2] ETH Zürich
daniel.kroening@inf.ethz.ch
[3] Mellanox Technologies
ishai@mellanox.co.il

Abstract. We present a hybrid method for software model checking that combines explicit-state and symbolic techniques. Our method traverses the control flow graph of the program explicitly, and encodes the data values in a CNF formula, which we solve using a SAT solver. In order to avoid traversing control flow paths that do not correspond to a valid execution of the program we introduce the idea of a *representative* of a control path. We present favorable experimental results, which show that our method scales well both with regards to the non-deterministic data and the number of threads.

1 Introduction

In the hardware industry, *model checking* [6] is one of the most commonly used formal verification techniques. However, while computer programs are just as error-prone as circuitry, model checking has not yet been adopted by the software industry on a wide scale. Hardware model checkers do not perform well on software due to the state explosion problem, which is especially acute in software. Specialized software model checkers such as Spin [13], Zing [1] and VeriSoft [11] attempt to address this problem. However, most existing software model checkers use explicit-state enumeration, and thus, are unlikely to scale to programs that use large amounts of data.

In the past, we have applied symbolic methods to the software verification problem in order to enable the verification of programs with non-trivial amounts of data [3,4,8,17,9,18]. Other symbolic methods were introduced by NEC [15] and SLAM [2]. Symbolic model checking handles non-deterministic data efficiently, whereas explicit-state model checking handles non-deterministic scheduling of concurrent processes easily using partial order reduction [14]. An additional advantage of explicit-state model checking is that pointer dereferencing is trivial.

⋆ The work described in this paper was performed while the author was an employee of the IBM Haifa Research Lab.

We present a hybrid explicit-state and SAT-based method for software model checking. Our approach combines the merits of SAT-based symbolic model checking and explicit-state model checking using a hybrid data structure that stores both an explicit state vector and a symbolic CNF formula for the state. JPF [16] implements a similar hybrid approach and uses theorem proving to reason about the symbolic part. Our contribution is a method to guide the symbolic search towards legal program paths, using explicit values and a *representative*. As we show, the use of explicit values and a representative allows optimizations such as the Partial Order Reduction to be used in more cases than the simple hybrid approach of JPF.

Our method verifies invariant properties. We support user-specified assertions and additional implicit properties that must hold in every state, such as "no *NULL* pointer dereference", "no out-of-bounds array access", and "no data race".

Dynamic verification algorithms use sophisticated heuristics for guidance to likely locations of programming errors [10]. These heuristics are applicable for our method as well, as concrete program states are available.

We implemented our hybrid algorithm for concurrent C++ programs in a prototype named ExpliSAT. Our experimental results show that ExpliSAT outperforms state-of-the-art purely explicit or symbolic algorithms, and scales well both with regards to the non-deterministic data and the number of threads.

Related Work. Though most model checkers for software use explicit-state enumeration, e.g., Spin [13], Zing [1], and Verisoft [11], there exist several purely symbolic model checkers. CBMC [17] performs symbolic simulation on sequential programs. It addresses the path-explosion problem by transforming the program into static single assignment (SSA) form [7]. TCBMC [18] is an extension of the algorithm to concurrent programs. In contrast to CBMC, the approach presented in this paper is based on building SSA for single control paths only, which results in significantly smaller SAT instances. In addition, a pre-determined bound on the number of loop iterations or recursion steps is not required.

JPF [16], originally an explicit-state model checker for Java Bytecode, now features a hybrid state representation similar to the one we propose. JPF instruments the code such that it builds a symbolic formula when executed. JPF utilizes a theorem prover for examining the symbolic representation, as opposed to our use of a SAT solver. In JPF, the full symbolic formula is passed to the decision procedure; no attempt is made to reduce its size. JPF tries to avoid exploring non-legal control paths by calling the theorem prover to check whether there exists an execution that follows the path. In contrast, our method of guiding the symbolic search allows us to avoid exploration of non-legal control paths without calling the SAT solver in most cases.

DART [12] integrates symbolic execution into a random test generator. It also replaces symbolic values by explicit values, but only if the solver is unable to handle the symbolic constraint. Explicit values are not used to simplify constraints. Concurrency is not supported in DART but it is supported in its successor, CUTE [19].

Outline. The rest of the paper is organized as follows. Section 2 states the preliminaries. Section 3 shows how to explore only the control paths of the program explicitly while representing all executions symbolically. Section 4 defines the representative and shows how it can be used to guide the search. Section 5 presents our experimental results.

2 Preliminaries and Definitions

We first define the *control flow graph* (CFG), which is an abstract representation of a program. A vertex in a CFG represents a program statement, and there is a designated vertex representing the initial statement of the program. An edge in the CFG represents the ability of the program to change the control location.

For the purpose of this paper, we consider the CFG to be "flat". That is, the CFG considers the whole program as one piece, in which procedure calls are inlined. Therefore the CFG may be infinite. Note that such a CFG spares us the effort of paying special attention to the call stack.

Definition 1 (CFG). *A control flow graph (*CFG*) is a directed graph $G = \langle V, E, \mu \rangle$ where V is the set of vertices, E is the set of edges, and $\mu \in V$ is the initial vertex.*

We assume that each conditional statement has only one condition, i.e., the vertex v of a conditional statement has an out-degree of exactly 2, and we define the guard of an edge based on the condition of its source. The guard represents the condition that must be satisfied if the program changes its control location by traversing the edge.

Definition 2 ($cond(v)$, $guard(e)$). *Let $cond(v)$ denote the condition of a conditional statement v. For an edge $e = (v, u)$, $guard(e)$ equals true if v is not a conditional statement, $cond(v)$ if the edge is traversed when the condition is satisfied, and $\neg cond(v)$ otherwise.*

Given a concurrent program, we build a CFG representing it using the CFGs of its threads as follows. We denote the CFG of thread t by $G_t = \langle V_t, E_t, \mu_t \rangle$, and the guard labeling function of thread t by $guard_t(e)$. The control flow graph G of the concurrent program is $\langle V, E, \mu \rangle$ where $V = V_1 \times V_2 \times \ldots$, $\mu = (\mu_1, \mu_2, \ldots)$ and E contains edges $(\bar{v}, \bar{u})$ such that only one thread changes its control location. Formally, $(\bar{v}, \bar{u}) \in E$ with $\bar{v} = (v_1, v_2, \ldots)$ and $\bar{u} = (u_1, u_2, \ldots)$ iff $\exists_k.(v_k, u_k) \in E_k \wedge \bigwedge_{i \neq k} v_i = u_i$. The guard of $(\bar{v}, \bar{u})$ is equal to the guard of the thread that makes the transition. If k denotes that thread, then $guard((\bar{v}, \bar{u})) = guard_k((v_k, u_k))$.

Definition 3 (Explicit State). *An* explicit state *s of the program is a triple $\langle t, v, L \rangle$, where $t \in \mathbb{N}$ denotes the number of threads, $v \in V$ denotes the current control location of each thread (i.e., the vertex in the CFG) and L denotes the valuation of all the variables of the program over their domain. We write $s \models \varphi$ iff the predicate φ evaluates to true if evaluated using L, and $s \not\models \varphi$ otherwise.*

Definition 4 (Kripke Structure of a Program). *Let the CFG of the program be given by the triple $\langle V, E, \mu \rangle$. The* Kripke structure *of that program is the triple $\langle S, I, T \rangle$ where S is the set of explicit states, $I = \{\langle t, v, l \rangle \mid v \in \mu\} \subset S$ is the set of initial states of the program and $T = \{(\langle t_1, v_1, l_1 \rangle, \langle t_2, v_2, l_2 \rangle) \mid \exists e = (v_1, v_2) \in E$ s.t. $\langle t_1, v_1, l_1 \rangle \models guard(e)\}$ is the set of transitions between the states.*

Definition 5 (Execution). *An* execution *π of a program is a sequence of explicit states $(s_1, s_2, \ldots, s_n)$ s.t. $s_1 \in I$ and for every $1 \leq i < n. (s_i, s_{i+1}) \in T$. A state s is said to be* reachable *iff there exists an execution π that contains s.*

The property we are interested in is *reachability* of states s that violate a given predicate $p(v)$, where v is the control location of s. As an example, if v is a user-specified assertion with condition x, $p(v)$ is $\neg x$.

Definition 6 (Control Path). *A* control path *c of a program is a path through the CFG of the program, i.e., a finite sequence $(v_1, \ldots, v_n)$ of nodes of the CFG where $v_1 = \mu$ and $\forall_{1 \leq i < n}.(v_i, v_{i+1}) \in E$. The set of control paths is denoted by $\mathcal{C}$. If c is a projection of an execution π on V, we call c a* legal *control path.*

We denote the projection of an execution π onto the CFG by $cp(\pi)$. The execution π is said to follow the control path $cp(\pi)$. There may be many different executions that follow the same control path. They differ only in the data (i.e., the valuation of the variables).

The *Static Single Assignment (SSA)* [7] form is a representation of a program in which every variable is assigned exactly once. Existing variables in the original representation are split into versions. New variables are distinguished from the original name with a subscript such that every assignment has a unique left hand side. In SSA form, the function returning a non-deterministically chosen input *input*() is replaced by an indexed variable $input_i$, which denotes the value returned by the i^{th} call to the *input*() function.

CBMC [17] transforms a whole program into SSA. In contrast to that, we only consider the SSA of a control path, which is a much simpler transformation, as a control path is linear whereas a program is branching. The variable x in a particular assignment to x may be indexed differently in different control paths.

Since every variable in the SSA form of a control path is assigned exactly once, it can be considered as a set of constraints that must be satisfied in any execution that follows that control path. We denote the conjunction of the SSA constraints of a control path c by $SSA(c)$.

Definition 7 (Path guard). *The* path guard *of a control path c is denoted by $cpg(c)$ and is the conjunction of the guards of all edges in c given that c is in SSA form:*

$$cpg(v_1, v_2, \ldots, v_n) = \bigwedge_{1 \leq i < n} guard(v_i, v_{i+1}).$$

3 Symbolic Verification Using Explicit CFG Traversal

3.1 The Naïve Hybrid Algorithm

We propose a model checking technique that traverses the abstract representation of the program represented by the CFG, rather than the Kripke structure of the program. That is, we explicitly explore only the control paths and use symbolic methods to cover the various executions.

We define the following equivalence relation over all executions of a program:

Definition 8 (Control equivalent). *Two executions π_1, π_2 are said to be* control equivalent*, denoted $\pi_1 \sim \pi_2$, iff they follow the same control path, i.e., $cp(\pi_1) = cp(\pi_2)$.*

The equivalence classes that this relation induces serve us in decreasing the size of the software model on which we perform an explicit search. We explicitly traverse each control path in the CFG. Each such control path is a representative for all the executions in the control equivalence class. Note that not every control path in the CFG has an associated execution. We first present a naïve algorithm that ignores this detail, then introduce the technique we use to avoid traversing such control paths.

The naïve algorithm traverses all control paths of the CFG and maintains a CNF encoding of each control path. The encoding is constructed from the SSA form of the path using the constraints given by the guards and the assignments to variables. We use a propositional SAT solver for searching for a satisfying assignment to the encoding of the path and the negation of the property associated with the last control location. We benefit from the fact that SAT-solvers are known to be practically efficient for large number of variables.

```
1.   a = 1;
2.   if (a>0) {
3.      a = input();
4.      if (a ≤ 1) {
5.         c = a+2;
6.         assert(c<3);
7.      }
8.      if (a≤1)
9.         a=2;
10.     else
11.        a=1;
12.  }
```

Fig. 1. A non-deterministic program

```
1.   $a_1$ = 1;
2.   if ($a_1$>0) {
3.      $a_2$ = $input_1$;
4.      if ($a_2$ ≤ 1) {
5.         $c_1$ = $a_2$+2;
6.         assert($c_1$<3);
```

```
1.   $a_1$ = 1;
2.   if ($a_1$>0) {
3.      $a_2$ = $input_1$;
4.      if ($a_2$ ≤ 1) {
8.      if ($a_2$≤1)
10.     else
11.        $a_3$=1;
```

Fig. 2. The paths π_1 and π_2

If a satisfying assignment is found, there exists an execution that follows the control path and violates the property. The satisfying assignment contains a valuation of the non-deterministic choices made on the control path, and thus, the extraction of a counterexample is straight-forward.

For example, consider the program in Fig. 1. Two possible control paths π_1 and π_2 of this program are shown in their SSA form in Fig. 2. The guards of these two paths are:

$$\begin{aligned} cpg(\pi_1) &\iff (a_1 > 0) \wedge (a_2 \leq 1) \\ cpg(\pi_2) &\iff (a_1 > 0) \wedge \neg(a_2 \leq 1) \end{aligned}$$

The SSA constraints of these control paths are given by the following two equivalences:

$$\begin{aligned} SSA(\pi_1) &\iff (a_1 = 1) \wedge (a_2 = input_1) \wedge (c_1 = a_2 + 2) \\ SSA(\pi_2) &\iff (a_1 = 1) \wedge (a_2 = input_1) \wedge (a_3 = 1) \end{aligned}$$

Let $c = (v_1, \ldots, v_n)$ be a control path. If there exists a satisfying assignment to $\zeta \equiv cpg(c) \wedge SSA(c) \wedge \neg p(v_n)$, then there exists a reachable state s with control location v_n that violates the property $p(v_n)$. By transforming ζ to CNF, we can use a SAT solver to check if there exists an execution that follows c and violates the property. For example, for π_1:

$$\begin{aligned} \zeta \equiv\ & (a_1 > 0) \wedge (a_2 \leq 1) \wedge \\ & (a_1 = 1) \wedge (a_2 = input_1) \wedge \\ & (c_1 = a_2 + 2) \wedge \neg(c_1 < 3) \end{aligned}$$

3.2 Simplifying Constraints Using Explicit Values

The naïve algorithm proposed above is improved using the notion of explicit values.

Definition 9 (Explicit Values). *For a given control path c in SSA form, a variable has an* explicit value *if any two executions π_1, π_2 with $\pi_1 \sim \pi_2 \wedge cp(\pi_1) = cp(\pi_2) = c$ assign the same value to this variable. If a variable does not have an explicit value, it has a* symbolic value.

Obviously, if an expression has an explicit value, it can be replaced by a constant during the traversal of the control path. Symbolic values may differ from one execution to the other even if both executions follow the same control path, while explicit values do not differ.

Refer to control path π_1 in Fig. 2. The value of variable a_1 is an explicit value, as every execution that follows this control path assigns 1 to a_1. On the other hand, the value of variable a_2 is symbolic, as the value of a_2 may differ in different executions that follow this control path.

Note that explicit values may depend on non-deterministic choices. In Fig. 3 the value of x depends on $input()$ yet it is explicit in the control path (1, 3, 4),

```
1.  if (input())
2.     x=3;
3.  else
4.     x=2;
```

Fig. 3. The value of variable x is explicit on the path (1,3,4) even though it depends on non-deterministic data

```
1.  x=input()
2.  if (x==3)
3.     y=0;
4.  else
5.     z=6;
```

Fig. 4. The value of variable x is explicit on the path (1,2,3). Syntactically, we approximate it as a symbolic value.

since every execution that follows this path assigns the value 2 to x. Fig. 4 shows another example, in which there exists a control path (1,2,3) on which x is always assigned 3, and thus, has an explicit value.

As identifying whether a value is explicit in a given control path is hard, we make a syntactic, but sound approximation: Values returned by *input*() are symbolic, and values that are a result of an assignment of symbolic values are also symbolic. In any other case, the value is explicit. We denote the set of explicit values for a given control path c by $\mathcal{E}(c)$. In some cases, our approximation classifies explicit values as symbolic. For instance, it classifies x in Fig. 4 as symbolic on the control path (1,2,3). However, it never classifies a symbolic value as explicit.

Definition 10 (Restricted Guards). *Let $e \in E$ denote an edge in the CFG and c denote a control path. We define the* restricted guard $guard'(e,c)$ *as a function over $e \in E$ and c. If $guard(e)$ has an explicit value in c, then $guard'(e,c)$ is that explicit value, otherwise $guard'(e,c) = guard(e)$. The* restricted path guard $cpg'(c)$ *is the conjunction of the restricted guards of the edges on $c = (v_1, v_2, \dots, v_n)$, i.e., $cpg'(c) = \bigwedge_{1 \le i < n} guard'((v_i, v_{i+1}), c)$.*

Pointers. The value of a pointer is often, at least in part, an explicit value. Though a pointer holds the address of a memory location, which may change from one execution to the other, it conceptually points to an object. Pointers can therefore be seen as a tuple $\langle base, offset \rangle$ where *base* is the base object and *offset* is the offset within that object. The value of a pointer is generally determined in four different ways:

– Explicit address of an object (e.g., operator &): In this case, the value of the pointer is always explicit.
– Memory allocation (e.g., `malloc`): In this case, the pointer may point to any free memory location, hence its value is symbolic. However, since conceptually, the value is a fresh cell location in the heap, we may optimize this and treat its value as an explicit value: the *base* object is given the value of a fresh object that was not used before. This optimization may conceal certain bugs that might occur as a result of accessing an adjacent memory

```
foo( int* p) {
  ...
}
```

Fig. 5. When examining *foo* procedure on its own, the value of the pointer p is symbolic, and is equivalent to p=*input*()

```
void main() {
  int a,b;
  foo(&a);
  foo(&b);
}
```

Fig. 6. When examining the entire program, the value of the pointer p in *foo* is explicit: in the first instance it is the address of a and in the second it is the address of b

location or using values of an already freed memory location. However, since we detect an invalid pointer dereferencing and the use of uninitialized values, this optimization can be used without concealing any bugs. Therefore, the value of a pointer is explicit in this case.

- Non deterministic value (e.g., input to the verified program): both the *base* object and the *offset* have symbolic values.
- Pointer arithmetic (e.g., $p = q + 1$): The *base* value is left unchanged with regards to the original pointer. If it was an explicit value, it remains so. The value of *offset* is explicit only if the original pointer's *offset* value is explicit and the arithmetic calculation uses solely explicit values.

When examining a program from its beginning, as opposed to starting the verification from a procedure with parameters, most pointers are of explicit value. In Figure 5 the value of the pointer p is non-deterministic since the procedure *foo* is examined as a stand-alone function. In Figure 6, the value of p is explicit in each call of the *foo* function. When verifying a complete program, the value of *base* is usually an explicit value. Thus, dereferencing of pointers, which is problematic for most symbolic model checkers, is usually simple using our hybrid state representation. Let b denote the object indicated by *base*. An expression $*p$ with $p = \langle base, offset \rangle$ in which the value of *base* is explicit, can be replaced by b if b is a simple or struct type, and by $b[offset]$ if b is of an array type.

When a pointer has an explicit value, pointer dereferencing is done the same way as in explicit model checking. Otherwise, if its value is symbolic, we may encode its value in the $SSA(c)$, similarly to CBMC [17], or use lazy initialization, as described in JPF [16]. Note that when *base* is explicit, as it is usually the case, both methods are improved, since they take into account only the different possible *offset*s and not all the existing objects.

When computing ζ for verifying a control path, expressions that have an explicit value can be replaced by a constant, as justified by the following lemma. We can use the restricted path guard $cpg'(c)$ (Def. 10) instead of the full path guard $cpg(c)$ (Def. 7).

Lemma 1. *Let $c = (v_1, \ldots, v_n)$ be a control path. There exists a satisfying assignment to $\zeta' \equiv cpg'(c) \wedge SSA(c) \wedge \neg p(v_n)$ iff it also satisfies $\zeta \equiv cpg(c) \wedge SSA(c) \wedge \neg p(v_n)$.*

Continuing our previous example, the guards are simplified as follows:

$$cpg'(\pi_1) \iff (a_2 \leq 1)$$
$$cpg'(\pi_2) \iff \neg(a_2 \leq 1)$$

An additional improvement is that no call to the SAT solver is needed for properties for which $p(v_n)$ is an explicit value. For data intensive programs as well as programs with complicated control graphs, the bring up time of the SAT solver is not always negligible.

Lemma 2. *If the property $p(v_n)$ has an explicit value in the legal control path $c = (v_1, \ldots v_n)$, then there exists a satisfying assignment to $cpg'(c) \wedge SSA(c) \wedge \neg p(v_n)$ iff the explicit value of $p(v_n)$ is false.*

Note that the implicit properties that we add, such as "no data-race", are verified by evaluating a predicate over each state. The naïve algorithm invokes the SAT solver for each location in order to verify these predicates. If the predicate has an explicit value, the SAT solver is not needed.

4 The Path Representative

We introduce the concept of the *path representative.* Path representatives allow us an easy way of filtering out control paths that need not be traversed because they do not have an associated execution path.

Definition 11 (Representative). *Let c be a control path in its SSA form. A* representative *ρ of a control path is a valuation of all the variables in c such that the guards in the control path are satisfied, i.e., $\rho \models SSA(c) \wedge cpg(c)$. We denote the set of representatives by $\mathcal{R}$.*

Lemma 3. *Let c be a control path in its SSA form. A representative ρ exists for a control path c iff there exists an execution $\pi_r = (s_1, s_2, \ldots, s_n)$ with $cp(\pi_r) = c$.*

The proof makes use of the fact that c is in SSA form, and thus, the last state, s_n, holds the values of all the variables that were assigned at any point during the execution.

In our example in Fig. 1, the path $(1, 2, 3, 4, 8, 9)$ has no representative (because if $a \leq 1$ then 5 and 6 would be on the path as well). Guiding the CFG traversal using a path representative ensures that the traversed path is a legal path. The CFG traversal is guided by the path representative, traversing the control path it follows. Full coverage of the CFG is gained by using a representative of each control equivalence class.

The algorithm we propose is illustrated by means of pseudo code in Fig. 7. The algorithm maintains a hybrid representation $\langle c, \rho \rangle$ of the paths that are

```
// Variables: Priority queue $\mathcal{Q} \subseteq (\mathcal{C} \times (\mathcal{R} \cup \{\bot\}))$
// of control paths and path representatives
HybridReachability($P$)
1  Compute initial state $c_I$
2  $\mathcal{Q}:=\{\langle c_I, \bot\rangle\}$;
3  while ($\mathcal{Q} \neq \emptyset$)
4          Let $\langle c, \rho\rangle \in \mathcal{Q}$;
5          $\mathcal{Q}:=\mathcal{Q} \setminus \langle c, \rho\rangle$;
6          if $\rho = \bot$ then $\rho$:=GetRepresentative($c$);
7          if $\rho = \bot$ then continue;
8          if CheckProperty($c, \rho$) then return true;
9          $\mathcal{Q} := \mathcal{Q} \cup$ GetSuccessors($P, c, \rho$);
10 end
11 return false;
```

Fig. 7. High Level Description of the Hybrid Reachability Algorithm

explored. The first component $c \in \mathcal{C}$ is a control path. The second part ρ is either a path representative, i.e., a valuation to the state variables, or $\bot$, which denotes the case that the assignment has not yet been computed.

The algorithm uses a priority queue $\mathcal{Q}$ of hybrid path representatives that are to be explored. In line 2, the initial state is put into the queue. While the queue is non-empty, a search heuristic removes a hybrid path representative $\langle c, \rho\rangle$ from the queue (lines 4 and 5). The algorithm checks if there is already a representative for c. If $\rho = \bot$, GetRepresentative(c) is called to compute a representative for c (line 6). If such a representative does not exist, c is not a legal control path and thus, is not examined (line 7).

In line 8, the algorithm proceeds by calling CheckProperty. If the property is violated, then a counterexample was found.

Finally, the successors of the last location of c are computed by GetSuccessors(P, c, ρ), appended to c, and added into the queue $\mathcal{Q}$ (line 9).

The procedure CheckProperty(c, ρ) determines if the last vertex of the path c is an assertion (Fig. 8). If so, it checks if the condition p of the assertion has an explicit value (line 3). If so, ρ provides the truth value of p for all executions that follow c. If p is symbolic, a SAT solver is used to check if p can be violated (line 6). The formula passed to the SAT solver uses the restricted guard simplified using the explicit values given by ρ.

4.1 Computing the Path Representative

The procedure GetSuccessors(P, c, ρ) computes the successor states of the last state v_n of a given control path $c = (v_1, \ldots, v_n)$. If v_n is not a conditional, the computation of v_{n+1} and a representative ρ' for $(v_1, \ldots, v_n, v_{n+1})$ is straightforward. If v_n contains a non-deterministic choice ι, $\rho'(\iota)$ is simply an arbitrary

```
   CheckProperty($c \in \mathcal{C}, \rho \in \mathcal{R}$)
1  Let $c = (v_1, \ldots, v_n)$;
2  if $v_n = \mathsf{assert}(p)$ then
3        if $p \in \mathcal{E}(c)$ then
4              if $\rho(p) = \mathsf{false}$ then return true;
5        else
6              if IsSatisfiable(
7                          $cpg'(c) \wedge SSA(c) \wedge \neg p$) then
8                    return true;
9        endif
10 endif
11 return false;
```

Fig. 8. Checking assertions using the path representative

but constant value. By choosing one possible value of ι, we maintain a valid representative of the explored control path. Thus, if we have an explicit value for the condition, we can traverse a conditional statement without a call to our SAT solver, as opposed to the call to the theorem prover needed by JPF [16]. However, since we also maintain a symbolic representation of the value of ι, backtracking to v_n, as performed by explicit state model checkers such as SPIN, is not necessary. If the value of ι affects future control decisions, different valuations are examined when other representatives are computed using GetRepresentative(c).

If v_n is a conditional, let $v'(\alpha)$ denote the successor of v_n for a given truth value $\alpha \in \{\mathsf{true}, \mathsf{false}\}$ of $cond(v_n)$. If $cond(v_n)$ has an explicit value, let $t = \rho(cond(v_n))$ denote that value. The only successor of $\langle c, \rho \rangle$ is $\langle (c, v'(t)), \rho \rangle$.

If $cond(v_n)$ is symbolic, we still compute the truth value t of the guard given by the path representative ρ. Two successors are computed:

1. $\langle (c, v'(t)), \rho \rangle$ corresponds to the branch suggested by the truth assignment made by the representative. Note that ρ is also a representative for $(c, v'(t))$.
2. $\langle (c, v'(\neg t)), \bot \rangle$ corresponds to the other branch. A new representative has to be computed for this control path.

The priority queue $\mathcal{Q}$ should usually give preference to pairs that have a path representative ρ. Only when a pair $\langle c, \bot \rangle$ is chosen, GetRepresentative(c) is called to compute a new representative using a SAT solver. If c is not a legal control path GetRepresentative(c) returns $\bot$, and c is not explored. When a pair $\langle c, \rho \rangle$ is chosen, the SAT solver is not utilized. The main benefit of using the representative is that in at least half of the cases, we avoid calling GetRepresentative(c) and hence avoid utilizing a SAT solver.

The GetRepresentative procedure can be improved by using previous representatives as initial partial assignments of the formula of the examined control path. Also note that the GetRepresentative procedure is likely to produce a large amount of similar SAT instances, and thus, an incremental SAT solver that preserves previously learnt clauses, when applicable, should be used.

4.2 Concurrency

For concurrent programs, we explore all paths for any order of execution of the statements in the different threads. The number of such executions is typically exponential in the number of program statements. In traditional explicit-state model checking, partial order reductions (POR) are applied to significantly reduce the size of the traversed model. In contrast to other symbolic methods that are unable to apply this reduction, the application of POR is trivial in our hybrid method. Thus, we are able to reduce the size of traversed model without affecting the correctness of the results. POR is usually difficult if there are pointer dereferencing operators in a statement. In contrast, for example, to JPF we often have explicit or at least partially explicit values for pointers, and therefore in many cases can avoid this difficulty.

In contrast to sequential programs, if two control paths c_1, c_2 differ only in the scheduling of the threads, c_1 and c_2 may have the same representative. Thus, when backtracking to a control path c_2 that only differs in its scheduling from another control path c_1 we already traversed, we can use the representative of c_1 instead of computing a new one. This optimization avoids unnecessary calls to the SAT solver. Hence, the path representative does not guide the CFG traversal when the branching of the CFG is the outcome of different possible schedulings.

5 Experimental Results

We have implemented our hybrid algorithm in a prototype named *ExpliSAT*. ExpliSAT utilizes an internal IBM state-of-the-art SAT solver named Mage. ExpliSAT verifies C and the POSIX thread library. ExpliSAT supports heap memory allocation and dynamic thread creation. Like all explicit model checkers, ExpliSAT will not terminate in the case of unbounded recursion. However, unlike symbolic methods in the case of bounded recursion, the user is not required to provide the bound up front. Instead, heuristics may be utilized on-the-fly in order to decide how deep ExpliSAT should explore the unbounded recursion or loop. Of course, in such cases, ExpliSAT does not perform as a model checker, but rather it is simply a bug-hunting tool.

5.1 Case Studies

ExpliSAT was used internally inside IBM to verify several protocols and code segments. In this subsection we review how ExpliSAT is used to improve IBM's products quality.

ExpliSAT examined a complex locking protocol in an industrial middleware software. A prototype of the mechanism was devised, which was verified using

ExpliSAT. The prototype has about 250 lines of code, in which 5 threads exercise the locking mechanism in a non-deterministic order.

ExpliSAT detects a write-write data race in the protocol within approximately 20 seconds. The race results from a subtle definition of the critical section.

ExpliSAT was also used in the verification of communication protocol between three controllers in IBM microcode. This protocol, which is designed for a Linux device driver, should withstand failures of controllers. During a failure of a controller it may exhibit limited Byzantine behavior. The prototype of this protocol has about 400 lines of code, in which 2 controllers pass 6 random messages between them. Two additional processes are used in order to simulate the random failures of the controllers.

ExpliSAT detects that due to a failure of the receiving controller, the sending controller may not be informed that the message was in fact received. Since the sending controller is uncertain that the message was received, it may send a duplicate message. Hence, the protocol should be revised and handle such duplicate messages.

For more information on these two protocols and their verification refer to [5].

An additional verification effort using ExpliSAT was made on synchronization code that was extracted from an IBM random test generator tool. In this code segment, one thread makes a non-deterministic choice which should be passed to all the other threads as well. This process may be repeated several times.

We employed the synchronization code using 4 threads. The threads were allowed to make up to 4 non-deterministic choices. The code segment ExpliSAT examines has about 100 lines.

ExpliSAT verifies that all the threads are synchronized on all the non-deterministic choices. However, it detects a possible deadlock on a specific scheduling. The entire inspection of this code takes ExpliSAT approximately three minutes.

5.2 Artificial Examples

We evaluate the performance benefit of using a path representative using a sequential benchmark. We compare the performance of ExpliSAT when it utilizes a path representative for guiding the CFG traversal with its performance when such guidance is not utilized. As is explained in section 4.2, the path represen-

Table 1. Run time comparison of two versions of the naïve algorithm and ExpliSAT on “Bubble-sort” benchmark. The number of bits in the non-deterministic input is denoted by b, s denotes size of array.

	Naïve algorithm						ExpliSAT		
	Late reachability check			Early reachability check					
Benchmark	b=8	b=16	b=32	b=8	b=16	b=32	b=8	b=16	b=32
Bubble-sort s=3	0.43s	0.43s	0.56s	2.01s	2.04 s	4.05s	0.60s	0.55s	0.85s
Bubble-sort s=4	7.44s	8.43s	12.4s	23.46s	30.37s	43.04 s	5.91s	4.94s	8.59s
Bubble-sort s=5	768.33s	885.95s	1708.26s	245.76s	282.98s	458.28s	69.58s	52.89s	90.95s

tative is used to decide what conditional branch to explore, but not for deciding what thread schedule to follow. Hence, we compare the performance of ExpliSAT with the naïve algorithm on a sequential benchmark. The benchmark we use is a sorting algorithm parameterized in the array size.

We compare ExpliSAT with two different versions of the naïve algorithm. In one version, reachability of a state is verified only when a property is examined, i.e., only when reaching an `assert` statement. In this version, the algorithm traverses the entire CFG including all the non-legal control paths. In the second version, the reachability analysis is done at an earlier stage. As soon as a branch in the CFG is encountered, the tool checks feasibility of each of the branches. This version never traverses a non-legal control path. JPF implements such a CFG traversal as well.

As can be seen in Table 1, ExpliSAT scales better than both versions of the naïve algorithm. It is interesting to note that the early reachability analysis is not necessarily better than the late reachability analysis. Using an array size of 4, the late reachability method outperforms the early reachability method. This can be ascribed to the overhead of verifying reachability for every vertex in the CFG. This overhead is worthwhile only if there are a significant number of non-legal paths in the CFG. As an example, this seems to be the case when using array size 5.

Table 2. Run time comparison of Zing and ExpliSAT on three classes of benchmarks. The number of bits in the non-deterministic input is denoted by b. In "Bubble-sort", s denotes size of array, In "Producers", p denotes number of producers and c denotes number of consumers.

	Zing					ExpliSAT		
Benchmark	b=2	b=4	b=6	b=8	b=16	b=8	b=16	b=32
Bubble-sort s=3	1s	3s	277s	>2h	>2h	0.60s	0.55s	0.85s
Bubble-sort s=4	1s	58s	>2h	>2h	>2h	5.91s	4.94s	8.59s
Bubble-sort s=5	1s	>2h	>2h	>2h	>2h	69.58s	52.89s	90.95s
Producers p=1 c=1	1s	21s	317s	>2h	>2h	38.38s	41.03s	40.95s
Producers p=1 c=2	41s	690s	>2h	>2h	>2h	73.38s	78.28s	78.57s
Producers p=1 c=3	1160s	>2h	>2h	>2h	>2h	130.82s	131.17s	141.24s
Producers p=2 c=2	18s	230s	>2h	>2h	>2h	0.6s	0.63s	0.65s
Producers p=2 c=3	443s	>2h	>2h	>2h	>2h	0.66s	0.71s	0.70s
Producers p=2 c=4	>2h	>2h	>2h	>2h	>2h	0.72s	0.75s	0.75s
Random-Choice	<1s	<1s	<1s	4s	>2h	0.33s	0.34s	0.43s

Using three other code examples, we compare the performance of ExpliSAT with Zing from Microsoft Research [1], a state-of-the-art explicit-state model checker for software. Note that Zing and ExpliSAT are executed on different platforms[1].

The first benchmark is the same as in Table 1. As the number of legal control paths is exponential in the array size, the performance of ExpliSAT deteriorates

[1] Zing was executed on Windows while ExpliSAT was executed on Linux. Both were executed on a Pentium 4 with 2 GHz and 1GB of memory.

when the array size is increased. Still, ExpliSAT scales better than Zing on this parameter. The second comparison is a producer-consumer protocol that has a bug in the producer code. Atomicity of the critical section is not enforced, and thus, two producers may overflow the shared buffer. Both ExpliSAT and Zing verify the correctness of the program if only one producer exists, and detect the bug when two producers co-exist. We compare the performance in the number of active consumers and producers. The third program is "Random-choice". This is a program with two threads, where both threads utilize the same function to compute a value. The two threads assert that the same value was computed. Under some rare conditions the computed values are different.

We also compare ExpliSAT and TCBMC [18] (Table 3)[2]. The benchmark program has two threads that sort the same array using the bubble-sort algorithm. Even though atomicity of the critical sections is maintained, for some inputs and a specific scheduling, the threads fail to sort the array correctly. Unlike ExpliSAT, TCBMC requires a bound on the number of context-switches, which is denoted by n. ExpliSAT performs much better than TCBMC. It also scales better in the array size. Though both methods are SAT-based, ExpliSAT performs better since it provides the SAT solver with several small CNF formulas unlike TCBMC which searches a satisfying assignment to one big CNF formula. ExpliSAT allows the SAT solver to slice different literals from each CNF according to the specific clauses this CNF entails. As opposed to TCBMC whose CNF formula encodes all calculations in the program, including those who are irrelevant for finding the bug that exists in the program.

Table 3. Run time comparison of TCBMC and ExpliSAT. The size of the array is denoted by s. The number of bits in the non-deterministic input is denoted by b. The bound on the number of context-switch TCBMC enforces is denoted by n.

	TCBMC						ExpliSAT		
Benchmark	b=8		b=16		b=32		b=8	b=16	b=32
	n=6	n=10	n=6	n=10	n=6	n=10			
Buggy-sort s=3	0.4s	0.2s	3.6s	4.0s	20.3s	48.3s	4.68s	4.72s	5.88s
Buggy-sort s=4	11.5s	1.3s	14.6s	58.7s	135.2s	323.0s	43.55s	37.01s	44.79s
Buggy-sort s=5	71.0s	94.1s	125.7s	3013.0s	1124.0s	$>$ 1h	140.43s	131.86s	154.44s

On all benchmarks, ExpliSAT scales much better than Zing, TCBMC and the two versions of the naïve algorithm in the size of the non-deterministic input. The effect the bit-vector size has on the performance of ExpliSAT is marginal.

6 Conclusion and Future Work

We have presented a novel algorithm for software verification that combines explicit and symbolic methods. Experimental results show that this hybrid representation outperforms both conventional explicit-state model checking and

[2] We provide the figures for TCBMC from [18] for reference.

purely symbolic methods. The symbolic part of the representation allows the method to scale with an increasing amount of non-deterministic data, while the explicit state enables powerful search and state-space reduction techniques, such as partial order reduction. In comparison to previous hybrid approaches, our contributions are the concepts of an *explicit value* and a *representative*, which are exploited to reduce the size of the verification conditions as well as the number of calls to the SAT solver. In addition, they allow wider application of explicit techniques such as partial order reduction, for instance, in the case of many pointers.

For future work, we plan to investigate automatic slicing of the formulas according to the assertions, and the merging of control flow paths in order to reduce the number of formulas to be checked. Another promising research direction is to use the proof of unsatisfiability of verification conditions to direct the search towards an error during backtracking.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR*, 2004.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
3. S. Barner, Z. Glazberg, and I. Rabinovitz. Wolf - bug hunter for concurrent software using formal methods. In *CAV*, pages 153–157, 2005.
4. S. Barner and I. Rabinovitz. Effcient symbolic model checking of software using partial disjunctive partitioning. In *CHARME*, pages 35–50, 2003.
5. H. Chockler, E. Farchi, Z. Glazberg, B. Godlin, Y. Nir-Buchbinder, and I. Rabinovitz. Formal verification of concurrent software: two case studies. In *Proceedings of 4th International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2006.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
7. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35. ACM, 1989.
8. C. Eisner. Model checking the garbage collection mechanism of SMV. In *ENTCS*, volume 55. Elsevier Science Publishers, 2001.
9. C. Eisner. Formal verification of software source code through semi-automatic modeling. *Software and Systems Modeling*, 4(1):14–31, February 2005.
10. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, page 286b. IEEE, 2003.
11. P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *CAV*, pages 476–479. Springer, 1997.
12. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.
13. G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
14. G. Holzmann and D. Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211. Chapman & Hall, 1994.
15. F. Ivancic, Z. Yang, A. Gupta, M. K. Ganai, and P. Ashar. Efficient SAT-based bounded model checking for software verification, 2004.

16. S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.
17. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.
18. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.
19. K. Sen and G. Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *CAV. Tool Paper*, 2006.