Symbolic model checking for asynchronous Boolean programs

Byron Cook
Microsoft Research

Daniel Kroening
ETH Zurich

Natasha Sharygina
Carnegie Mellon University

Abstract. Software model checking problems generally contain two different types of non-determinism: 1) non-deterministically chosen values; 2) the choice of interleaving among threads. Most modern software model checkers can handle only one source of non-determinism efficiently, but not both. This paper describes a SAT-based model checker for asynchronous Boolean programs that handles both sources effectively. We address the first type of non-determinism with a form of symbolic execution and fix-point detection. We address the second source of non-determinism using a symbolic and dynamic partial-order reduction, which is implemented inside the SAT-solver's case-splitting algorithm. The preliminary experimental results show that the new algorithm outperforms the existing software model checkers on large benchmarks.

1 Introduction

Model checking [1] is a formal verification technique for detecting behavioral anomalies in system descriptions. In recent years, a number of model checkers have been built specifically for the analysis of software. These tools have uncovered defects that would have otherwise gone undetected. However, they do not scale gracefully when applied to software of substantial size. Thus, much of the research on model checking has focused on improving scalability.

The size of the state space of a system is directly related to the amount of non-determinism present in the model. Concurrent software with asynchronous interleaving semantics has two sources of non-determinism: 1) Non-deterministic choice of data values, given explicitly in the program, and 2) the non-deterministic choice of the interleavings between the threads.

Powerful techniques have been developed to address both of these forms of non-determinism. Partial-order reduction is specifically designed to mitigate the concurrency between threads. Symbolic data structures concisely represent large sets of states. Unfortunately, these two techniques are difficult to combine. For this reason, with few exceptions, model checkers for software systems tend to come in one of two flavors: *Symbolic software model checkers* are strong when proving properties of programs with symbolic data but are not good at reasoning about concurrent programs with many threads; *Explicit-state model checkers* have powerful methods for the verification of programs with multiple threads, but are not useful when applied to systems with significant amounts of symbolic data.

In this paper, we propose a model checking algorithm that efficiently analyzes programs with both non-deterministic data values and multiple threads of execution.

The algorithm is limited to Boolean programs [2, 3] extended with asynchronous threads [4, 5]. Boolean programs—which are like C programs, but are limited to variables with type `bool`—have become a common model for tools that implement counterexample-guided abstraction refinement for software verification. Boolean programs allow the programmer to choose values non-deterministically. We restrict ourselves to non-recursive programs, which we have found to be acceptable when performing analysis on system-level code. We also restrict the set of properties that can be verified to those that can be expressed in terms of reachability.

The algorithm described in this paper can be used immediately from within software model checkers such as SLAM [6] or BLAST [7]. These model checkers implement software predicate abstraction, i.e., they abstract a C program into a Boolean program. Using SLAM, we can now verify properties of device drivers with an accurate representation of the threads together with abstract representations of their environments.

The contribution of this paper is a method for combining SAT-based symbolic model checking and the partial-order reduction. We represent the states states symbolically using a parametric representation [8, 9]. The data structure grows linearly in the number of execution steps, even in the presence of non-deterministically chosen data values. As the parametric representation is not canonical, the fix-point detection becomes harder. We use solvers for Quantified Boolean Formulae (QBF) for this task. We leverage the recent remarkable improvements in this technology [10, 11].

We use a propositional logic SAT-solver as part of the symbolic simulation algorithm. This allows us to implement a form of partial-order reduction as a modification of the SAT-solver. The key idea behind this method is that the case-splitting algorithm used within backtracking-based SAT-solvers can be modified to eliminate undesired interleavings. This turns out to be much faster than alternative combination methods, such as adding constraints to the the query that is passed to the SAT-solver. The resulting reduction is dynamic, as the choice of interleaving depends on the particular set of states found during the reachability analysis.

The remainder of this paper is organized as follows. We provide some background on Boolean programs in Section 2. We then describe our algorithm in the Sections 3 and 4. We describe the results for our experimental evaluations in Section 5. In Section 6, we conclude and discuss some ideas for future work.

Related Work Several model checkers support sequential Boolean programs. BEBOP [2] and MOPED [3] are BDD-based symbolic model checkers, and both handle recursive procedures. In principle, because BOPPO supports only a fixed number of threads and non-recursive procedures, the threaded programs could be converted into sequential programs that BEBOP and MOPED could process. This is not practical, however, because only a lightweight and static form of partial-order reduction could be applied during the translation, rather then the dynamic one that BOPPO employs.

DIZZY [12] uses SAT-based symbolic simulation. The fix-point detection is done by computing BDDs representing the set of reachable states. Our work uses a similar

algorithm, but uses QBF for the fix-point detection and also implements support for multiple threads using partial-order reduction.

Several previous efforts have also applied model checking to Boolean programs with asynchronous threads. For example, Jain, Clarke and Kroening [5] use the BDD-based model checker NuSMV [13] to verify concurrent Boolean programs with only very limited success.

Forms of partial-order reduction for explicit-state model checking (examples include [14, 15]) has been a particularly effective technique for verifying programs and protocals with many threads. Fore example, Ball, Chaki and Rajamani [4] describe a partial-order reduction based explicit state model checker, called Beacon, for asynchronous Boolean programs. Beacon, however, was overly sensitive to the occurrence of symbolic data generated by Slam.

The idea of combining symbolic reasoning with partial-order reduction is not new. Our proposal shares a great deal of motivation with Alur *et al.* [16], who describe a method of combining partial-order reduction together with a BDD-based symbolic model checker. Their algorithm first computes a constrained transition relation, called an *ample transition relation*. This is then given to a BDD-based model checker. Our experiments indicate that this technique does not provide much benefit in the context of SAT-solvers. The overhead of adding static constraints to the SAT-solver's data structure seems to abate the potential benefit of less state-space exploration. As it turns out, many of the constraints that are added are actually never used, resulting in wasted effort. Our implementation, which simply limits the assignments from which the SAT-solver can choose when case-splitting, requires less overhead when computing representative paths. In [17], the reduction is applied before passing the model to a Bounded Model Checker (BMC).

In [18], Lerda, Sinha and Theobald integrate partial-order reduction into a BDD-based model checker, as opposed to a pre-processing step. This approach is similar to our proposal. The difference between this previous work and our proposal is in the representations of data, the class of solvers used, and methods of implementing the dynamic partial-order reduction. Whereas they use BDDs, we use SAT and QBF solvers and must therefore implement the partial-order reduction within the SAT-solver in a different manner.

Several methods address the problem of scalability in the presence of threads and non-deterministically chosen data via forms of decomposition [19, 20]. These techniques usually either sacrifice some amount of completeness or require small amounts of intervention from the user. The advantage of these approaches is that the analysis is much more scalable. In the future, researchers interested in thread modular approaches may be able to use our method of combining partial-order reduction and symbolic reachability in a way that allows them to improve on the completeness and user-interaction required.

Unsound approaches have also proved successful in finding bugs in concurrent programs. For example, Qadeer & Rehof [21] note that many bugs can be found when the analysis is limited to execution traces with only a small set of context-switches. This analysis supports recursive programs. Our approach complements

these techniques because, while they are unsound, they are able to analyize a larger set of programs.

2 Boolean Programs

2.1 Boolean Programs and Predicate Abstraction

Predicate abstraction [22, 23] is a commonly used method for systematically constructing conservative abstractions of software. When combined with reachability analysis and an automatic abstraction refinement mechanism, it forms an effective model checking strategy. Predicate abstraction constructs the abstraction by tracking only certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Extra non-determinism is added into the abstraction in order to maintain soundness of the sequential control-flow constructs in the abstraction. When predicate abstraction is performed on software systems with threads, the result is an abstraction that makes fundamental use of both non-deterministically chosen values and non-deterministically scheduled threads. Therefore, we need an efficient reachability analysis for these abstract models.

The following example shows code that is typical of a Windows device driver:

```
void DecrementIo(DEVICE_OBJECT *  DeviceObject) {
    EXT * ext = (EXT*)DeviceObject->DeviceExtension;
    int IoIsPending = InterlockedDecrement (&ext->IoIsPending);
    if (!IoIsPending) { KeSetEvent (&ext->event, IO_NO_INCREMENT, FALSE); }
}
```

An abstraction of this function is obtained by passing it to SLAM [6]. In the first iteration of the abstraction refinement loop, SLAM computes the following Boolean program fragment:

```
void DecrementIo_abstraction() {
    InterlockedDecrement_abstraction();
    goto L1,L2;
    L1: KeSetEvent_abstraction();
    L2: return;
}
```

This example demonstrates how predicate abstraction generates Boolean programs that make non-trivial use of both forms of non-determinism. This abstraction is using a non-deterministic `goto` instruction to model the conditional operator in the original function. This code fragment is also calling an abstraction of the Windows kernel synchronization primitive `KeSetEvent`.

In further refinement iterations, SLAM usually adds variables to the abstraction. Suppose the following predicates are used to refine the abstraction above:

$$\begin{aligned}\{\ & \texttt{b1} \triangleq \texttt{ext} == \&\texttt{envext},\ \texttt{b2} \triangleq \texttt{envext.IoIsPending} == 1\\ ,\ & \texttt{b3} \triangleq \texttt{envext.IoIsPending} == 2,\ \texttt{b4} \triangleq \texttt{IoIsPending} == 2\\ ,\ & \texttt{b5} \triangleq \texttt{IoIsPending} == 1,\ \texttt{b6} \triangleq (*\texttt{ext}).\texttt{IoIsPending} == 1\\ ,\ & \texttt{b7} \triangleq (*\texttt{ext}).\texttt{IoIsPending} == 2\}\end{aligned}$$

This results in the following new abstract model:

```
bool b1,b2,b3;
void DecrementIo_abstraction() {
    bool b4,b5,b6,b7;
    b1,b6,b7 = *,*,*
     constrain((!(b1' && b2) || b6') && (!(b1' && b3) || b7'));
    b4,b5 = InterlockedDecrement_abstraction(b6,b7);
    goto L1,L2;
    L1: assume(!b4 && !b5);
        KeSetEvent_abstraction();
    L2: return;
}
```

Due to the imprecision of the abstraction, we cannot prove that `ext==&envext`, nor can we prove that `ext!=&envext`. Therefore, a non-deterministically chosen value has to be assigned to the variable `b1`, which represents this predicate. This is necessary to preserve the soundness of the analysis.

Furthermore, using the `constrain` operator, this assignment statement restricts the choice such that `b6` must be true after the assignment if `b1` is true after the assignment and `b2` is true before the assignment. Analogously, `b7` must be true after the assignment if `b1` is true after the assignment and `b7` is true before the assignment. This abstraction also refines the non-deterministic `goto` using an `assume` statement: the program declares that any transition passing through the `L1` location must ensure that `b4` and `b5` are false.

2.2 Formal Semantics of Boolean programs

In this section, we provide a simple operational semantics for asynchronous, concurrent Boolean programs. Later, in Section 3.2, we use the semantics to construct an algorithm that transforms Boolean program reachability into a propositional logic formula. The formalization is based on the description of sequential Boolean programs in [2].

Definition 1. *An* explicit state η *of a Boolean program is a tuple* (i, Ω)*, with* $PCs : T \mapsto L$ *and* $\Omega : V \mapsto \mathbb{B}$.

The first component of an explicit state η, called i, is a mapping from the set of threads T into the set of program locations L. $i(t)$ denotes the instruction that is to be executed next by thread $t \in T$. The second component, called Ω, is a mapping from the set of variables V into the set of the two Boolean values, i.e., it assigns an explicit value to each state variable.

Notation Given a valuation Ω and an expression e over the variables V, we use $\Omega(e)$ in order to denote the evaluation of e. This is defined in the usual way. In addition to that, we also allow expressions that refer to the values of variables in two different states η_1 and η_2. Syntactically, the values of the two states are distinguished by using primed versions of the variables. We use $(\eta_1, \eta_2)(e)$ in order to denote the evaluation of e in the states η_1 and η_2. The unprimed variables in e are substituted by the values given in η_1, while the primed variables in e are substituted by the values given in η_2. As an example, consider the valuation $\Omega_1 = \{(x, 1), (y, 0)\}$ and $\Omega_2 = \{(x, 0), (y, 0)\}$. For these valuations, and an expression $e = x \vee x'$, we have $(\eta_1, \eta_2)(e) = 1 \vee 0$.

We also allow additional choice variables $\iota_1, \ldots, \iota_k$ inside the expressions. We use ι to denote the vector of these variables. Given a particular non-deterministic choice ι and a state η, we denote the evaluation of the expression e in η with the choice ι as $(\eta, \iota)(e)$.

Given an explicit state η, we denote the first component by $\eta.i$, and the second component by $\eta.\Omega$. For any function $f : D \to T$, we define $f[d/r] : D \to R$ as $f[d/r](x) = r$ if $d = x$, and $f[d/r](x) = f(d)$ otherwise.

Execution Semantics Assume the scheduler picks thread $t \in T$ to execute in state η. We use $\eta_1 \to_t \eta_2$ to denote the fact that a transition from state η_1 is made to η_2 by executing one statement of thread t. The statement that is executed is $P(i(t))$. The relation $\eta_1 \to_t \eta_2$ is defined by a case-split on this instruction. The conditions for each statement are shown in Table 1. We explain the formalization of each statement as follows:

- The `skip` statement increments the program counter of thread t. The values of the variables and the program counters of the other threads do not change.
- The `goto` $\theta_1, \ldots, \theta_k$ statement changes the program counter of thread t to one of the program locations $\theta_1, \ldots, \theta_k$ given as argument. The choice is arbitrary, i.e., non-deterministic. The values of the variables and the program counters of the other threads do not change.
- The `assume` e statement behaves like `skip`, but with the additional constraint that the expression e must evaluate to `true` in state η_1. If the expression evaluates to false, η_1 has no successor states.
- The constrained assignment statement $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e changes the program counter like `skip`. It also updates the values of the variables using the expressions $e_1, \ldots, e_k$. The expressions are evaluated in state η_1. The expressions may contain choice variables $\iota_1, \ldots, \iota_k$. These variables allow a non-deterministic choice on data, and are quantified existentially.
  The transition also has an additional constraint e. The constraint e is a predicate in terms of the current state η_1 and the next state η_2. It is evaluated in both states accordingly, where the next state variables are primed. If there is no choice for ι, which satisfies the constraint, state η_1 has no successor states.

We do not define semantics for syntactic sugar such as `if` or `while`, as these statements can easily transformed using `goto` and `assume`, as illustrated in section 2.1. Also, function calls can be inlined; we do not support unbounded recursion, as the reachability problem for concurrent programs with unbounded recursion is undecidable.

Finally, we write $\eta_1 \to \eta_2$ if there exists a thread $t \in T$ such that $\eta_1 \to_t \eta_2$. We say that there is a transition from η_1 to η_2 in this case, or that η_1 is reachable from η_2 with one transition.

A state η_2 is reachable from a state η_1 in k transitions if there exists a state η', η' is reachable from η_1 in $k-1$ transitions, and η_2 is reachable from η' in one transition. Given an initial state η_I, the set of reachable states is the set of states that is reachable from η_I in any number of transitions. We property we check is reachability of states with particular program locations.

$P(i_1)$	i_2	Ω_2
`skip`	$i_2(x) = i_1[t/i_1(t)+1]$	$\Omega_2 = \Omega_1$
`goto` $\theta_1, \ldots, \theta_k$	$i_2(x) = i_1[t/\theta_1] \vee \ldots \vee$ $i_2(x) = i_1[t/\theta_k]$	$\Omega_2 = \Omega_1$
`assume` e	$i_2(x) = i_1[t/i_1(t)+1]$	$\Omega_2 = \Omega_1 \wedge$ $\Omega_1(e) = \mathsf{true}$
$x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e	$i_2(x) = i_1[t/i_1(t)+1]$	$\exists \iota.\ \Omega_2 = (\Omega_1[x_1/(\Omega_1, \iota)(e_1)]$ $\ldots [x_k/(\Omega_1, \iota)(e_k)] \wedge$ $(\eta_1, \eta_2, \iota)(e)$

Table 1. Conditions on the explicit state transition $\langle i_1, \Omega_1 \rangle \rightarrow_t \langle i_2, \Omega_2 \rangle$, for each type of statement $P(i_1)$.

3 SAT-based Symbolic Simulation

In this section we describe how we represent a set of states symbolically using formulae, and then how to transform Boolean programs into such formulae.

3.1 Representation of States

Definition 2. *A symbolic formula is defined using the following syntax rules:*

1. *The Boolean constants* true *and* false *are formulae.*
2. *The non-deterministic choice variables $\iota_1, \ldots$ are formulae.*
3. *If f_1 and f_2 are formulae, then $f_1 \wedge f_2$, $f_1 \vee f_2$, and $\neg f_1$ are formulae.*

The set of such formulae is denoted by $\mathcal{F}$.

A symbolic formula may evaluate to multiple values due to the choice variables. As an example, the pair of formulae $\langle \iota_1, \iota_2 \wedge \neg \iota_1 \rangle$ may evaluate to $\langle 0, 0 \rangle$, $\langle 1, 0 \rangle$, $\langle 0, 1 \rangle$, but not to $\langle 1, 1 \rangle$. We use these symbolic formulae in order to represent a sets of states:

Definition 3. *A* symbolic state σ *is a triple $\langle i, \omega, \gamma \rangle$, with $i : T \mapsto L$, $\omega : V \mapsto \mathcal{F}$, and $\gamma : \mathcal{F}$.*

Given a particular valuation for the choice variables ι, we denote the value of a symbolic formula f as $\iota(f)$.

The first component of a symbolic state σ, called i, is identical to the first component of an explicit state (definition 1). The second component, called ω, is a mapping from the set of variables V into the set of formulae. It denotes the *symbolic* valuation of the state variables. The third component, called γ, is a formula that represents the guard of the state symbolically.

Thus, we represent the program counters explicitly, while the program variables are represented symbolically. The set of explicit states represented by σ are those states η that satisfy the following conditions:

– They have the same PC values given by i.

$$\eta.i = \sigma.i \tag{1}$$

– There exists a non-deterministic choice ι, which satisfies the guard γ, and assigns values to the variables that match the values given by Ω.

$$\exists \iota.\iota(\gamma) \ \wedge \ \forall v \in V.\Omega(v) = \iota(\omega(v)) \tag{2}$$

Thus, the set of explicit states corresponding to a symbolic state is defined using a predicate in the parameter ι. Thus, we have a *parametric representation*. Parametric representations of sets of states have been used in formal verification before [8, 9], but mostly in the context of hardware verification.

Note that the problem of whether there exists an explicit state represented by a given symbolic state is equivalent to the problem of propositional satisfiability. A satisfying assignment contains concrete valuations for the state variables and for the choice variables, and thus, a SAT-solver provides a witness.

3.2 Symbolic Execution

Assume that the scheduler picks thread $t \in T$ to execute in the symbolic state σ. In analogy to the explicit state model, we use $\sigma_1 \rightarrow_t \sigma_2$ to denote the fact that a transition from state σ_1 is made to σ_2 by executing one statement of thread t. Again, the statement that is executed is $P(i(t))$. The definition of the relation $\sigma_1 \rightarrow_t \sigma_2$ is done using a case-split on this instruction. The conditions for each statements are shown in table 2. The column describing the constraints on the program counters i_1 and i_2 is identical to the column in table 1, and therefore not repeated here. We explain the formalization of each statement as follows:

– The definitions of the `skip` and `goto` statement follow the definitions for the explicit state case. The formulae for the guards are not changed by these statements.
– In the symbolic case, the `assume` e statement does not have the precondition that e is true. Instead, the condition e is instantiated in the state σ_1. This results in a symbolic formula. The symbolic formula is conjoined with the guard γ_1, forming the formula γ_2.
– In the symbolic case, a constrained assignment statement $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e updates the values of the variables using the expressions $e_1, \ldots, e_k$. The expressions are evaluated in state η_1. It is no longer necessary to instantiate the values of the non-deterministic choice variables ι, as $\omega(v)$ is now a formula, and not a Boolean value. Thus, the choice variables become part of the formula. Also, the additional constraint e is added to the guard, in analogy to an `assume` statement.

3.3 Reachability Algorithm

In order to check reachability of a particular program location $b \in L$ using the symbolic model, we implement an exhaustive search of the state space. This is done by most explicit state model checkers as well, e.g., by SPIN [24]. The basic algorithm is shown in Figure 1. The main differences between our implementation and an explicit state model checker are as follows:

1) We maintain a queue of symbolic states for the search. A search heuristic picks the next state to explore from the queue.

$P(i_1)$	ω_2	γ_2
`skip`	$\omega_2 = \omega_1$	$\gamma_2 = \gamma_1$
`goto` $\theta_1, \ldots, \theta_k$	$\omega_2 = \omega_1$	$\gamma_2 = \gamma_1$
`assume` e	$\omega_2 = \omega_1$	$\gamma 2 = (\gamma_1 \wedge \omega_1(e))$
$x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e	$\omega_2 = (\omega_1[x_1/\omega_1(e_1)] \ldots [x_k/\omega_1(e_k)]$	$\gamma 2 = (\gamma_1 \wedge (\omega_1, \omega_2)(e))$

Table 2. Conditions on the symbolic transition $\langle i_1, \omega_1, \gamma_1 \rangle \rightarrow_t \langle i_2, \omega_2, \gamma_2 \rangle$, for each type of statement $P(i_1)$. For the constraints on i_1 and i_2, see table 1.

2) Before reachability of a bad state σ can be concluded, we must run a SAT solver (denoted by the function IsSatisfiable) in order to check that $\sigma.\gamma$ is satisfiable, and thus, the set of concrete states represented by σ is non-empty. Note that the guards of the states on one path only get stronger, and never weaker, and thus, it is sufficient to check the guards of the bad states only.

3) In order to conclude that no bad states are reachable, explicit state model checkers maintain a history of the states that have been explored. This set of states is typically organized using a hash table. Because of the symbolic representation, we cannot use this approach. Instead, we use a symbolic solver in order to compare the symbolic state that is chosen next to explore with the states that have been explored so far. This is implemented in the procedure IsHistory. The details of this function are described in section 4.

3.4 Partial-Order Reduction

When computing the successors of a given symbolic state σ, we usually have to consider the possibility that any of the treads $t \in T$ can make a transition. The choice is non-deterministic. Formally, we have to compute all states σ' for which a thread $t \in T$ exists which can make a transition from σ to $state'$. A sequence of choices for a particular thread t is called an *interleaving*. The problem is that the number of states explored can grow dramatically with the number of threads. Even with just two threads, the number of interleavings blows up in the number of execution steps. In contrast to that, a sequential program only requires as many symbolic states as there are execution steps.

The purpose of *Partial-Order Reduction* [15] is to reduce the number of states that have to be explored. This is done in a way that preserves the property, i.e., the property holds on the reduced model if and only if it holds on the full, original model.

Symbolic Partial-Order Reduction using SAT The approach we take is related to what many explicit state model checkers implement. We aim at finding a thread t that makes an *invisible* transition, i.e., a transition which is independent from a transition made by any other thread $t' \neq t$. We compute the sets of variables written and read by each of the threads. Let R_t denote the set of variables that are read, and W_t the set of variables that are written by thread t in the current state. If thread t is not enabled, these sets are empty. If a thread t is found with $W_t \cap (\bigcup_{i \neq t} R_i \cup W_i) = \emptyset$

```
// Input: Boolean Program P with locations L, bad location b ∈ L
// Output: true iff b is reachable in P
// Variables: Queue Q of symbolic states
SymbolicReachability(P, b)
1  Compute initial state σ_I
2  Q := {σ_I};
3  while (¬Q ≠ ∅)
4        σ := Element from Q;
5        if IsHistory(σ) then
6              Q := Q \ σ;
7        elseif ∃t ∈ T. σ.i(t) = b ∧ IsSatisfiable(σ.γ) then
8              return true;
9        else
10             Q := (Q \ σ) ∪ GetSuccessors(P, σ);
11       endif
12 end
13 return false;
```

Fig. 1. High Level Description of the Symbolic Reachability Algorithm

and $R_t \cap \bigcup_{i \neq t} W_i = \emptyset$, we only explore the successors generated by executing t. All other transitions are discarded.

This reduction preserves the property we are checking, i.e., reachability of program locations. The computation of reduction requires knowledge of the enabled transitions and of the dependencies between the transitions. This is computationally inexpensive in case of an explicit state model checker, as all the values of the variables are known. In contrast, we use a symbolic representation. The question of whether a particular transition is enabled or not corresponds to a SAT instance. A syntactic over-approximation of the set of enabled transitions and the dependencies is feasible, but often does not result in a significant reduction. We therefore use a modified SAT solver in order to compute the set of interleavings we explore.

SAT has been used in the context of asynchronous transition systems before. As in most existing approaches, we build a SAT instance that has non-deterministically chosen variables for the thread selector and an encoding of the transitions out of the given state. Typically, constraints on the thread selector variables are added upfront in order to limit the possible choice of interleavings. However, our initial experiments showed that most of these constraints are unnecessary, as they eliminate transitions out of states that are unreachable, and often make the instance much harder.

We therefore use the following, alternative approach: the SAT instance we form uses a one-hot encoding for a thread about to make an invisible transition. We implement the constraints on the variables that are read and written as part of the case-splitting heuristic of ZChaff, and not by adding appropriate clauses, as

this information is known statically. The SAT-solver only needs to determine which threads are enabled, i.e., have a satisfiable guard.

Once a local interleaving is found, it is explored. If no local interleaving is found, the thread to be executed is chosen by the SAT-solver's decision heuristic. Once its successors are computed, we add a blocking clause to prevent the same transition from being explored again and backtrack.

Cycle Detection The method of removing interleavings that we described above could lead to unsound results. In fact, there is a possibility that some transitions will be delayed forever because of a cycle in the reduced model.

To prevent the loss of transitions, partial-order reduction techniques require satisfaction of a *cycle condition* [25]. The cycle condition prohibits cycles that contains a state in which some transition is enabled, but is never taken for any state on the cycle. The intuitive reason for this condition is to avoid postponing a transition indefinitely while generating the reduced model.

Algorithmically, we solve this issue in the same way as most explicit state model checkers: when postponing a transition, we note this fact on the search stack. If the HISTORY procedure detects that a state has been explored before, we resume the evaluation of the postponed transitions.

4 Fix-point Detection

In order to detect fix-points, we need to compare the new set of states to the set of states that we have already explored. When using BDDs, two sets of states can be compared by simply comparing the graphs of the BDDs. The drawback of using BDDs is that already only very few steps of symbolic simulation may result in prohibitively large BDDs.

As described in the previous section, we store the states using a non-canonical symbolic representation. While this representation allows us to execute a statement symbolically in linear time, we pay a price in form of a harder fix-point detection problem.

The fix-point detection is implemented in the HISTORY procedure. It takes a new symbolic state σ_n as input and returns true if it is subsumed by an old symbolic state σ_o in a set H. The program counter part of the state is stored explicitly. Thus, the first step of the algorithm is to obtain the set of old states $H' \subseteq H$ with program counter values that match those of state σ_n. This is implemented using a hash table, as is done in most explicit state model checkers. The number of entries in this table is limited by the partial-order reduction. We therefore do not expect a blowup in this data structure.

The set H' corresponds to a *disjunctive partitioning* of the set of states. Disjunctive partitionings are commonly used in symbolic model checkers for asynchronous concurrent programs, e.g., in [13, 26].

The second step is to check whether a symbolic state in $\sigma_o \in H'$ subsumes the symbolic state σ_n, i.e., if all explicit states represented by σ_n are also contained in σ_o. Note that we will not detect the case that σ_n is not covered by any single $\sigma_o \in H'$, but rather by a combination of states in H'. Comparing the new state with

the union of the symbolic states in H' would be too expensive. This may delay the detection of the fix-point, but will neither affect soundness nor termination.

A state σ_n is subsumed by a state σ_o if for all explicit states represented by σ_n there exists an identical state represented by σ_o. As the program counter components already match, we only need to compare the values of the state variables. As given by Equation 2, the set of explicit states represented by a symbolic states is defined using an existential quantification over the choice variables ι. Formally, for each choice of inputs ι_n for the new state σ_n, there must exist a (possibly different) choice of inputs ι_o for the old state σ_o that results in the same state:

$$\forall \iota_n | \iota_n(\gamma_n). \quad \exists \iota_o | \iota_o(\gamma_o). \quad \iota_n(\omega_n) = \iota_o(\omega_o) \tag{3}$$

Equation 3 can be transformed into a Quantified Boolean Formula (QBF) and passed to a QBF solver such as Quantor [10] or Quaffle [11]. We have found that modern QBF solvers, and especially Quantor, can handle surprisingly large instances that we generate. If the QBF solver determines the formula to be true, we can discard the state σ_n. Otherwise, we insert σ_n into H, and proceed with the state space exploration using the successors of state σ_n.

Optimization In Equation 3, the outermost quantification is done over the non-deterministic choice variables used as parameter for the states represented by σ_n. Given a deep symbolic simulation, this may be a large number of variables.

Note that we only care about the values of the state variables in a state represented by σ_n. Thus, we can re-write Equation 3 such that the outer quantification is done over the state bits, and not over the non-deterministic choices.

$$\forall x_n. \quad \exists \iota_n, \iota_o . x_n = \iota_n(\omega_n) \wedge (\iota_n(\gamma_n) \Longrightarrow (\iota_o(\gamma_o) \wedge \iota_n(\omega_n) = \iota_o(\omega_o)) \tag{4}$$

The number of state-bits maybe much smaller than the number of non-deterministic choices, and thus, the complexity of the formula is reduced.

Another simple optimization is to restrict the set of variables we consider to $V' \subseteq V$. V' is the set of variables that are *active* in any of the program locations $L' \subseteq L$ given by any of the program counters.

A variable is active in a program location if its value is of relevance to any instruction reachable from the location. E.g., local variables that are not yet in scope can be disregarded when comparing the values of the state variables.

A third optimization is to partition the set of variables into groups $C_1, \ldots, C_k$ that share choice variables. Indirect sharing, through other variables, has to be considered.

5 Experimental Results

We have implemented the algorithm described above in a tool called BoPPo. We use Limmat as the SAT solver, and Quantor as the QBF solver.

In this section, we compare BoPPo with other model checkers. We use the explicit state model checkers SPIN [24] and Zing [27]. We also compare our BoPPo with Moped [3] and Bebop [2], which are BDD-based symbolic model checkers. Neither Bebop nor Moped supports multiple threads, however. The experimental results are summarized in Table 3.

Benchmark	MOPED	SPIN	BEBOP	ZING	BOPPO
1	**0.1s**	*	0.1s	n/a	0.6s
2	*	3.8s	120s	n/a	**27.0s**
3	n/a	n/a	**0.17s**	n/a	0.43s
4	n/a	*	2058s	n/a	**75.6s**
5	n/a	*	n/a	*	**55.8s**

Table 3. Experimental results: n/a denotes that the model checker does not handle the benchmark due to lacking features, * denotes that the time limit (1 hour) or memory limit (2 GB) was exceeded

Benchmarks 1-4 are sequential; the first two benchmarks are artificial and contain about 30 Boolean variables. In the first benchmark, most states are reachable. The symbolic model checkers BEBOP, BOPPO, and MOPED handle this benchmark easily, while the explicit state model checkers run out of memory even with such a small number of state bits. The second benchmark encodes a multiplication over the Boolean variables. SPIN handles this benchmark easily, while MOPED exceeds the 2GB memory limit.

The benchmarks 3-5 are generated by SLAM. The SLAM model checker implements counterexample guided abstraction refinement for C programs. Benchmark 3 is a summary of 572 individual, small sequential benchmarks; the times given for the benchmark denote the average runtime. On the small benchmarks, BEBOP outperforms BOPPO. Benchmark 4 is a large sequential device driver.

An experimental version of SLAM provides support for the verification of concurrent programs[1]. In this mode, ZING is used as a replacement for SLAM's sequential reachability engine, BEBOP. Benchmark 5 is generated from a 4500 LOC Windows device driver with three threads in this manner.

As ZING is an explicit state model checker, it is not well-adapted to handle the larger Boolean programs that are produced by predicate abstraction. As discussed in Section 2, SLAM generates abstractions that make frequent use of non-deterministic choice. When SLAM is used to verify the correctness of Windows device drivers, we must also provide abstract representations of the kernel, other device drivers, and user-level applications. This environment adds a large amount of additional non-determinism. For this reason, SLAM in combination with ZING can process only relatively small model checking examples.

With BOPPO, SLAM is now able to solve much larger problems. ZING is unable to solve benchmark 6 after more than an hour of execution. BOPPO is able to solve the benchmark within a minute. We attempted to run this same benchmark using SPIN and NUSMV without any positive result.

Surprisingly, BOPPO appears to make a contribution for sequential programs as well. As we try to apply SLAM to more difficult properties and larger programs, BEBOP is sometimes the performance bottleneck. This problem is exacerbated by experiments where we have used a theorem prover that is accurate with respect

[1] Thanks to Georg Weissenbacher and Jakob Lichtenberg

to pointer arithmetic, bit-vectors, structures and unions [28] — this causes many additional Boolean variables to be added to the abstraction and also causes the logic used in the transition relation of the Boolean program to become more complicated. This puts additional strain on Bebop.

In the worst case, the predicates can begin to resemble the arithmetic from the original C program. BoPPo, because its symbolic representation is based on SAT and QBF and not BDDs, is better able to scale to larger and more complicated sequential Boolean programs.

6 Conclusion and Future Work

Symbolic model checking and partial-order reduction are hard to combine. For this reason model checkers for software systems typically treat non-trivial amounts of symbolic data, or non-trivial numbers of threads, but not both. We have presented a SAT-based model checking approach that can be used to efficiently reason about the safety of Boolean programs with both symbolic data and multiple threads. This allows model checkers which abstract software into Boolean programs to verify multi-threaded programs.

The algorithm presented in this paper implements partial-order reduction using SAT. The reduction is based on a change to the case-splitting algorithm used within the SAT-solver. This implementation strategy turns out to be better than an approach in which constraints on the interleavings are encoded as part of the input to the SAT-solver.

As future work, we want to experiment with other techniques for checking state subsumption for parametric representations. In [29], the authors use a SAT solver to compute a new parametric representation from a set of constraints. The new parametric representation is canonical for a given variable ordering, and thus allows an efficient fix-point detection. We would also like to try our techniques for checking liveness properties and for checking equivalence of two programs.

Note to reviewers: We intend to make BoPPo publically available. A preliminary release with examples (for reviewers) can be found at

`http://www.inf.ethz.ch/personal/daniekro/boppo/`

References

1. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN 00: SPIN Workshop. LNCS 1885, Springer-Verlag (2000) 113–130
3. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV. LNCS 2102, Springer-Verlag (2001) 324–336
4. Ball, T., Chaki, S., Rajamani, S.K.: Parameterized verification of multithreaded software libraries. In: TACAS, Springer-Verlag (2001)
5. Jain, H., Clarke, E., Kroening, D.: Verification of SpecC and Verilog using predicate abstraction. In: Proceedings of MEMOCODE 2004, IEEE (2004) 7–16
6. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: IFM. (2004)
7. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 02: Symposium on Principles of Programming Languages, ACM Press (2002) 58–70

8. Coudert, O., Madre, J.: A unified framework for the formal verification of sequential circuits. In: ICCAD, IEEE (1990) 78–82
9. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Formal verification using parametric representations of boolean constraints. In: DAC, ACM Press (1999) 402–407
10. Biere, A.: Resolve and expand. In: Proc. SAT'04. LNCS, Springer (2004)
11. Zhang, L., Malik, S.: Conflict driven learning in a quantified boolean satisfiability solver. In: ICCAD. (2002)
12. Leino, K.R.M.: A SAT characterization of Boolean-program correctness. In: SPIN. (2003)
13. A. Cimatti et al.: NuSMV 2: An opensource tool for symbolic model checking. In: CAV. (2002) 359–364
14. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 05: Symposium on Principles of Programming Languages, ACM Press (2005)
15. Holzmann, G., Peled, D.: An improvement in formal verification. In: Proc. Formal Description Techniques, FORTE94, Chapman & Hall (1994) 197–211
16. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state-space exploration. FMSD **18** (2001) 97–116
17. Jussila, T., Niemelä, I.: Parallel program verification using BMC. In: ECAI 2002 Workshop on Model Checking and Artificial Intelligence. (2002) 59–66
18. Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. In: Software Model Checking (SoftMC). ENTCS (2003)
19. Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., Tasiran, S.: MOCHA: Modularity in model checking. In: CAV. LNCS. Springer (1998) 521–525
20. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread modular abstraction refinement. In: CAV, Springer (2003) 262–274
21. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS 05: Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2005)
22. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: CAV. Volume 1254 of LNCS., Springer (1997) 72–83
23. Colón, M., Uribe, T.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV. Volume 1427 of LNCS., Springer (1998) 293–304
24. Holzmann, G.: The model checker SPIN. IEEE Trans. on Software Engineering **23** (1997) 279–295
25. Peled, D.: All from one, one for all: on model checking using representatives. In: In Proc.of CAV. (1993)
26. Barner, S., Rabinovitz, I.: Effcient symbolic model checking of software using partial disjunctive partitioning. In: CHARME. (2003) 35–50
27. T. Andrews et al.: Zing: Exploiting program structure for model checking concurrent software. In: CONCUR 2004. (2004)
28. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In: Proceedings of CAV 2005. Lecture Notes in Computer Science, Springer Verlag (2005) To appear.
29. Chauhan, P., Clarke, E., Kroening, D.: A SAT-based algorithm for reparameterization in symbolic simulation. In: DAC 2004, ACM Press (2004) 524–529