IJIT: An API for Boolean Program Analysis with Just-in-Time Translation

Peizun Liu(✉) and Thomas Wahl

Northeastern University, Boston, USA
lpzun@ccs.neu.edu

Abstract. Exploration algorithms for explicit-state transition systems are a core back-end technology in program verification. They can be applied to *programs* by generating the transition system on the fly, avoiding an expensive up-front translation. An on-the-fly strategy requires significant modifications to the implementation, into a form that stores states directly as valuations of program variables. Performed manually on a per-algorithm basis, such modifications are laborious and error-prone.

In this paper we present the IJIT Application Programming Interface (API), which allows users to automatically transform a given transition system exploration algorithm to one that operates on *Boolean* programs. The API converts system states temporarily to program states *just in time* for expansion via image computations, forward or backward. Using our API, we have effortlessly extended various non-trivial (e.g. infinite-state) model checking algorithms to operate on multi-threaded Boolean programs. We demonstrate the ease of use of the API, and present a case study on the impact of the just-in-time translation on these algorithms.

1 Introduction

Boolean programs [4], a finite-data abstraction of general-purpose software obtained by predicate abstraction [13], have proved to be an intermediate notation very useful for verification that factors out the data complexity from programs. State exploration algorithms, however, are typically designed to operate on forms of transition systems. To apply these algorithms to Boolean programs, one can in principle translate the input program into a transition system, before starting the exploration. This input translation incurs, however, a blow-up that is exponential in the number of program variables.

This classic problem in program verification has led to sophisticated algorithms that translate the program into a transition system *on the fly*, as the state space is explored. This idea was pioneered for model checking algorithms by the SPIN tool [14]. In general, to convert an exploration algorithm into an on-the-fly version, the state representation data structure needs to be changed everywhere in the implementation to a tuple over program variable valuations. Consequently, operations on the state representation, notably image computations, need to be re-implemented as well, to reflect the program semantics.

This work is supported by NSF grant no. 1253331.

A. Cimatti and M. Sirjani (Eds.): SEFM 2017, LNCS 10469, pp. 316–331, 2017.
DOI: 10.1007/978-3-319-66197-1_20

Such an algorithm re-implementation avoids the exponential program-to-transition-system translation, but comes with its own cost: due to its low-level nature, it is laborious and error-prone, especially for sophisticated algorithms. In the rest of this paper we describe a way to *automatically* construct on-the-fly program state explorers from implementations operating on transition systems. We leave the system state data structure intact (hence no algorithm re-implementation), and pass the Boolean program as input (hence no input program translation). Our strategy is then as follows: whenever predecessor or successor images need to be computed, the current system state is converted temporarily and *just in time* for the image computation into a Boolean program state. The image is then computed using the program execution semantics, e.g. via pre- or post-conditions. The resulting image states are converted back to, and stored as, system states. This process is repeated for each image computation.

This simple strategy has one crucial advantage: it requires very little change on a per-algorithm basis: once we have provided image operations for Boolean programs (a one-time effort), all we need to do is replace the calls to image functions in the original implementation by new functions that take a system state and (i) convert it to a Boolean program state, (ii) apply the image, and (iii) convert the result back. These steps can be encapsulated into a single operation.

Being largely independent of the underlying algorithm, this strategy can be automated. To this end, we present an Application Programming Interface (API) that provides conversion functions between system and Boolean program states. It further offers implementations of common image operations on Boolean programs, including standard pre- and post-images, as well as more complex image operations for infinite-state system exploration. Our API permits users to transform a wide range of transition system exploration algorithms into Boolean program versions automatically—with little effort and a high degree of reliability—, including sophisticated reachability and coverability algorithms for infinite-state systems such as Petri nets.

For an experimental case study, we have implemented several exploration algorithms in three versions: (a) one that uses the naive **input translate** option, (b) one that implements the manual **algorithm re-implement** option, and (c) one that uses our API to perform **just-in-time translation**. The comparison (c) against (b) demonstrates that the repeated state representation conversion is not harmful: using our API we achieve almost the same efficiency as the gold standard of re-implementation by hand. The comparison (c) against (a) demonstrates that the just-in-time version is vastly more efficient than the version employing up-front input translation.

2 Boolean Programs and Thread-Transition Systems

Our API allows exploration algorithms that operate on transition systems-derived from *Boolean programs* (BP) [4] to be applied directly to such programs,

circumventing the blow-up incurred by the input translation. In this section we formalize the language of (possibly threaded) BPs and the transition system model of *thread transition systems*. The latter serve as the input language of exploration algorithms that we later wish to apply directly to BPs.

2.1 Boolean Programs

Boolean programs typically arise from predicate abstractions of application code in system-level languages. All variables are of type `bool`. Control flow constructs are optimized for synthesizability and therefore include "spaghetti statements" like **skip** and **goto**. An overview of the syntax of BPs is given in Fig. 1. A program consists of a **decl**aration of *global* Boolean variables, followed by a list of functions. A function consists of a **decl**aration of *local* Boolean variables, followed by a list of labeled statements.

We illustrate the intuition behind individual statements of BPs. Among the sequential statements (*seqstmt*), **skip** advances the program counter (pc); **goto** *labellist* nondeterministically chooses one of the given labels as the next pc; **assume** terminates executions that do not satisfy the given expression. Statement **:=** assigns, in parallel, each value in the given *exprlist* to the respective variable in the same-length *varlist*, but terminates the execution if the result does not satisfy the **constrain** expression, if any. Statement **assert** indicates assertions for verification and otherwise acts like **skip**. The meaning of function calls (possibly recursive) and return statements is standard and omitted. In all cases, *expr* is a Boolean expression over global and local program variables, the constants 0 and 1, and the choice symbol $\star$; the latter nondeterministically evaluates to 0 or 1.

In the presence of multiple threads, the global variables are *shared* (both read and write) between the threads. The executing thread is called *active*, the others *passive*. All sequential statements have asynchronous semantics, i.e. they change the local variables of only the active thread. The other statements in Fig. 1 intuitively behave as follows:

prog ::= **decl** *varlist*; *func**

stmt ::= *seqstmt*
| **start_thread** *label*
| **end_thread**
| **atomic** { [*stmt*;]* }
| **wait**
| **signal**
| **broadcast**

func ::= **void** *name* (*varlist*) **begin**
decl *varlist*;
[*label*: *stmt*;]*
end

seqstmt ::= **skip**
| **goto** *labellist*
| **assume** (*expr*)
| *varlist* **:=** *exprlist* [**constrain** *expr*]
| **assert** (*expr*)

Fig. 1. Boolean program syntax (partial)

start_thread ***label*** (i) advances the program counter of the executing thread, and (ii) creates a new thread whose local variables are copied from the executing thread and whose pc is given by *label*;
end_thread terminates the executing thread;
atomic $\{\boldsymbol{stmt}^*\}$ denotes atomic execution: a thread executing inside an atomic section cannot be preempted;
wait blocks the execution of a thread (see next);
signal advances the pc of the executing thread and nondeterministically wakes up *one* thread blocked at a **wait** statement, if any, i.e. it advances its pc;
broadcast advances the pc of the executing thread and wakes up *all* threads currently blocked at a **wait**.

Wait and release via **signal** or **broadcast** are powerful synchronization mechanisms, allowing many threads to change state at the same time. None of the above six statements change global variables; only **start_thread** and **end_thread** change the number of threads. Fig. 2 (left) shows an example of a BP with an assertion. A precise small-step operational semantics for multi-threaded BPs is given in App. A of [20].

2.2 From Boolean Programs to Thread Transition Systems

Transition systems are the input formalism for many exploration algorithms, such as breadth-first search for reachability analysis, or the Karp-Miller algorithm for deciding *coverability* in infinite-state systems [16]. To apply these to BPs (and thus connect them, via predicate abstraction, to software verification), the programs are typically translated into transition systems.

Let Boolean program $\mathcal{B}$ be defined over sets of global and local variables V_G and V_L, respectively, and let $\{1..pc_{\max}\}$ be the set of program locations.[1] We translate $\mathcal{B}$ into a finite-state *thread transition system* (TTS) $M = (S, R)$, over the state space $S = \{0,1\}^{|V_G|} \times \{1..pc_{\max}\} \times \{0,1\}^{|V_L|}$ and *edges* R.

Individual BP statements are translated into edges, as follows. A given state $s \in S$ determines a (single-threaded) program state $s_{\mathcal{B}}$ of $\mathcal{B}$ in a straightforward way: s encodes a valuation of all global variables (the $\{0,1\}^{|V_G|}$ part, the *global state*), a program counter, and a valuation of all local variables (the $\{0,1\}^{|V_L|}$ part, the *local state*). Executing $\mathcal{B}$ on $s_{\mathcal{B}}$ has several effects: first, it generally changes both the global variables, and the local variables of the active thread (including the pc). These changes result in a new state $t \in S$ again in a straightforward way, defining an edge $(s,t) \in R$.

Second, thread creation and termination, as well as signals and broadcasts, typically have "side effects" that alter the thread count, or local variables of passive threads. To capture such effects in the (single-thread) data structure M, each edge comes with a *type*. It is then left to the exploration algorithm, which has access to the current system state, to fully implement transition semantics. As an example, Fig. 2 shows a BP and a translation into a TTS. Symbol $\rightarrowtail$

[1] We write $\{l..r\}$ compactly for $\{n \in \mathbb{N} : l \leq n \leq r\}$.

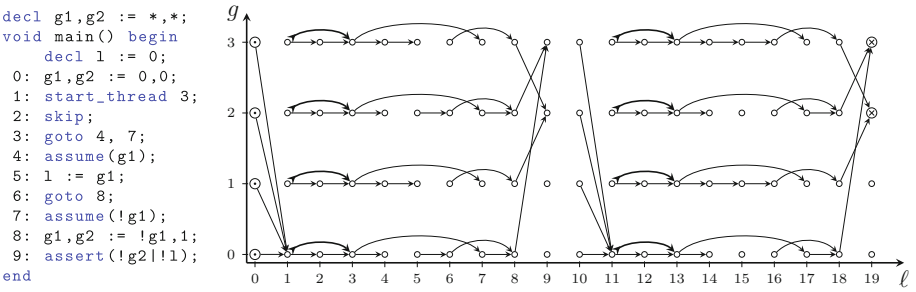

Fig. 2. A Boolean program (left) and a possible translation into a TTS (right). Global variable valuation $(\mathtt{g1}, \mathtt{g2})$ is encoded as state $g = 2 \times \mathtt{g2} + \mathtt{g1} \in \{0..3\}$. Similarly, local variable valuation $(pc, \mathtt{l})$ is encoded as state $\ell = 10 \times \mathtt{l} + pc \in \{0..19\}$. With this encoding, the four initial program states are shown as $\odot$, the two assertion failure states (satisfying $pc = 9 \wedge \mathtt{g2} = \mathtt{l} = 1$) as $\otimes$.

marks edge $(0,1) \rightarrowtail (0,3)$ as a thread creation edge. The semantics of thread creation (App. A of [20]) prescribes that the active (creat*ing*) thread moves on (to $pc = 2$); this is reflected by an ordinary edge $(0,1) \rightarrow (0,2)$ in the TTS. The creat*ed* thread needs a start location, which is the pc value of the BP state $(\mathtt{g1}, \mathtt{g2}, pc, \mathtt{l}) = (0,0,3,0)$ encoded by the target TTS state $(0,3)$ of the edge. Other than above two types of edges shown in Fig. 2, there is one more type, denoted by $\rightsquigarrow$, used in the TTS to characterize broadcasts.

The problem with such a translation from $\mathcal{B}$ to M is of course the potential blow-up: the nominal state space S of M is exponential in the number of global and local variables. This problem has long been known and has led to sophisticated *on-the-fly* temporal-logic model checkers such as Spin [14], but also to ad-hoc re-implementations of specific exploration algorithms [7,19]. In the rest of this paper we describe an API that automates the construction of on-the-fly program state explorers.

3 BP Analysis with JIT Translation: Overview

We target *exploration algorithms*, i.e. algorithms that operate on a transition system representation of the given program and involve *image computations*: given a system state, they repeatedly compute some notion of successors or predecessors of the state. Figure 3 (left; ignore the boxes for now) shows a schematic version of such algorithms. Input is a transition system M and some target state set T, such as a bad system state whose discovery would indicate a reachable error in the system. The algorithm maintains a worklist W of states to be explored, typically initialized to the initial or bad states of the system, depending on whether the search proceeds forward or backward. It also maintains a set X of explored states, initially empty. The exploration proceeds by extracting an unexplored state w from W and iterating through the set of states w' in w's image, computed by image. If w' is new, we test whether it belongs to the target states T.

Scheme 1 EXPLORE(M, T)

Input: transition system M, target T

1: Initialize W and X
2: **while** $\exists w \in W$
3: $\quad W := W \setminus \{w\}$
4: $\quad$ **for** $w' \in \mathsf{image}(w)$
5: $\quad\quad$ **if** w' not in X **then**
6: $\quad\quad\quad$ **if** w' in T **then**
7: $\quad\quad\quad\quad$ **return** "found"
8: $\quad\quad\quad$ merge w' into W and X
9: **return** "not found"

Scheme 2 EXPLORE_IJIT$(\mathcal{B}, T)$

Input: Boolean Program $\mathcal{B}$, target T

1: Initialize W and X
2: **while** $\exists w \in W$
3: $\quad W := W \setminus \{w\}$
4: $\quad$ **for** $w' \in f^{-1}(\mathsf{image}_{\mathcal{B}}(f(w)))$
5: $\quad\quad$ **if** w' not in X **then**
6: $\quad\quad\quad$ **if** w' in T **then**
7: $\quad\quad\quad\quad$ **return** "found"
8: $\quad\quad\quad$ merge w' into W and X
9: **return** "not found"

Fig. 3. State exploration over a transition system (left) and a Boolean program (right). Lines 5 and 6 test whether w' has not been explored and w' is a target state, respectively. In a concrete algorithm these tests may involve more than set membership.

If so, we report the success of the search. The search terminates when no more unexplored states exist (in W).

Now suppose the transition system M is actually a translation of a Boolean program $\mathcal{B}$, which we want to explore directly, using the same algorithm scheme. One way to achieve that is to change the data structure that Scheme 1 relies on: instead of storing states to be explored as states of M, we store them as Boolean program states, one entry per program variable. Images are then computed by "executing" $\mathcal{B}$ in accordance with $\mathcal{B}$'s execution model.

However, like with any data structure change in any non-trivial program, the required effort is significant: all of T, W, X must be changed, and therefore virtually every line in a program that implements Scheme 1. Re-implementing image to operate on a Boolean program $\mathcal{B}$ is also involved. The whole change process is not only error-prone; it also creates an entirely new implementation that needs to be maintained independently of the one operating on M.

An alternative to this strategy is shown in Scheme 2 on the right, which is almost identical to that on the left. States are stored as transition system states of M as before, but the input is now the Boolean program $\mathcal{B}$. Since M is no longer available, we cannot apply M's transition relation to compute images. However, since there is a one-to-one correspondence between states of $\mathcal{B}$ and of M, we can compute images by converting, using function f, to $\mathcal{B}$'s state representation *just in time* for the image computation, and reverting the resulting image states back to the system state format of M (Line 4). Note that f^{-1} needs to operate on (and return) *sets* of states.

Operation $\mathsf{image}_{\mathcal{B}}$ computes images of an intermediate program state $p := f(w)$. Its implementation depends on the kind of image computation performed by the algorithm: For standard forward exploration, it can be computed by executing, from p, the statement of $\mathcal{B}$ pointed to by the pc encoded in p. For a backward exploration algorithm, $\mathsf{image}_{\mathcal{B}}$ is more complicated: we need to identify statements

leading to the current pc via $\mathcal{B}$'s control flow graph, and then symbolically execute such statements backwards, e.g. via weakest preconditions [19].

The API presented in this paper supplies an implementation of the $\mathcal{B} \leftrightarrow M$ conversion functions (f, f^{-1}) and of various common image operations applied to (multi-threaded) Boolean program states, including backward statement execution for backward search algorithms. In many cases, all the user needs to do is to replace the image operation in their algorithm, as shown in Fig. 3 (boxes).

A minor runtime cost of using an algorithm according to Scheme 2 is that the repeated conversion will take some time. This time is linear in the number of Boolean program variables (and the number of threads of the current system state, if multi-threaded). The state conversion in either direction is a simple operation that can be highly optimized. We will demonstrate in Sect. 5 that the benefit of avoiding the explicit construction of M often far outweighs the conversion overhead.

We end this section by discussing desirable characteristics of algorithms that will benefit from using our API. We target exploration (search, model checking) algorithms for state transition systems (e.g. TTS) of Boolean programs. The term "exploration" here refers to the reliance of such algorithms on the computation of *standard* pre- and postimages of (sets of) states. The transition systems must relate to the Boolean programs in a way that there is a one-to-one correspondence between program states and system states. In particular, the systems cannot be (lossy) *abstractions* of the Boolean programs; otherwise, a system state may not map to a unique program state, or vice versa.

4 The IJIT Application Programming Interface

In this section we sketch usage and design of our API, named IJIT: **I**nterface for **J**ust-**I**n-**T**ime translation. A detailed tutorial and documentation can be found in [18].

4.1 API Usage

We use a fictitious procedure `explore` to illustrate the use of our API; see Fig. 4 (left). The procedure explores the state space of some transition system given as a TTS. It begins by reading the TTS into a data structure called `R` (Line 5) and extracts from `R` sets of initial and final states, respectively (Lines 7 and 8). The procedure then enters some kind of loop to explore the state space represented by `R`, perhaps until no more unexplored states are available (this is immaterial for our API). Crucial is that the loop body will invoke an image operation on a state `tau` (Line 12), likely at least once in each iteration. We assume `R` is nondeterministic, so that the call returns a set of states, `Tau`.

Figure 4 (right) highlights (in gray) the changes the programmer needs to make to have procedure `explore` operate on a Boolean program; we call the resulting procedure `explore_jit`. We explain these changes in the following.

```
// user's headers, namespace, etc.

void explore() {
  // ...
  auto R = read_file("filename.tts")1

  set<state> I = R.init();
  set<state> F = R.final();
  state tau;
  while (...) { // state exploration
    tau = ... ; // an unexplored state
    set<state> Tau = image(tau);

    ...
  } // end while
}
```

```
1  #include "ijit.hh"
2  using namespace ijit;
3  void explore_jit() {
4    // ...
5    auto P = parser::parse("filename.bp", mode::POST);
6    converter c;
7    set<state> I = c.convert(P.init());
8    set<state> F = c.convert(P.final());
9    state tau;
10   while (...) {
11     tau = ... ;
12     set<state> Tau = c.convert(
13                  image(c.convert(tau), mode::POST));
14     ...
15   }
16 }
```

Fig. 4. An example illustrating the usage of Ijit. Left: a fictitious state space exploration procedure. Right: the just-in-time version obtained using Ijit. Line numbers in the middle; highlighted code shows places that have changed from the original version.

- Instead of reading a TTS, we now read a Boolean program as input (Line 5). This is done using a parser supplied by Ijit. Procedure `parse` has two arguments: the name of input file, and the parser's direction mode: `POST` will cause the parser to generate code for subsequent forward-directed analysis (via postimages). Mode `PREV` does the analogous for backward analysis; a mode of `BOTH` will generate code for both. The parser also offers functionality to return sets `I` and `F` of initial and final program states, extracted from the initial variable declarations and assertions in the BP, respectively.
- The conversion between different state representation formats, explained below, is done via methods of a class `converter`. The user needs to instantiate this class before any conversion methods of the API can be called (Line 6).
- Conversion between state representation formats happens in several places: to convert the initial and final Boolean program state sets into TTS state sets (Lines 7 and 8), and in the image computations. If the algorithm implemented by procedure `explore` operates on TTS as defined in Sect. 2, the JIT version of the procedure can be implemented using conversion functions supplied by the API (Line 12): the current (unexplored) TTS state `tau` is converted into a BP state, followed by a Boolean program image computation using the given direction mode, followed by a back-conversion into a set of TTS states.
  If the API's conversion functions cannot be used, users must supply their own functions. To reduce the programming burden, the API provides an inheritance interface that allows defining conversion functions via specialization. Users are free to define stand-alone conversions.

4.2 API Design

API Ijit is implemented in C++. A schematic overview is shown in Fig. 5.

Parser. The main purpose of the parser is to process the input BP and populate the data structures to be used in image computations. These include the program's control flow graph, and pre- and postcondition expressions for pre- and

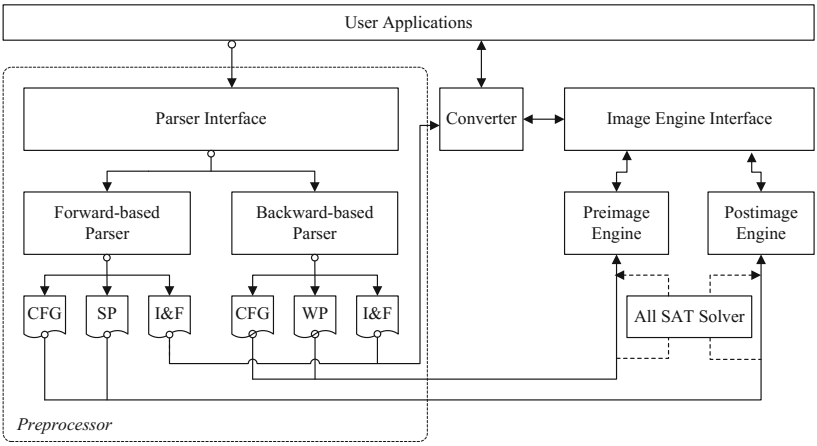

Fig. 5. Schematic overview of IJIT. The preprocessor part is usually called only once. CFG: control flow graph; SP/WP: strongest postcondition/weakest precondition; I/F: the set of initial/final states

postimage computations, respectively. The parser also extracts initial and final state information, the latter by collecting all states violating any of assertions in the Boolean program.

Converter. The converter provides an adapter between system states and program states. In our design, the converter is an abstract C++ class with default implementations of conversion functions. If desired or necessary, users can either inherit the abstract class and override the default implementation, or write a stand-alone converter from scratch.

Image Engine. At the core of our API are the engines to compute the preimage or postimage of a given Boolean program state. These routines make use of the control flow graph obtained by the parser, especially for preimages, in order to determine the set of statements that can lead to the current *pc*. Once the statement to be executed forward or backward has been determined, the statement's semantics determines the effect on the program data. The semantics is given as a set of first-order predicates expressing strongest post- or weakest preconditions. To perform image computations, the engine instantiates these formulas with the current-state valuations of the program variables. It then invokes an All-SAT solver to obtain the pre- or postimages as satisfiable assignments.

All-SAT Solver. The All-SAT solver used in image computations is not based upon a state-of-the-art SAT solver, which would require CNF conversion. Instead we found it to be more efficient to simply build a custom SAT solver that enumerates solutions. Note that input formulas to the solver formalize Boolean program statements and thus tend to be very short.

5 Case Study: Performance Benefits of IJIT

We evaluate the benefit of our API on a number of diverse benchmark algorithms. All are designed to operate on thread-transition systems (TTS) for either a fixed or an unbounded number of threads; we wish to apply them to multi-threaded Boolean programs directly. For each algorithm, we compare the performance of three versions: (i) the *TTS version*, which is the original version, but prefixed by an input translation from BPs into TTS; (ii) the *BP version*, which is a manual and *optimized* re-implementation where the internal state data structure has been changed to BP states; and finally (iii) the *JIT version*, which employs our API. We expect a performance ranking of the form

$$BPversion \;<\; JITversion \;\ll\; TTSversion$$

where "$<$" ("$\ll$") means "(much) faster". In particular, the hand-crafted BP version makes repeated conversion between state representations unnecessary and can therefore be considered the gold standard for efficiency. We hope the automated JIT version of the algorithm to perform nearly as well.

5.1 Benchmark Algorithms

We sketch the purpose and basic concepts of four diverse algorithms used in our case study; more details are provided in App. B of [20]. The algorithms cover the spectrum of finite- and infinite-state searches, and of forward and backward explorations.

Cutoff Detection via Finite-State Search (Ecut) [15]. Ecut implements *dynamic cutoff detection* for parameterized thread transition systems. A *cutoff* point is a number n_0 of threads that are sufficient to reach all reachable thread states. The core procedure of Ecut is a (multi-threaded but) finite-state search, BFS style. The TTS version of Ecut can be transformed into the JIT version without any programming beyond the few changes discussed in Sect. 4.

Karp-Miller Procedure [16]. We experiment with two variants of this classic procedure; both are in use in unbounded-thread program verification:

(1) Km decides the reachability of a specific target state t: it stops when a state covering t has been encountered;
(2) Akm ("All-Km") builds the complete coverability tree, i.e. it runs Km until a fixpoint is reached.

WQOS Backward Search (BWS) [1,2]. This technique is a sound and complete algorithm to decide coverability for *well quasi-ordered systems* (WQOS), a broad family of transition systems that subsumes replicated Boolean programs, Petri nets, VASS, and many more. Note that BWS is a backward exploration. In contrast, the previous three algorithms explore forward.

5.2 Case Study

Experimental Setup. We compare the impact of our API on the efficiency of the four algorithms described in Sect. 5.1. For each algorithm $A \in \{\text{ECUT}, \text{KM}, \text{AKM}, \text{BWS}\}$, we compare three different versions: (1) the TTS version — named A(TTS); (2) the JIT version obtained using our API — named A(JIT); and (3) the hand-implemented Boolean program version — named version A (BP).

We perform the comparison using a collection of Boolean programs obtained via predicate abstraction from 30 concurrent `C` programs. The `C` programs are detailed in Table 1. We use SATABS [8] to construct the BPs from these programs. The BPs are also concurrent; threads execute the same Boolean procedure. In most cases, the same `C` source program generates several BPs (since SATABS goes through several abstract-verify-refine iterations). In the end we obtained 155 BPs for the 30 `C` programs. For the TTS version of each algorithm, we use SATABS to generate the TTS from the Boolean program (option `--build-tts`; this is where the input format explosion inevitably happens).

For each benchmark, we consider verification of a safety property, specified via an assertion that is pushed, during predicate abstraction, from `C` to the Boolean program. All experiments are performed on a 2.3 GHz Intel Xeon machine with 64 GB memory, running 64-bit Linux. The timeout is set to 30 min; the memory limit to 4 GB. All benchmarks and implementations are available at [18].

Table 1. Benchmark statistics: $GV/LV/LOC$ = # of global/local `C` program variables/lines of code; $|V_G|/|V_L|/|PC|/Its.$ = # of global/local Boolean variables/program counters/CEGAR iterations; $|G|/|L|/|R|$ = # of global/local states/transitions in TTS; *Safe?* = ✓: program safe; $|\cdot|$ represents the median of the feature across different BP/TTS resulting from the same `C` program. Note that often $|G| > 2^{|V_G|}$, due to auxiliary states used by SATABS in the BP → TTS translation

ID/Program	C Program			BP				TTS			*Safe?*
	GV	*LV*	*LOC*	$\|V_G\|$	$\|V_L\|$	$\|PC\|$	*Its.*	$\|G\|$	$\|L\|$	$\|R\|$	
01/INCREM-L	2	1	46	3	1	40	2	33	71	688	✓
02/INCREM-C	1	3	57	0	4	35	4	5	449	784	✓
03/PRNG-L	2	4	63	2	3	45	2	17	265	1488	✓
04/PRNG-C	1	5	95	0	5	48	2	5	993	1760	✓
05/FINDMAX-L	3	3	59	1	0	43	2	9	25	57	✓
06/FINDMAX-C	2	5	79	0	1	48	2	5	59	76	✓
07/MAXOPT-L	3	4	69	1	1	48	2	9	63	162	✓
08/MAXOPT-C	2	6	86	0	2	53	2	5	137	196	✓
09/STACK-L	4	2	79	1	3	53	3	9	157	360	✓
10/STACK-C	3	3	89	3	1	54	2	33	81	740	✓
11/BS-LOOP	0	6	24	0	7	30	1	65	24	448	✗
12/COND	1	3	56	0	3	29	2	33	25	200	✓
13/FUNC-P	2	1	67	2	6	32	3	5	3969	9728	✓
14/S-LOOP	5	0	60	4	0	37	20	5	209	296	✓
15/PTHREAD	5	0	85	7	0	60	5	17	3329	20608	✗
16/TAS-LOCK	2	2	58	3	1	48	2	16385	54	269488	✓
17/DBLOCK-1	3	0	70	7	1	79	10	513	151	20928	✓
18/DBLOCK-2	3	0	73	6	1	47	22	33	71	688	✓
19/DBLOCK-3	3	0	66	4	1	73	3	257	67	4976	✓
20/TICKET-HC	3	1	61	5	1	73	5	257	139	10912	✓
21/TICKET-LO	3	1	46	5	1	63	5	65	115	2048	✓
22/BSD-AK	1	7	90	3	1	119	2	33	196	1922	✓
23/BSD-RA	2	21	87	3	0	138	2	33	107	996	✓
24/NETBSD	1	28	152	3	1	278	3	33	423	4096	✓
25/SOLARIS	1	56	122	5	1	182	2	129	283	10847	✓
26/BOOP	5	2	89	5	2	61	4	129	213	8064	✗
27/QRCU-2	7	6	120	3	0	129	15	33	103	1001	✓
28/QRCU-4	8	8	182	5	2	275	21	129	873	35024	✓
29/UNVER-IF	2	1	25	4	0	53	3	129	95	4096	✓
30/SPINLOCK	2	0	37	3	0	47	2	129	79	3584	✓

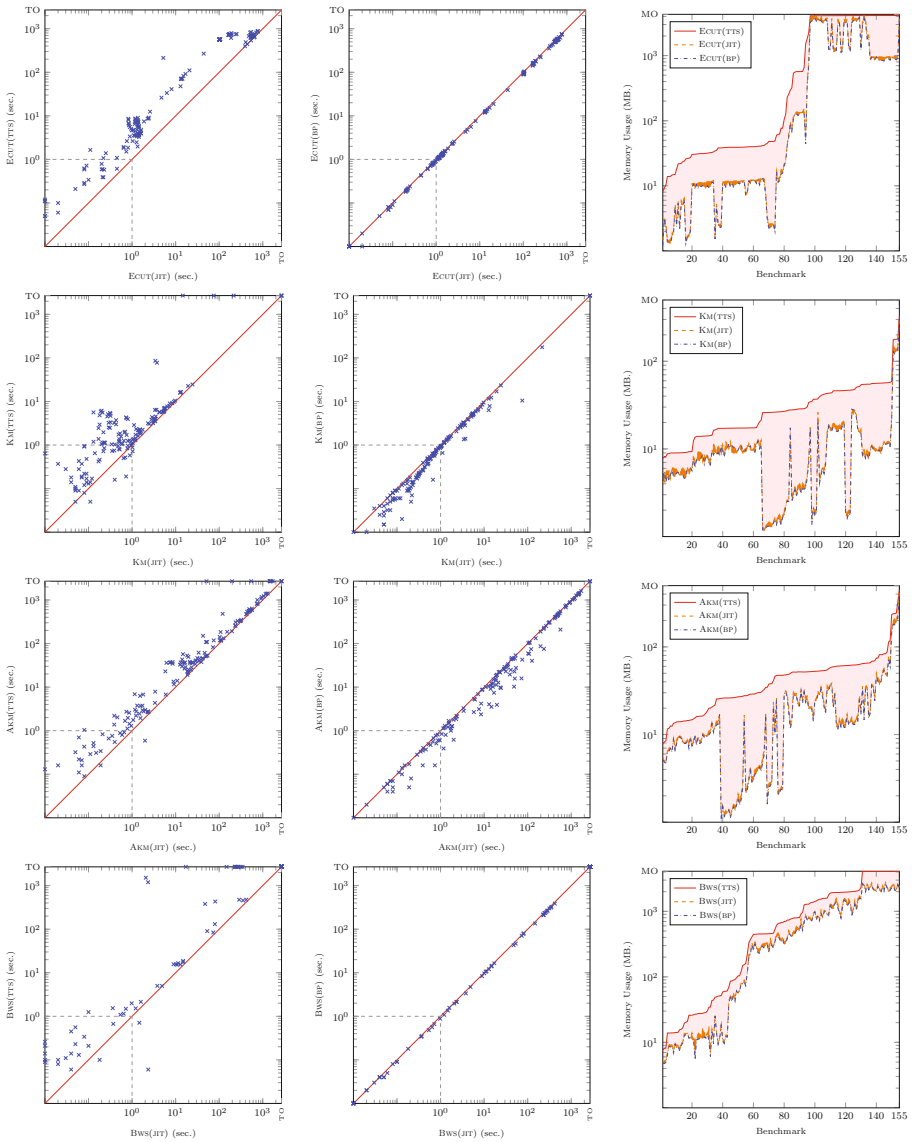

Fig. 6. Performance impact of our API (TO: timeout, MO: memory out). For $A \in \{\text{ECUT}, \text{KM}, \text{AKM}, \text{BWS}\}$: ▶ runtime comparison: left column: A(TTS) against A(JIT); center: A(BP) against A(JIT). Each dot = execution time on one example. Square in the lower left corner of each chart: runtime of less than 1 second for both algorithms, hence unreliable. ▶ memory usage comparison: right column: comparing memory usage across the three different versions. The plots are sorted by the memory usage of the TTS version of A. The shadowed areas show the difference. (Color figure online)

Results. The results of our case study are shown in Fig. 6. The first column shows, for the four algorithms, the runtime comparison of the JIT version (lower right in each chart) against the original TTS version of the algorithm (upper left). The log-scale charts clearly demonstrate the performance advantage — sometimes several orders of magnitude – of not pre-translating the input BP into a potentially large TTS. In many cases, runs that timed out in the TTS version can now be completed within the 30mins limit. We point out that, while the conversion time BP → TTS is included in the runtime for the TTS version, it is not even to blame for the weaker TTS version performance: the conversion usually takes a few seconds. What makes the TTS version slow is the relatively large input TTS to the TTS-based algorithm.

The second column shows the runtime comparison of the JIT version (lower right in each chart) against the hand-implemented BP version of the algorithm (upper left). Here the expectation is the opposite: we would like to get as close to the diagonal as possible. This is achieved in all four cases to a satisfactory degree. For the backward search algorithm, the comparison is more favorable for JIT than for the two KM-based algorithms, with a performance nearly indistinguishable from that of the BP version. This can be attributed to the fact that BWS overall takes more time than the forward search implemented in KM, since backward exploration faces more nondeterminism and in general visits a larger number of configurations. The relative overhead of state representation conversion is thus smaller.

The third column shows that the memory consumption of the JIT and BP versions of each algorithm are very similar, and both are vastly below that of the TTS version. This reflects in part the fact that the TTS version needs to store the (relatively large) generated TTS in memory. More relevant, however, is the fact that the TTS contains many redundant (since unreachable) transitions — their absence is the very advantage of on-the-fly exploration techniques. Such redundant transitions translate into a large number of redundant configurations explored by the TTS version of the algorithm.

6 Related Work

Promoted by the success of predicate-abstraction based tools such as SLAM [6] and SATABS [8], Boolean programs are widely used in verification. Accordingly, extensive research has been done on their analysis, leading to a series of efficient algorithms, e.g., recursive state machines [3], and the symbolic verifiers BEBOP [5], MOPED [11,12], BOPPO [9], and GETAFIX [17]. Most of the above approaches use BDDs as symbolic representation, which do not lend themselves to an efficient on-the-fly model construction.

In contrast, explicit-state model checking techniques often construct the state space of the program they are exploring on the fly. A prominent tool that pioneered this strategy is the explicit-state model checker SPIN [14]. Another notable explicit-state on-the-fly model checker is Java PathFinder [21], which takes Java™ bytecode and analyses all possible paths through the program, checking for deadlocks, assertion violations, etc.

Solutions addressing the translation blow-up in connection with (more complex) unbounded-thread verification techniques are rare. While these techniques have been applied to program analysis, the application is typically preceded by an up-front translation of the program into an explicit transition system [10,15]. For Boolean programs generated via predicate abstraction, this only works for small local state spaces, for example when the number of predicates is small. When going through several iterations of the predicate abstraction CEGAR loop, in contrast, the number of Boolean program variables quickly becomes large.

On-the-fly techniques for unbounded-thread algorithms applied to Boolean programs are given in tools by Basler et al. [7], and by Liu et al. [19]. Both are re-implementations of the algorithms they are targeting, which is the Karp-Miller procedure for VASS in the former case, and the backward search algorithm for broadcast Petri nets in the latter. Both demonstrate the benefits of exploring BPs directly, but they do not come for free: the re-implementation is low-level work involving tricky data structure changes, affecting the very foundation of the implementation. In fact, the Karp-Miller implementation in [7] generated runtime errors on some of our benchmarks, so we excluded it from our case study.

7 Summary

The problem of the blow-up between programs and transition systems that describe the programs' semantics and are often used in exploration algorithms is well known. Translating a program into an explicit transition system undermines the practical runtime performance of these algorithms, and thus diminishes their value. This problem has been addressed in an ad-hoc way, by re-implementing these algorithms into ones operating on programs. This process is painful and prone to programming errors, to which we attribute the fact the input translation cost is often grudgingly accepted.

In this paper we have introduced an API that largely automates the required transformations. In the best case, programmers mostly need to add calls to an API-provided `convert` method to (usually few) places in the code where images are computed. In the worst case, programmers have to supply this conversion method. We have demonstrated the huge impact of the use of the API on various algorithms that rely on an up-front BP $\rightarrow$ TTS translation. We have also compared the performance of the JIT version to the version re-implemented by hand that operates entirely on Boolean programs, and found nearly no performance difference to this gold-standard implementation.

We have presented our API with dedicated support for algorithms that operate on Boolean programs and thread-transition systems, due to their popularity in, and significance for, software verification. Given proper state representation conversion functions, we believe our API to be able to bridge the gap between other types of modeling languages, such as Boolean programs and Petri nets. We leave implementing, and experimenting with, such extensions for the future. Extending the API to support algorithms like partial order reduction

that need to perform a nonstandard image computation is another promising research direction and we leave it for the future work too.

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bull. Symbolic Logic **16**(4), 457–515 (2010)
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321 (1996)
3. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. **27**(4), 786–818 (2005)
4. Ball, T., Rajamani, S.: Boolean programs: a model and process for software analysis. Technical report MSR-TR-2000-14, Microsoft Research (2000)
5. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000). doi:10.1007/10722468_7
6. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL, pp. 1–3 (2002)
7. Basler, G., Hague, M., Kroening, D., Ong, C.-H.L., Wahl, T., Zhao, H.: Boom: taking boolean program model checking one step further. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 145–149. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12002-2_11
8. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31980-1_40
9. Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous boolean programs. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 75–90. Springer, Heidelberg (2005). doi:10.1007/11537328_9
10. Delzanno, G., Raskin, J.-F., Begin, L.: Towards the automated verification of multithreaded Java programs. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 173–187. Springer, Heidelberg (2002). doi:10.1007/3-540-46002-0_13
11. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000). doi:10.1007/10722167_20
12. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001). doi:10.1007/3-540-44585-4_30
13. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). doi:10.1007/3-540-63166-6_10
14. Holzmann, G.J.: The model checker Spin. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
15. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14295-6_55

16. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst. Sci. **3**(2), 147–195 (1969)
17. La Torre, S., Parthasarathy, M., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222 (2009)
18. Liu, P.: http://www.ccs.neu.edu/home/lpzun/ijit/
19. Liu, P., Wahl, T.: Infinite-state backward exploration of Boolean broadcast programs. In: FMCAD, pp. 155–162 (2014)
20. Liu, P., Wahl, T.: IJIT: an API for Boolean program analysis with just-in-time translation (extended technical report) (2017). CoRR arXiv.org/abs/1706.03167
21. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)