Concurrent Bounded Model Checking

Quoc-Sang Phan\textsuperscript{1}, Pasquale Malacaria\textsuperscript{1}, and Corina S. Păsăreanu\textsuperscript{2}

\textsuperscript{1} Queen Mary University of London
\textsuperscript{2} Carnegie Mellon Silicon Valley, NASA Ames

Abstract. We introduce a methodology, based on symbolic execution, for Concurrent Bounded Model Checking. In our approach, we translate a program into a formula in a disjunctive form. This design enables concurrent verification, with a main thread running symbolic execution, without any constraint solving, to build sub-formulas, and a set of worker threads running a decision procedure for satisfiability checks.

We have implemented this methodology in a tool called JCBMC, the first bounded model checker for Java. JCBMC is built as an extension of Java Pathfinder, an open-source verification platform developed by NASA. JCBMC uses Symbolic PathFinder (SPF) for the symbolic execution, Z3 as the solver and implements concurrency with multi-threading.

For evaluation, we compare JCBMC against SPF and the Bounded Model Checker CBMC. The results of the experiments show that we can achieve significant advantages of performance over these two tools.

1 Introduction

Model checking techniques are often classified in two categories: explicit-state or symbolic, depending on how they process the states of the system. While explicit-state model checking enumerates all possible states of the system explicitly, possibly on-the-fly \cite{1,2}, symbolic model checking represents sets of states symbolically, and hence more efficiently, by using Binary Decision Diagrams \cite{3} or Boolean formulae \cite{4}. SAT or SMT-based Bounded Model Checking (BMC) \cite{4} unwinds the transition relation of a program for a fixed number of steps $k$ and checks whether a property violation can occur in $k$ or fewer steps. This bounded verification is reduced to a satisfiability check performed by a SAT or SMT solver. BMC is widely used in the hardware industry.

For software, the application of BMC for ANSI-C is embodied in CBMC \cite{5,6}, which has been successfully used for many practical applications \cite{7}. In CBMC a C program containing assertions is translated into a formula (in Static Single Assignment form) which is then fed to a SAT or SMT solver to check its satisfiability. A satisfying assignment indicates that an error was found.

Bounded model checking has not been explored so far for many other languages, including Java. However, explicit-state model checking tools such as Java PathFinder (JPF) \cite{1} have been successfully used for the verification of many Java applications. Furthermore, there has been an explosion of symbolic execution \cite{8} tools that have been used successfully for test case generation and
error detection in the context of many high level languages, for example [9–12].
In particular, relevant for the work reported here, Symbolic PathFinder (SPF) [13] is a symbolic execution tool built as an extension of JPF, that provides a symbolic analysis for Java programs involving multi-threading and complex data structures.

In this paper, we describe a new methodology for BMC which is based on “classical” symbolic execution (SE) in the sense of King [8]. Note that the way CBMC transforms a program into Static Single Assignment form can also be viewed as executing the program symbolically. However, this encoding is different from the Symbolic Execution of King and we evaluate the two encodings as part of the work reported here. Our methodology is not language specific and only relies on a symbolic executor for that language and an SMT solver.

By using symbolic execution we obtain a translation of a program and assertions into a disjunctive formula encoding the path conditions for each bounded (complete) path explored in the code. This suggests a simple concurrent verification strategy that relies on the observations that any subset of disjuncts in a disjunction $F = F_1 \lor \cdots \lor F_n$ can be separately checked for satisfiability and whenever a subset is found to be satisfiable the satisfiability task can be stopped. Hence the verification of disjunctive formulas is naturally parallelizable.

We have implemented this methodology in a tool JCBMC, a Java Concurrent Bounded Model Checker, which uses SPF to generate the disjunctive formula from the code (constraint solving is turned off in SPF itself) and while generating the formula it sends sub-formulas to multiple worker threads for satisfiability checking. JCBMC handles programs with multi-threading and recursive input data structures and relies on a standard SMT solver, namely Z3 [14] for solving the constraints. Other solvers can easily be incorporated. One can even use different solvers for solving different path constraints in parallel.

Although JCBMC is only a prototype, and concurrency is implemented by multi-threading but not parallelized yet, its performance, compared with existing tools, i.e. Symbolic PathFinder and CBMC, is remarkable. We summarize our contributions as follows:

- A methodology for concurrent bounded model checking that is based on “classical” symbolic execution and it is naturally parallelizable.
- The methodology is language independent and supports assume-guarantee reasoning.
- A tool JCBMC, a concurrent bounded model checker for Java.
- Experiments to show effectiveness of the tool for verification of programs with multi-threading and data structures.
- Comparisons with bounded model checking and “classical” symbolic execution, as embodied by CBMC and SPF respectively.

The rest of the paper is organized as follows. We start describing our approach with an illustrative example in Section 2. In Section 3 we provide a brief introduction to CBMC and SPF. Section 4 presents the main contribution: we first propose an approach for BMC based on SE, and compare it with traditional BMC and SE. We then show how to parallelize it to build a Concurrent
# Illustrative example

We illustrate our approach using the simple example program in Fig. 1. We want to check if the assertion in line 9 is valid for all possible inputs. Note that an analysis using an explicit state model checker such as JPF would not be feasible, as this would involve enumerating all the possible inputs to the program.

Fig. 2 illustrates part of classical SE with constraints solving on the program following the path \((1, 2, 3, 4, 6, 7)\). This analysis could be performed using e.g. Symbolic PathFinder. Instead of concrete values, SE takes the symbols \(x_0\) and \(y_0\) as inputs and executes them just like concrete values. It also keeps track of the path condition \(pc\) which consists of the conditions true along that path and the symbolic environment \(\sigma\) which maps variables into expressions over the input symbols \(x_0, y_0\). Typically, whenever the \(pc\) is updated, SE checks the satisfiability of \(pc\) using an off-the-shelf solver.

Initially, \(pc\) is true, and \(\sigma\) maps inputs to theirs symbols. When SE reaches line 2, it updates \(pc\) as \((x_0 > 5)\) since this is the condition to reach line 3 where \(\sigma\) becomes \(x \mapsto x_0\). In line 4, the condition \((x < 3)\) is translated in \(\sigma\) to \(c \equiv (x_0 + 1 < 3)\). At this point SE calls an SMT solver, and detects that \(pc \models \neg c\) because \((x_0 > 5) \models \neg (x_0 + 1 < 3)\), therefore it jumps to line 6 with \(pc\) unchanged.

In our approach for BMC, SE plays the role of generating the formula which encodes the program behaviour and the property to be checked. The satisfiability of the resulting formula will be checked separately by an SMT solver. Therefore, we execute SE without invoking constraint solving whenever \(pc\) is updated, and we postpone checking the \(pc\) until the end of the execution path. In this way, we can save the execution time of calling the solver, but the trade-off is that infeasible paths are also included. However, this will not affect the soundness of the analysis, since constraint solving is performed later. When SE reaches line 9 following the path \(\{1, 2, 3, 4, 6, 7\}\), we have:

\[
pc = (x_0 > 5) \land \neg (x_0 + 1 < 3)
\]
Here we reach the property $\mathcal{P}$ to verify, which is $(x < 10)$. We denote by $\mathcal{P}|_\sigma$ the evaluation of $\mathcal{P}$ in the symbolic environment $\sigma$. At this point, $\sigma$ maps $x$ to $x + 1$, which leads to the following:

$$\mathcal{P}|_\sigma = (x_0 + 1 < 10)$$

The property $\mathcal{P}$ is violated in this path if we can find a model for

$$pc \land \neg \mathcal{P}|_\sigma = (x_0 > 5) \land \neg(x_0 + 1 < 3) \land \neg(x_0 + 1 < 10)$$

Setting $x_0 = 11$ will provide such a model. The whole formula generated by our approach for the code in Fig. 1 is:

$$( (x_0 > 5) \land \neg(x_0 + 1 < 3) \land \neg(x_0 + 1 < 10)) \lor ( (x_0 > 5) \land (x_0 + 1 < 3) \land \neg(x_0 < 10)) \lor ( \neg(x_0 > 5) \land \neg(x_0 < 10))$$

In general, we use SE to explore all possible symbolic paths up to a certain length, and then encode the program together with the property to check into a formula of the form:

$$\bigvee_{i=0}^{M} (pc_i \land \neg \mathcal{P}|_{\sigma_i})$$

where $N$ is the number of paths that may trigger the error. This form allows us to divide the formula into blocks of $D$ disjunctions:

$$\bigvee_{i=0}^{D-1} (pc_i \land \neg \mathcal{P}|_{\sigma_i}) \lor \bigvee_{i=D}^{2D-1} (pc_i \land \neg \mathcal{P}|_{\sigma_i}) \ldots \lor \bigvee_{i=(k-1)D}^{kD-1} (pc_i \land \neg \mathcal{P}|_{\sigma_i}) \lor \bigvee_{i=kD}^{M} (pc_i \land \neg \mathcal{P}|_{\sigma_i})$$

In this way, we can solve the formula concurrently using several threads, each one solving a single block. A model of a single block is also a model of the formula, therefore the procedure stops when any of the threads find out a model. In JCBMC, after the main thread generates a sub-formula and passes it to a worker thread, it moves on to generate the next sub-formula, while the worker thread solves the given sub-formula concurrently.

### 3 Preliminaries

For simplicity, we illustrate our methodology using the guarded command (GC) language [15] instead of Java. The grammar of the language is depicted in Fig. 3.

We focus here on safety properties: note that the two commands $\text{assume}(c)$ and $\text{assert}(c)$ are powerful enough to encode expressive temporal properties [5], and also support assume-guarantee style compositional reasoning. A program $P$ is modelled as a transition system:

$$P = (S, I, F, T)$$

(1)
program $\equiv$ stmt$
$ stmt $s$ $\equiv$ assume $e$ | assert $e$ | $v = e$ | if $e$ then goto $s$ else goto $s$
$v$ $\in$ Var (variables)
$e$ $\in$ Exp (expressions)

Fig. 3. The guarded command language

where $S$ is the set of program states; $I \subseteq S$ is the set of initial states; $F \subseteq S$ is the set of final states; and $T \subseteq S \times S$ is the transition relation. Under this setting, a trace of (a concrete) execution of the program $P$ is represented by a sequence of states:

$$\rho = s_0s_1..s_k$$

such that $s_0 \in I$, $s_k \in F$ and $\langle s_i, s_{i+1} \rangle \in T$ for all $i \in \{0, .., k - 1\}$.

3.1 Bounded Model Checking and CBMC

A trace can be also seen in logical form: the set $I$ and the relation $T$ can be written as their characteristic functions: $s_0 \in I$ iff $I(s_0)$ holds; $\langle s_i, s_{i+1} \rangle \in T$ iff $T(s_i, s_{i+1})$ holds. In this way, a trace $\rho$ is represented by the formula:

$$I(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \quad (2)$$

Clearly the transition system $P$ is a model for such a formula, i.e. $P$ is a model for all formulas representing traces of the program. The aim of BMC is to find bugs or prove their absence up to some bounded $k$ number of transitions. That means it explores all traces $\rho = s_0s_1..s_k$ of the program $P$, in which $s_k$ needs not to be in $F$. Notice that because of the bound $k$ there are only a finite number of traces to explore hence we can represent the bounded program as a formula $C$ which is a conjunction of formulas, whose conjoints are possible traces. Notice formulas can also represent symbolic traces, for example if in a formula the value of a program variable is left unspecified then there can be several concrete traces satisfying that formula. Formulas satisfied by set of concrete traces can be referred to as symbolic traces.

CBMC translates a C program into a logical formula $C$ which is then used as a model for the property $P$ to be verified. The property is verified by the C program iff $C \land P$ is valid. This can be checked by a satisfiability solver on $C \land \neg P$. In fact if $C \land \neg P$ is true in the model then one trace will satisfy $\neg P$ hence the property is not valid. On the other hand if $C \land \neg P$ is false in the model then no trace will satisfy $\neg P$ hence $P$ is valid.

3.2 Symbolic Pathfinder: Symbolic Execution for Java Bytecode

Symbolic PathFinder (SPF) is a symbolic execution framework built on top of the Java PathFinder (JPF) model checking tool-set for Java bytecode analysis.
SPF implements a Bytecode interpreter that replaces the standard, concrete execution semantics of bytecodes with a non-standard symbolic execution. Non-deterministic choices in branching conditions are handled by means of JPF’s choice generators. Each non-deterministic choice has associated a path condition. JPF’s listeners are used to monitor and influence the symbolic execution and to collect and print its results. Symbolic execution of looping programs may result in an infinite symbolic execution tree; for this reason, SPF is run with a user-specified bound on the search depth. By default, whenever the path condition is updated, SPF invokes an off-the-shelf solver to check its satisfiability; if the path condition is found to be unsatisfiable, SPF backtracks. As a result, by default, SPF explores only feasible paths. Note that SPF also has an option to run with no solving (that we use in our work here); as a result of this option, SPF will explore all the possible paths (feasible and infeasible) through the program, up to the given bound. SPF uses lazy initialization [16] to handle dynamic input data structures (e.g., lists and trees). Multi-threading is handled systematically using the search mechanisms in JPF core.

4 Concurrent Bounded Model Checking

Our method for concurrent bounded model checking is illustrated in Fig. 4. The inputs to the method are: a program under test, a property to verify and three parameters – $B$ is the search bound, $N$ is the number of workers and $D$ is the number of disjuncts to give to one worker. The goal is to check if the property holds in the program, up to exploration bound $B$.

4.1 Bounded Model Checking based on Symbolic Execution

The program under test is analysed using “classical” bounded symbolic execution with constraint solving turned off. This means that whenever a path condition is updated, we do not check its satisfiability, but rather continue the exploration. As a result, symbolic execution may explore infeasible paths, which will be checked later using constraint solving. Our approach can be used for the
bounded verification of safety properties, which we assume have been reduced to checking assertions embedded in the code. Furthermore, our method supports both assume and assert statements to enable assume-guarantee style verification. The assumed conditions are simply added to the path conditions during the symbolic execution.

The result of symbolic execution is a disjunction of path conditions, encoding constraints on the inputs to follow those paths, up to the pre-specified search bound. From among these paths, only the ones that may lead to assert violations are selected for solving. This is achieved by the controller which collects sets of $D$ violating path conditions and sends them for solving to parallel worker threads, using off-the-shelf solvers. The workers start solving as soon as they receive the disjunctive formulas, which may happen while the symbolic execution is still exploring the program. The verification terminates as soon as one of the threads finds a satisfying assignment, in which case an error is reported, or when all the disjunctions are found to be unsatisfiable, in which case the assertion holds (no error) up to the given bound. Note that if the symbolic execution discovers no potentially violating paths (i.e. the error is unreachable), then no solving will be performed.

4.2 Comparing our approach with BMC and SE

Compared with classical BMC we use an explicit enumeration of paths, while BMC uses an implicit enumeration of paths. Although at first glance the implicit encoding should be better our experiments, even with sequential JCBMC, show that this is not the case. Furthermore, the explicit enumeration is easily parallelizable, with simple and natural load balancing for different threads. Crucially our approach stops as soon as a path leading to an error is found to be satisfiable, while with classical BMC, all the program needs to be explored.

Compared with classical symbolic execution: we solve only in the end. So obviously the price to be paid is the exploration of infeasible paths. On the other hand, again, it is naturally parallelizable and constraint solving, which is one of the bottlenecks in SE, can be done in parallel, even with different solvers, with little coordination, if any, needed.

In the following we describe in more detail our method. We start with a description of a sequential approach, to clarify how we use symbolic execution to built a disjunctive formula of the path conditions. Solving this formula happens after the symbolic execution, sequentially.

4.3 Sequential Verification

Our approach of employing SE for BMC is based on a simple observation. Suppose $s_k$ is an error state in the transition system $P$. To determine the reachability of $s_k$ from the initial state $s_0$, BMC builds a series of transitions $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$, resulting in the formula in (2) of Section 3.1. On the other hand, SE builds the path condition $pc$ which is a first order formula also
characterising reachability of $s_k$ from $s_0$. Therefore, both condition (2) and $pc$ characterize all the inputs that reach the error.

Suppose we want to prove formally that the program $P$ satisfies, within given bounds, a property $P$ represented by a set of assertions. This means whenever the program reaches an assertion point, the assertion needs to be valid, which means: $\bigwedge (pc_i \rightarrow \mathcal{P}|_{\sigma_i})$. This can be verified by checking the satisfiability of $\bigvee (pc_i \land \neg\mathcal{P}|_{\sigma_i})$.

Proof. First notice one formula is the negation of the other one, i.e.

$$\bigwedge (pc_i \rightarrow \mathcal{P}|_{\sigma_i}) = \bigwedge (\neg pc_i \lor \mathcal{P}|_{\sigma_i}) = \bigwedge (\neg(pc_i \land \neg\mathcal{P}|_{\sigma_i}) = \neg \bigvee (pc_i \land \neg\mathcal{P}|_{\sigma_i})$$

hence if $\bigvee (pc_i \land \neg\mathcal{P}|_{\sigma_i})$ is satisfiable there exists an $i$ such that $pc_i \land \neg\mathcal{P}|_{\sigma_i}$ is satisfiable i.e. an initial state leading to the path $pc_i$ and not satisfying the property $\mathcal{P}|_{\sigma_i}$. On the other hand if $\bigvee (pc_i \land \neg\mathcal{P}|_{\sigma_i})$ if not satisfiable then no disjunct $pc_i \land \neg\mathcal{P}|_{\sigma_i}$ is satisfiable hence there exists no initial state leading to an execution path satisfying the assertion.

The algorithm in Fig. 5 shows how to build the formula $\Gamma = \bigvee (pc_i \land \neg\mathcal{P}|_{\sigma_i})$ for the GC language described in the previous section. In essence, the algorithm runs classical SE with no constraint solving for checking $pc$ satisfiability. A statement of GC is determined by its location $l$, and the function $\text{next}(l)$ returns the location of the next statement. An if-statement consists of a condition $c$, the location $l_\top$ if $c$ is true, and the location $l_\bot$ if $c$ is false. In the recursive procedure $\text{SymExBMC}$ as well as the function $\text{Process}$, all parameters are passed by value. In $\text{SymExBMC}$, the checking from line 6 to line 7 is only used in concurrent mode. In sequential mode, $\text{isSAT}$ is set to false, and it is not changed in the whole procedure. $\Gamma$ is declared as a global variable, initialised to false (denoted by $\bot$). $k$ is global constant, defining the bound of BMC.

Similar to standard SE, at the beginning the symbolic environment $\sigma$ maps program inputs to symbols, the path condition $pc$ is initialised to true (denoted by $\top$). The depth $i$ of the recursion is initialised to 0. The procedure $\text{SymExBMC}$ starts by ensuring the search depth $i$ not to reach the bound $B$ (line 2 to line 3). As from line 4 to line 16, it symbolically executes a basic block, i.e. without branching statement, of the program. The basic block ends by an if-statement or when the location $l$ reaches the end of the source file ($l = EOF$). Assumptions and assertions are evaluated by the current symbolic environment (line 9, 11). When there is an assignment, the symbolic environment updates the mapping for the variable in the left hand side by the evaluation of the right hand side. At the end of the block, if there is an if-statement at the next location, $\text{SymExBMC}$ is recursively called for both the $\text{then}$ path and the $\text{else}$ path. The condition is added to the path conditions without any checking of path feasibility.

### 4.4 Concurrent Verification

The simple algorithm presented in the previous sections will essentially enumerate all the possible feasible and infeasible paths through a program (up to a
function SymExBMC(\(\sigma, pc, l, i\))

1: if \((i > B)\) then
2: return
3: Extract statement \(s\) at \(l\)
4: while \((s\) is not an if-statement \& \(l \neq EOF\)) do
5: if \((\text{isSAT} = \top)\) then
6: return
7: if \((s\) is ‘assume \(c\)’) then
8: \(pc \leftarrow pc \land c|_\sigma\)
9: else if \((s\) is ‘assert \(c\)’) then
10: \(\text{Process}(pc \land \neg c|_\sigma)\)
11: else
12: Execute the assignment \(s\), update \(\sigma\)
13: \(l \leftarrow \text{next}(l)\)
14: Extract statement \(s\) at \(l\)
15: end while
16: if \((l = EOF)\) then
17: return
18: \(\{c, l_\top, l_\bot\}\) from if-statement
19: \(i \leftarrow i + 1\)
20: \(pc_1 \leftarrow pc \land c|_\sigma\)
21: SymExBMC(\(\sigma, pc_1, l_\top, i\))
22: \(pc_2 \leftarrow pc \land \neg c|_\sigma\)
23: SymExBMC(\(\sigma, pc_2, l_\bot, i\))
24: end function

function Process(\(\gamma\))

\(\Gamma \leftarrow \Gamma \lor \gamma\)

end function

Fig. 5. Formula generation for BMC

Fig. 6. Process error paths

given bound), collect the path conditions for each path into a formula in disjunctive form and then invoke a constraint solver to check the satisfiability, all at once. The disadvantage of the approach is that it needs to enumerate all the possible paths through the program, which can quickly become expensive, especially if multi-threading is also considered. We therefore propose a concurrent algorithm that parallelises SymExBMC by delegating the satisfiability check of the disjuncts in \(\Gamma\) to worker threads. The concurrency of the algorithm relies on two parameters: the number of concurrent workers available and the number of disjuncts sent to each worker: the optimal choice is architecture and SMT solver dependent. In our experiments we found 200 disjuncts to be a reasonable choice.

The main function is in Fig 7. It initialises \(\Gamma, pc, \sigma\) exactly the same as in sequential mode of SymExBMC. Here, \(\text{isSAT}\) is a boolean variable shared between the threads, and can be modified (set to true) by them. \(d\) is also a global variable of the main thread only to keep the number of current disjuncts. After initializing, the function SymExBMC is called. The main difference between sequential and concurrent mode is the function Process in Fig. 9 and the checking from line 6 to line 7 in SymExBMC. In sequential mode, \(\text{isSAT}\) is always false, so SymExBMC keeps building the formula until it reaches EOF. In concurrent mode, when the number of disjuncts in \(\Gamma\) reaches a bound \(B\), Process sends \(\Gamma\) for satisfiability
Γ, isSAT ← ⊥; pc ← ⊤; d, i ← 0
l ← first statement
SymExBMC(σ, pc, l, i)
if (i > 0 ∧ isSAT = ⊥) then
    Execute Run(Γ, isSAT) in worker thread
if (isSAT = ⊥) then
    Return Verification successful
else
    Return Verification failed
end function

function Run(Γ, isSAT)
Run SMT-Solver on Γ
if (Γ has a model) then
    isSAT = ⊤
end function

Fig. 7. Main thread

function PROCESS(γ)
Γ ← Γ ∨ γ
d ← d + 1
if (d ≥ D) then
    Execute Run(Γ, isSAT) in worker thread
    Γ ← ⊥; d ← 0
end function

Fig. 8. Worker thread

Fig. 9. Process error paths for Concurrent BMC

check to a worker thread. Crucially this worker thread can run in parallel to any other running thread as they run completely independent tasks.

Whenever a worker thread finds a model for its own Γ it sets the shared variable isSAT to true which will return control to the main thread and end the computation with Verification failed. If no thread sets isSAT to true then SymExBMC will eventually terminate and the remaining disjuncts (whose number is hence less than D) are sent for satisfiability check to a final thread. Verification successful is returned only if no thread has set isSAT to true during the computation.

Implementation. Our prototype tool, JCBMC, is implemented following the Observer Design Pattern [17]. SPF executes symbolically the Java bytecode program and acts as the subject. The Controller acts as the observer, it waits for SPF to have generated a sub-formula Γ with D disjuncts and then sends it to an available worker thread. D is a user chosen parameter of JCBMC. The worker thread executes Run(Γ, isSAT) which first writes Γ into a file in SMT2 format [18], then calls the SMT solver Z3 to check for satisfiability. JCBMC creates a thread pool of N workers; current architecture doesn’t support parallelism but only multi-threading.

JCBMC is built on top of SPF and as an extension of the JPF platform, therefore it inherits all the power of JPF and SPF as described in section 3.2 e.g. lazy initialization, multi-threading handling etc.
5 Evaluation

Our evaluation comprises cases studies to compare JCBMC with SPF (default configuration) and case studies to compare JCBMC with CBMC\(^3\). To compare with CBMC we have considered C code whose Java translation is almost literal.

An important reminder is that the current implementation of JCBMC is multi-threaded but not yet parallel and we expect a parallel implementation to have a significant advantage over the current one. By JSBMC we will denote the sequential implementation of JCBMC where a single thread is used. Experiments are run on a machine equipped with dual Xeon(R) E5-2670 CPUs. The results are shown in Tables 10 and 11. Unless otherwise specified times are in seconds, \(x\)\(y\) means \(x\) minutes and \(y\) seconds, “timed out” is one hour and \(x\) denotes a memory hit\(^4\). The source code for the examples can be found at: http://www.eecs.qmul.ac.uk/~qsp30/test/bmc_test.tar.gz.

5.1 Comparing with CBMC and SPF

**Bubble Sort** We consider the classical bubble sort algorithm, which has already been studied in the BMC community \([19, 20]\). Here, differently from \([20]\), we consider the more challenging symbolic version where the values of the array are non-deterministically chosen. We consider both the verification of the assertion “the elements of the array are ordered after bubble sort” and its negation “the elements of the array are not ordered after bubble sort”. We analyse a program implementing bubble sort. It will hence contain no bugs for the positive assertion and will be buggy for the negation. Results are shown in Fig 10. We notice that while CBMC is better for the positive assertion, JCBMC outperforms the other tools for the negative assertion and is capable of find a counterexample for array sizes of a higher order of magnitude.

**Sum of array** We consider the array case studies taken from \([19]\), in particular `sum_array_safe.c` for verification and `sum_arrayUnsafe.c` for refutation. The array size is set to 1000. Results show both SPF and JCBMC outperform CBMC for the unsafe version, while CBMC has a slight advantage for the safe version.

5.2 Comparing with SPF (Java code)

The following examples consist of substantial Java code which is not naturally translatable in C; we hence compare JCBMC only with SPF. Notice JCBMC and SPF are both extensions of JPF: in the case all inputs are concrete they both reduce to JPF-core hence their performance is identical. Hence we only consider programs with symbolic inputs.

**Flap controller** This case study is shipped with the distribution of SPF. It is a multi-threaded program modelling a simplified flap controller on an aircraft.

---

\(^3\) To compare both tools with the same solver in the experiments CBMC will be called with option `-smt2`, and we will use Z3 for satisfiability checks.

\(^4\) A memory hit is a “run out of memory” problem. This can be addressed by a different memory manager in JPF or with a direct implementation of SymExBMC.
### Array size

<table>
<thead>
<tr>
<th>SPF</th>
<th>JSBMC</th>
<th>CBMC</th>
<th>JCBMC (D = 10)</th>
<th>JCBMC (D = 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>5</strong></td>
<td>4.517</td>
<td>2.174</td>
<td>0.460</td>
<td>1.097</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>5.622</td>
<td>12.604</td>
<td>0.817</td>
<td>1.160</td>
</tr>
<tr>
<td><strong>15</strong></td>
<td>36.194</td>
<td>x</td>
<td>56m34.033</td>
<td>1.195</td>
</tr>
<tr>
<td><strong>30</strong></td>
<td>4m32.790</td>
<td>x</td>
<td>timed out</td>
<td>1.387</td>
</tr>
<tr>
<td><strong>100</strong></td>
<td>timed out</td>
<td>x</td>
<td>timed out</td>
<td>4.944</td>
</tr>
</tbody>
</table>

**Verification of bubble sort**

<table>
<thead>
<tr>
<th>SPF</th>
<th>JSBMC</th>
<th>CBMC</th>
<th>JCBMC (D = 10)</th>
<th>JCBMC (D = 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>5</strong></td>
<td>6m19.222</td>
<td>3.712</td>
<td>7.171</td>
<td>4.193</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>timed out</td>
<td>26.293</td>
<td>37.816</td>
<td>29.512</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>x</td>
<td>x</td>
<td>5m22.641</td>
<td>x</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>x</td>
<td>x</td>
<td>timed out</td>
<td>x</td>
</tr>
</tbody>
</table>

**Sum of array**

<table>
<thead>
<tr>
<th>SPF</th>
<th>JSBMC</th>
<th>CBMC</th>
<th>JCBMC (D = 10)</th>
<th>JCBMC (D = 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unsafe</strong></td>
<td>1.403</td>
<td>12.671</td>
<td>1m5.738</td>
<td>1.576</td>
</tr>
<tr>
<td><strong>safe</strong></td>
<td>failed</td>
<td>12.030</td>
<td>2.252</td>
<td>9.466</td>
</tr>
</tbody>
</table>

**Fig. 10.** Performance of all tools. “failed” refers to SPF failing to solve the constraints using the integrated solver.

<table>
<thead>
<tr>
<th>Tool</th>
<th>SPF</th>
<th>JSBMC</th>
<th>JCBMC (D = 10)</th>
<th>JCBMC (D = 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flap controller (unsafe)</td>
<td>1.141</td>
<td>2.899</td>
<td>0.948</td>
<td>1.370</td>
</tr>
<tr>
<td>Red-black tree (safe)</td>
<td>53.602</td>
<td>3.942</td>
<td>3.267</td>
<td>2.774</td>
</tr>
<tr>
<td>MER Arbiter (unsafe)</td>
<td>5.275</td>
<td>8.111</td>
<td>7.479</td>
<td>7.579</td>
</tr>
<tr>
<td>MER Arbiter (safe)</td>
<td>47.065</td>
<td>59.145</td>
<td>57.740</td>
<td>58.886</td>
</tr>
</tbody>
</table>

**Fig. 11.** Performance on Flap controller, Red-black tree and MER Arbiter.

It contains 3 classes, and 80 lines of code. This example demonstrates handling of multi-threading.

**Red Black Tree** This is another example from the SPF distributions (3474 LOC in one class). We check for consistency of the tree after performing put, remove, get and firstKey symbolically. Results show that both JSBMC and JCBMC significantly outperform SPF.

**MER Arbiter** The MER Arbiter models a component of the flight software for NASA JPL’s Mars Exploration Rovers (MER). The MER Arbiter has been modelled in Simulink/Stateflow and it was automatically translated into Java using the Polyglot framework and analyzed with SPF [21]. The configuration for our analysis involved two users and five resources. The example has 268 classes, 553 methods, 4697 lines of code (including the Java Polyglot execution framework) but only approx. 50 classes are relevant. We analyse the code with and without the error (see [21]). The performances of SPF, JSBMC and JCBMC are comparable, with SPF slightly better.

**Discussion** Compared to CBMC, JCBMC scores better in finding counterexamples than in verifying their absence; this is consistent with its design because a counterexample corresponds to a worker thread finding a model of the formula. Compared with SPF, JCBMC can be much better (see bubble sort or red
black tree) but can also be comparable or slightly worse (see MER Arbiter and Flap Controller results). The reason for the latter is that the cost of generating path conditions dominates the cost of solving them. Similarly, SPF failed to generate formulas for bubble sort for sizes 7 and higher. Furthermore, an error path (e.g. in MER) may occur at the beginning of the SE exploration, and it is therefore discovered quickly by SPF, while JCBMC still needs to generate the pre-specified number $D$ of error paths before solving them. The results suggest one direction for future work, namely to investigate improving the cost of SE-based path generation (see last section).

6 Related Work

Related approaches on parallelising BMC [22, 23] address parallel solving of the conjunctive formula that is built for BMC and aim at performing solving at different bounds, where some clauses are shared to enable more efficient SAT solving. In contrast we aim to solve the formulas generated with SE for the same bound, which are naturally disjoint resulting in a simpler parallel algorithm. Furthermore our work aims at verifying programs written in high-level languages such as Java and it is not clear how the previous work, performed in the context of finite state automata, would be applicable. Also related is the work on parallel SAT and SMT solving [24–26], which can be seen complementing the work presented here, in the sense that we can use e.g. the parallel version of Z3 [26] in each of the workers to further speed up our proposed approach.

PKIND [27] is a parallel model checker for Lustre that uses k-induction. PKIND runs in parallel the different tasks involved in performing the induction, e.g. the base step, the induction step and also the generation of auxiliary invariants used for verification. Therefore PKIND performs the parallel work at a higher level of granularity than JCBMC. It would be interesting to investigate if we can replace the parallel tasks in PKIND with our own version of SE-based bounded verification, which in turn is parallelized at the level of granularity of symbolic paths.

Parallel model checking has been investigated in the context of explicit-state [28–33] and symbolic [34, 35] exploration. The latter were done in the context of verification using Binary Decision Diagrams, and hence are very different from ours. These approaches concentrate on partitioning the state space to be explored in parallel and on dealing with the communication overhead between parallel workers. In contrast, in our approach the workers perform the solving independently, with no communication between them.

In previous work we have developed a framework for performing parallel symbolic execution in SPF [36]. We used a set of pre-conditions to partition the symbolic execution tree to distribute its processing. These pre-conditions were computed a-priori, using a “shallow” symbolic execution up to a small exploration bound, to statically compute the different partitions of the input space with no communication overhead. Other approaches to parallel symbolic execution [37–39] operate primarily by dynamically partitioning the symbolic ex-
ecution tree for load balancing, which may result in better use of computational resources but also in more communication overhead. All these approaches were done in the context of “classical” [36, 37] or dynamic [38, 39] symbolic execution, using constraint solving during path generation. In contrast the approach we advocate here has a clear separation between path generation and constraint solving, allowing us to easily achieve load balancing between workers, with little communication overhead. It would be interesting to compare experimentally and to also combine the techniques in JCBMC and the parallel version of SPF and we plan to do that in future work.

7 Conclusion and Future Work

We have presented a language independent methodology for concurrent bounded model checking. Based on this methodology we have implemented a concurrent bounded model checker for Java. A first improvement would be to upgrade its concurrency from single CPU multi-threading to true parallelism and to perform obvious optimisations. Another improvement would be to replace SPF with a lighter weight tool, or a parallel version, to reduce the cost of generating path conditions. It will also be interesting to implement the methodology for other languages (C, Python) and investigate how to use SE for IC3 style verification.

References


